Operación	Lista enlazada	Lista enlazada ordenada	Árbol binario de búsqueda	Árbol AVL	Heap	Trie
Pertenencia	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(m)$
Inserción	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(m)$
Borrado	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(m)$
Búsqueda del mínimo	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$	$O(n)$
Borrado del mínimo	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(n)$

Explicaciones Adicionales

1. **Lista enlazada**:
   - **Pertenencia**: Debe recorrer toda la lista para encontrar un elemento.
   - **Inserción**: Puede insertar un nuevo nodo al inicio en $O(1)$.
   - **Borrado**: Necesita buscar el nodo antes de borrarlo, lo cual toma $O(n)$.
   - **Búsqueda del mínimo**: Debe recorrer toda la lista para encontrar el mínimo.
   - **Borrado del mínimo**: Debe encontrar el mínimo primero, lo cual toma $O(n)$.
2. **Lista enlazada ordenada**:
   - **Pertenencia**: Debe recorrer la lista, similar a la lista enlazada no ordenada.
   - **Inserción**: Debe encontrar la posición correcta para mantener el orden, lo cual toma $O(n)$.
   - **Borrado**: Similar a la lista no ordenada, necesita buscar el nodo.
   - **Búsqueda del mínimo**: El primer nodo siempre es el mínimo, por lo que es $O(1)$.
   - **Borrado del mínimo**: Borrar el primer nodo es $O(1)$.
3. **Árbol binario de búsqueda (BST)**:
   - **Pertenencia**: En el peor caso, el árbol puede degenerar en una lista enlazada, tomando $O(n)$.
   - **Inserción**: Similar a la búsqueda, puede ser $O(n)$ en el peor caso.
   - **Borrado**: Igual que la búsqueda e inserción, en el peor caso es $O(n)$.
   - **Búsqueda del mínimo**: Puede requerir recorrer todo el árbol en el peor caso, $O(n)$.
   - **Borrado del mínimo**: Debe encontrar el mínimo primero, lo cual es $O(n)$.
4. **Árbol AVL**:

1. **Selection Sort**
   - **Mejor caso**: $O(n^2)$
     - No tiene una ventaja en el mejor caso, siempre realiza el mismo número de comparaciones y movimientos.
   - **Peor caso**: $O(n^2)$
     - Siempre realiza el mismo número de comparaciones y movimientos independientemente del orden de los elementos.

2. **Insertion Sort**
   - **Mejor caso**: $O(n)$
     - Ocurre cuando la lista ya está ordenada. Solo se realizan comparaciones sin movimientos.
   - **Peor caso**: $O(n^2)$
     - Ocurre cuando la lista está ordenada en orden inverso. Cada elemento debe compararse y moverse a la posición correcta.

3. **Merge Sort**
   - **Mejor caso**: $O(n \log n)$
     - Siempre divide y conquista la lista, realizando el mismo número de operaciones independientemente del orden inicial.
   - **Peor caso**: $O(n \log n)$
     - La estructura divide y vencerás asegura que el número de operaciones sea consistente.

4. **Heap Sort**
   - **Mejor caso**: $O(n \log n)$
     - La estructura del heap asegura que la complejidad sea $O(n \log n)$ en todos los casos.
   - **Peor caso**: $O(n \log n)$
     - Similar al mejor caso, la complejidad es consistente debido a la naturaleza del heap.

5. **Quick Sort**
   - **Mejor caso**: $O(n \log n)$
     - Ocurre cuando el pivote divide las listas de manera balanceada.
   - **Peor caso**: $O(n^2)$
     - Ocurre cuando el pivote es el menor o el mayor elemento repetidamente, dividiendo la lista en una muy desbalanceada.

6. **Counting Sort**
   - **Mejor caso**: $O(n + k)$
     - Siempre lineal, ya que cuenta las ocurrencias de cada elemento.

- **Peor caso**: $O(n + k)$
  - La complejidad es consistente si k (rango de los valores) es razonablemente pequeño en comparación con n.

7. Radix Sort

- **Mejor caso**: $O(n \cdot d)$
  - Es lineal respecto al número de elementos y la longitud de los dígitos.
- **Peor caso**: $O(n \cdot d)$
  - Similar al mejor caso, siempre lineal respecto al número de elementos y la longitud de los dígitos.

8. Bucket Sort

- **Mejor caso**: $O(n + k)$
  - Ocurre cuando los elementos están distribuidos uniformemente entre los buckets.
- **Peor caso**: $O(n^2)$
  - Ocurre cuando todos los elementos caen en el mismo bucket, lo que requiere ordenar ese bucket usando un algoritmo de ordenación diferente.

Resumen

Algoritmo	Mejor caso	Peor caso
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n^2)$
Counting Sort	$O(n + k)$	$O(n + k)$
Radix Sort	$O(n \cdot d)$	$O(n \cdot d)$
Bucket Sort	$O(n + k)$	$O(n^2)$

Explicaciones Adicionales

- **Selection Sort**: Siempre selecciona el menor elemento de la lista desordenada y lo coloca en la posición correcta. El número de comparaciones no cambia con el orden inicial de la lista.
- **Insertion Sort**: Muy eficiente para listas que ya están casi ordenadas.
- **Merge Sort**: Divide y conquista, garantizando un tiempo consistente de $O(n \log n)$.
- **Heap Sort**: Utiliza una estructura de heap para mantener el orden.
- **Quick Sort**: La elección del pivote es crucial para su rendimiento.
- **Counting Sort**: Eficiente para listas con un rango limitado de elementos.

1 Modulos Basicos

- LISTA ENLAZADA: listaVacia, longitud, vacia, agregarAdelante, agregarAtras,fin, comienzo, primero, ultimo O(1)
- obtener , eliminar , modificarPosicion O(n) , concatenar O(m).
- PILAsobreLista: pilaVacia , vacia, encolar, desencolar , tope O(1).
- COLAsobreLista: colaVacia, encolar, desencolar, proximo O(1).
- VECTOR: vectorVacio, longitud, vacia, primero, ultimo, obtener, modificarPosicion O(1)
- agregarAdelante, agregarAtras O(f(n)) , fin , comienzo, eliminar O(n) , concatenar O(m).
- **CONJLINEAL**: conjvacio, size, agregarRapido O(1).
- pertenece, agregar, sacar O(n) , unir restar intesecar O(n*m)
- **CONJLOG**:conjvacio , size O(1), pertenece, agregar, agregarRapido, sacar O(log(n)), unir , restar, intesecar O((n+m)*log(n+m))
- **DiccLineal**: diccionarioVacio ,size O(1), esta, definir, borrar, obtener O(n). $--$ > Se implementa con ListasEnlazadas
- **DiccLog**:esta, definir, obtener, definirRapido, obtener, borrar O(log(n)), size diccionarioVacio O(1). $--$ > Se implementa con AVL.
- **DiccDigital**: esta,definir,obtener,borrar O($|k|$) $-- >$ *Implementa trie*
- **ColaDePrioridadLog**: encolar O(log(n)) , desencolarMax O(log(n)) , cambiarPrioridad O(n).
- El m´odulo ColaDePrioridadLog implementa el TAD ColaPrioridad utilizando un Heap. Provee todas las operaciones de una cola de prioridad, inclu´ıdo cambiar la prioridad de un elemento. Es posible construir un ColaDePrioridadLog a partir de una secuencia en O(n) utilizando el algoritmo heapify En cuanto al recorrido de los elementos, se provee un iterador bidireccional que permite recorrer los elementos como si fuera una secuencia de pares ¡prioridad,valor¿. Las complejidades de las operaciones son las siguientes: