TADs y Estructuras de Datos Lineales. Conjuntos, Bolsas, Colas de prioridad y Diccionarios

Luis Llopis, 2024.

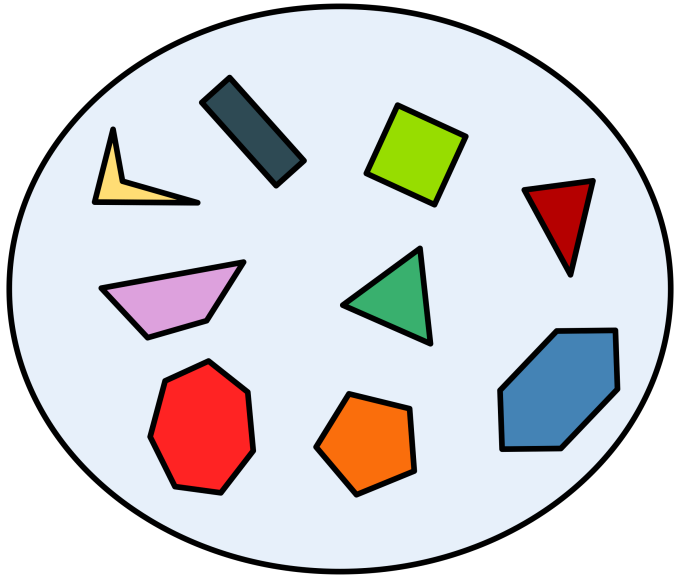
Dpto. Lenguajes y Ciencias de la Computación.

University of Málaga

UNIVERSIDAD
DE MÁLAGA | uma.es

El TAD conjunto

- Un *conjunto* es una colección que almacena elementos *sin duplicados*.
- A diferencia de una lista, los elementos de un conjunto *no tienen una posición* o índice.
- **Operaciones**:
  - `insert` : inserta un elemento en el conjunto. Si el elemento está ya en el conjunto, permanece inalterado.
  - `delete` : elimina un elemento del conjunto. Si el elemento no está en el conjunto, permanece sin cambios.
  - `contiene` : comprueba si un elemento está en el conjunto.
  - `isEmpty` : Comprueba si el conjunto está vacío.
  - `size` : Devuelve el número de elementos del conjunto.
  - `clear` : Elimina todos los elementos del conjunto.
  - `iterator` : Devuelve un `Iterador` para recorrer los elementos del conjunto.



Set of Polygons

El TAD Conjunto en Java: la interfaz

`Set<T>` define un conjunto con elementos de tipo `T`.

```
package dataStructures.set;

public interface Set<T> extends Iterable<T> {
  void insert(T element);
  void delete(T element);
  boolean contains(T element);
  boolean isEmpty();
  int size();
  void clear();
}
```

El TAD de conjunto ordenado en Java

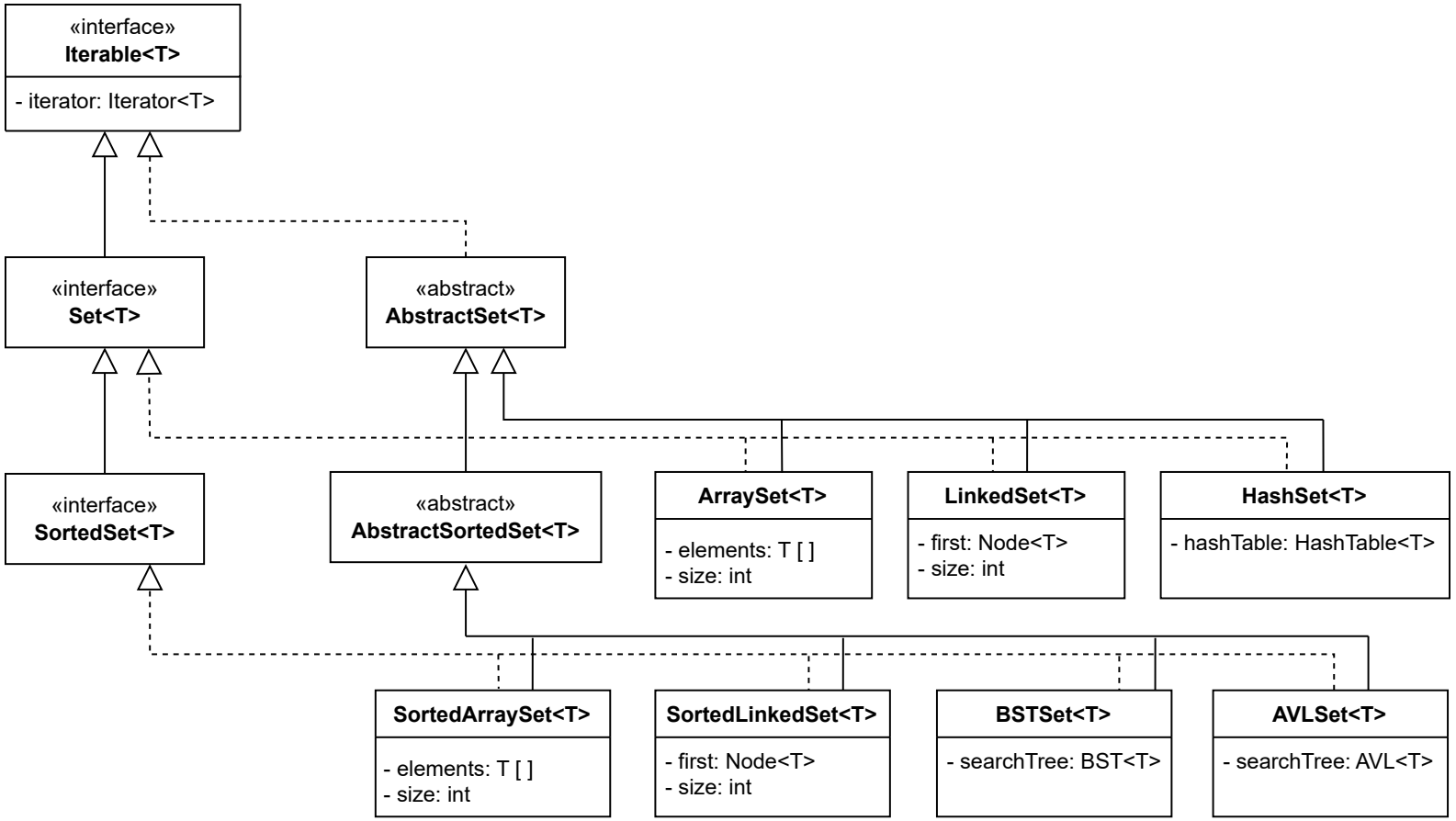
- Un *conjunto ordenado* es un conjunto donde los elementos se almacenan en un orden
- Cuando se itera un conjunto ordenado, los elementos se devuelven en orden.
- **Operaciones**:
  - `comparator`: Devuelve el comparador que define la *igualdad* y *ordenación* de elementos dentro del conjunto.
  - `minimum`: Devuelve el elemento más pequeño del conjunto, como determinado por el orden definido del comparador. Lanza una `NoSuchElementException` si el conjunto está vacío.
  - `maximum`: devuelve el elemento más grande del conjunto, según lo determinado por el orden definido en el comparador. Lanza una `NoSuchElementException` si el conjunto está vacío.

```
package dataStructures.set;

public interface SortedSet<T> extends Set<T> {
  Comparator<T> comparator();
  T minimum();
  T maximum();
}
```

Implementaciones de ADT de conjuntos y conjuntos ordenados

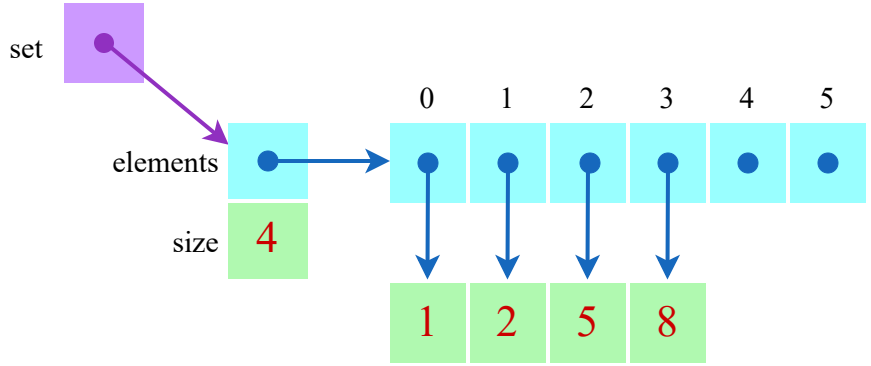
- Un conjunto se puede implementar utilizando diferentes *estructuras de datos*.
- Diferentes *clases* pueden implementar la interfaz `Set<T>`:
  - `ArraySet<T>`: utiliza un array sin ordenar para almacenar elementos.
  - `LinkedSet<T>`: utiliza una estructura enlazada no ordenada para almacenar elementos.
  - `HashSet<T>`: utiliza una *tabla hash* para almacenar elementos. La mayoría de las operaciones son O(1).
- Se puede implementar un conjunto ordenado utilizando diferentes *estructuras de datos*.
- Diferentes *clases* pueden implementar la interfaz `SortedSet<T>`:
  - `SortedArraySet<T>`: utiliza un array ordenado para almacenar elementos en orden. Puede utilizar la *búsqueda binaria* para acelerar las operaciones.
  - `SortedLinkedSet<T>`: utiliza una estructura enlazada ordenada para almacenar elementos en orden ordenado.
  - `BSTSet<T>`: utiliza un *árbol de búsqueda binaria* para almacenar elementos en orden.
  - `AVLSet<T>`: utiliza un *árbol AVL* para almacenar elementos en orden. La mayoría de las operaciones son O(log n).
- Las clases abstractas base `AbstractSet<T>` y `AbstractSortedSet<T>` proporcionan implementación para los métodos `equals`, `hashCode` y `toString`.

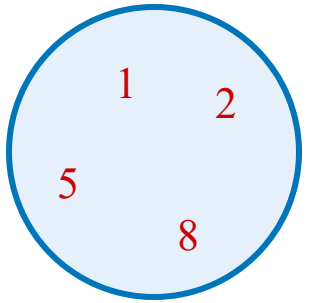
Implementaciones de ADT de conjuntos y conjuntos ordenados (II)



La clase `SortedArraySet`

- `SortedArraySet<T>` implementa la interfaz `SortedSet<T>` utilizando un Array *ordenada* para almacenar elementos.
- Inicialmente, la Array tiene un tamaño fijo (*capacidad* del conjunto), pero puede crecer dinámicamente cuando sea necesario.
- El constructor recibe un objeto `Comparator<T>` que se utiliza para determinar la igualdad y el orden de los elementos del conjunto.
- Los elementos se almacenan en la Array entre los índices 0 y `size -1`, *ordenado* según el comparador.
- A medida que se insertan nuevos elementos, se mantiene el orden.
- La clase también mantiene una variable entera `size` para realizar un seguimiento de la cantidad de elementos en el conjunto.





Implementación de `SortedArraySet`

```
package dataStructures.set;

public class SortedArraySet<T> extends AbstractSortedSet<T> implements SortedSet<T> {
  private static final int DEFAULT_INITIAL_CAPACITY = 6;

  private T[] elements;
  private Comparator<T> comparator;
  private int size;

  public SortedArraySet(Comparator<T> comparator, int initialCapacity) {  // SortedArraySet constructor
    if (initialCapacity <= 0) {
      throw new IllegalArgumentException("initial capacity must be greater than 0");
    }
    this.comparator = comparator;
    elements = (T[]) new Object[initialCapacity];
    size = 0;
  }

  public SortedArraySet(Comparator<T> comparator) {  // SortedArraySet constructor
    this(comparator, DEFAULT_INITIAL_CAPACITY);
  }

  ...
}
```

Métodos de fábrica para `SortedArraySet`

- `empty` : construye un *conjunto vacío* con una capacidad inicial *predeterminada* y ordenado según el *orden natural* de los elementos.
- `empty(Comparator<T> comparator)` : construye un *conjunto vacío* con una capacidad inicial *predeterminada* y ordenado según el comparador proporcionado.
- `withCapacity(int initialCapacity)` : construye un *conjunto vacío* con una capacidad inicial especificada y ordenado según el *orden natural* de los elementos.
- `withCapacity(Comparator<T> comparator, int initialCapacity)` : construye un *conjunto vacío* con una capacidad inicial especificada y ordenado según el comparador proporcionado.
- `of(T ... elementos)` : construye un conjunto *previamente rellenado* con los elementos proporcionados, ordenados según el *orden natural* de los elementos.
- `of(Comparator<T> comparator, T... elements)` : construye un conjunto *previamente rellenado* con los elementos proporcionados, ordenados según el comparador proporcionado.
- `copyOf(SortedSet<T> set)` : construye un nuevo conjunto que es un *duplicado* del `set` dado, preservando el orden de los elementos.
- `from(Iterable<T> iterable)` : construye un nuevo conjunto que contiene todos los elementos del *iterable* especificado, ordenados según el *orden natural* de los elementos.
- `from(Comparator<T> comparator, Iterable<T> iterable)` : construye un nuevo conjunto que contiene todos los elementos del *iterable* especificado, ordenados según el comparador proporcionado.

El algoritmo de búsqueda binaria

- El *algoritmo de búsqueda binaria* se utiliza para encontrar la posición de un elemento objetivo en un Array *ordenada*.
- En comparación con la *búsqueda lineal*, la búsqueda binaria es *mucho más eficiente* para matrices grandes (O(log n) vs O(n)).
- **Algoritmo**:
    i. Set `found` to `false`.
    ii. Set `left` to 0 and `right` to `size - 1`.
    iii. While `found` is `false` and `left` is less than or equal to `right`:
        - Set `mid` to the average of `left` and `right`.
        - If the target element is equal to the element at index `mid`, set `found` to `true`.
        - Else if the target element is greater than the element at index `mid`, set `left` to `mid + 1`.
        - Else, as the target element is less than the element at index `mid`, set `right` to `mid - 1`.
    iv. If found is `true`, the target element is at index `mid`. Otherwise, it should be inserted at index `left`.

La clase de utilidad `Finder` para `SortedArraySet`

- La clase interna `Finder` se puede utilizar para encontrar la posición de un elemento en la Array *ordenada* de elementos utilizando el *Algoritmo de búsqueda binaria*.

```
private final class Finder {
  boolean found;
  int index;

  Finder(T element) {
    found = false;
    int left = 0, right = size - 1, mid = 0;
    while (!found && left <= right) {
      mid = left + (right - left) / 2;
      int cmp = comparator.compare(element, elements[mid]);
      if (cmp == 0) {
        found = true;
      } else if (cmp > 0) {
        left = mid + 1;
      } else {
        right = mid - 1;
      }
    }
    index = found ? mid : left;
  }
}
```

- Tras la instanciación, el constructor de la clase `Finder` toma un `elemento` de destino y determina su `índice`, ya sea ubicándolo dentro de la Array ordenada o identificando el punto de inserción apropiado para mantener el orden de la Array.

Búsqueda, inserción y eliminación en `SortedArraySet`

- En todos los casos, se utiliza la clase `Finder` para encontrar la posición del elemento de destino en la Array ordenada.
- **Buscando**:
  - El método `contains` utiliza la clase `Finder` para determinar si el elemento de destino está en el conjunto.
- **Insertar**:
  - El método `insert` utiliza la clase `Finder` para encontrar la Posición donde debe insertarse el elemento de destino.
  - Si el elemento no está en el conjunto, los elementos en las posiciones `index` y los siguientes se desplazan hacia la derecha para dejar espacio para el nuevo elemento.
- **Eliminando**:
  - El método `delete` utiliza la clase `Finder` para encontrar el Posición del elemento objetivo.
  - Si el elemento está en el conjunto, los elementos en las posiciones `index + 1` y más allá se desplazan hacia la izquierda para llenar el espacio dejado por el elemento eliminado.

`unión`, `intersección` y `diferencia` en `SortedArraySet`

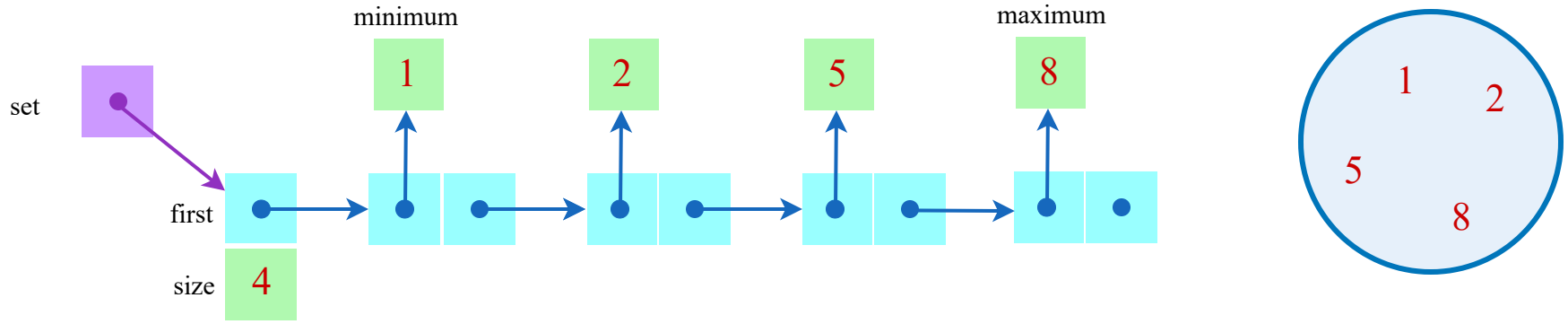
- Los métodos `unión`, `intersección` y `diferencia` se utilizan para realizar operaciones de conjuntos en dos conjuntos ordenados.
- **Unión**:
  - El método `union` devuelve un nuevo conjunto que contiene todos los elementos que se encuentran en cualquiera de los conjuntos.
- **Intersección**:
  - El método `intersection` devuelve un nuevo conjunto que contiene todos elementos que están en ambos conjuntos.
- **Diferencia**:
  - El método `difference` devuelve un nuevo conjunto que contiene todos los elementos que están en el primer conjunto pero no en el segundo.
  - Las implementaciones de estos métodos pueden *aprovechar* el orden de clasificación de los conjuntos para optimizar las operaciones.

- Complejidad computacional de las operaciones de `SortedArraySet`

Operation	Cost
`empty`	O(1)
`insert`	O(n)
`delete`	O(n)
`contains`	O(log n)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(n)

La clase `SortedLinkedSet`

- `SortedLinkedSet<T>` implementa la interfaz `SortedSet<T>` utilizando una estructura enlazada *sorted* para almacenar elementos.
- Los elementos se almacenan en nodos que están vinculados en orden.
- El constructor recibe un objeto `Comparator<T>` que se utiliza para determinar la igualdad y el orden de los elementos del conjunto.
- A medida que se insertan nuevos elementos, se mantiene el orden ordenado.
- La clase mantiene una referencia `first` al primer nodo de la estructura enlazada, que corresponde al elemento mínimo del conjunto.
- La clase también mantiene una variable entera `size` para realizar un seguimiento de la cantidad de elementos en el conjunto.



Implementación de `SortedLinkedSet`

```
package dataStructures.set;

public class SortedLinkedSet<T> extends AbstractSortedSet<T> implements SortedSet<T> {
  private static final class Node<E> {  // Node inner class
    E element;
    Node<E> next;

    Node(E element, Node<E> next) {  // Node constructor
      this.element = element;
      this.next = next;
    }
  }

  private Node<T> first;
  private Comparator<T> comparator;
  private int size;

  public SortedLinkedSet(Comparator<T> comparator) {  // SortedLinkedSet constructor
    this.comparator = comparator;
    this.first = null;
    this.size = 0;
  }

  ...
}
```

- La clase también proporciona *métodos de fábrica* para construir nuevas instancias que son similares a las proporcionadas por `SortedArraySet`.

La clase de utilidad `Finder` para `SortedLinkedSet`

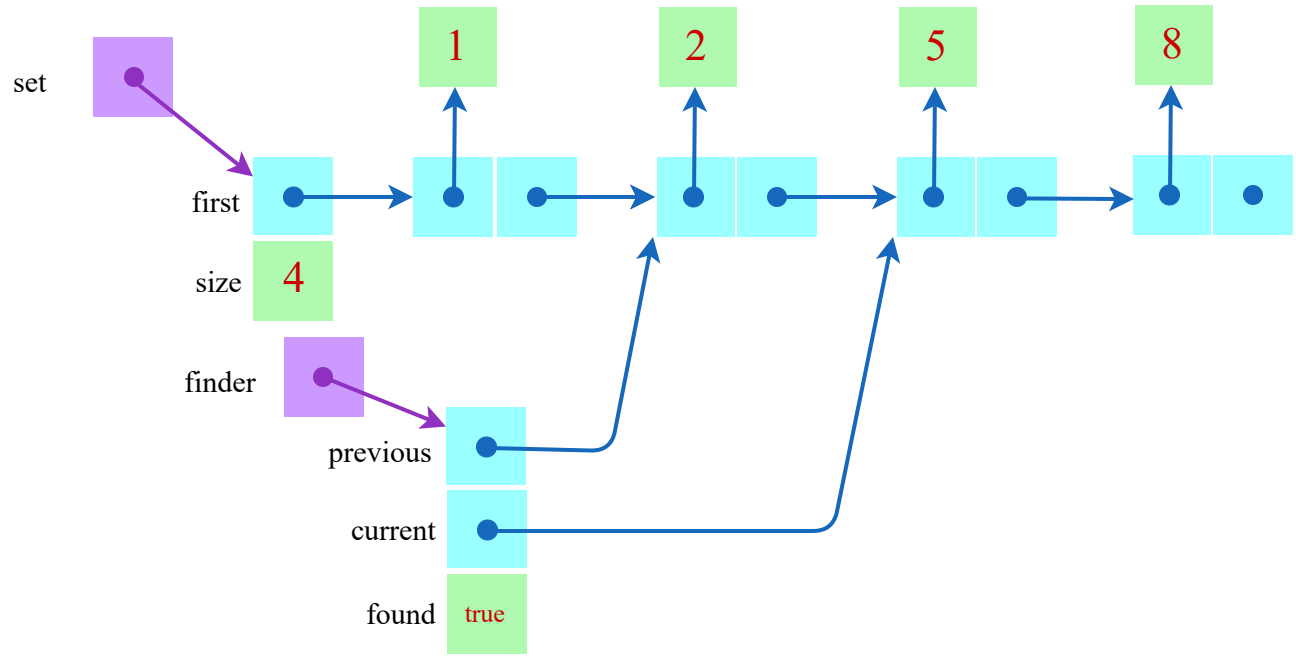
- La clase interna `Finder` ayuda a localizar una posición de un elementoen una estructura enlazada ordenada.

```
private final class Finder {
  boolean found;
  Node<T> previous, current;

  Finder(T element) {
    previous = null;
    current = first;

    int cmp = 0;
    while (current != null // the end of the linked structure is not reached
            && (cmp = comparator.compare(element, current.element)) > 0) {
      previous = current;
      current = current.next;
    }

    // element was found if we didn't reach the end of the linked structure
    // and target element is the same as the current element
    found = current != null && cmp == 0;
  }
}
```

- Al crear una instancia, el constructor de la clase `Finder` toma un `elemento` de destino y:
- Comprueba cada nodo hasta que encuentra el `elemento` o llega al nodo donde encajaría el `elemento`.
- Mientras que "actual" apunta a la coincidencia potencial, "anterior" apunta al nodo anterior. Esto resulta útil para insertar y eliminar elementos en la estructura enlazada ordenada.
- Tenga en cuenta que, si `actual` se establece en el `primer` nodo en el estructura enlazada, `anterior` se establecerá en `nulo`.
- `found` se establece en `true` cuando el elemento está presente en el Estructura enlazada.

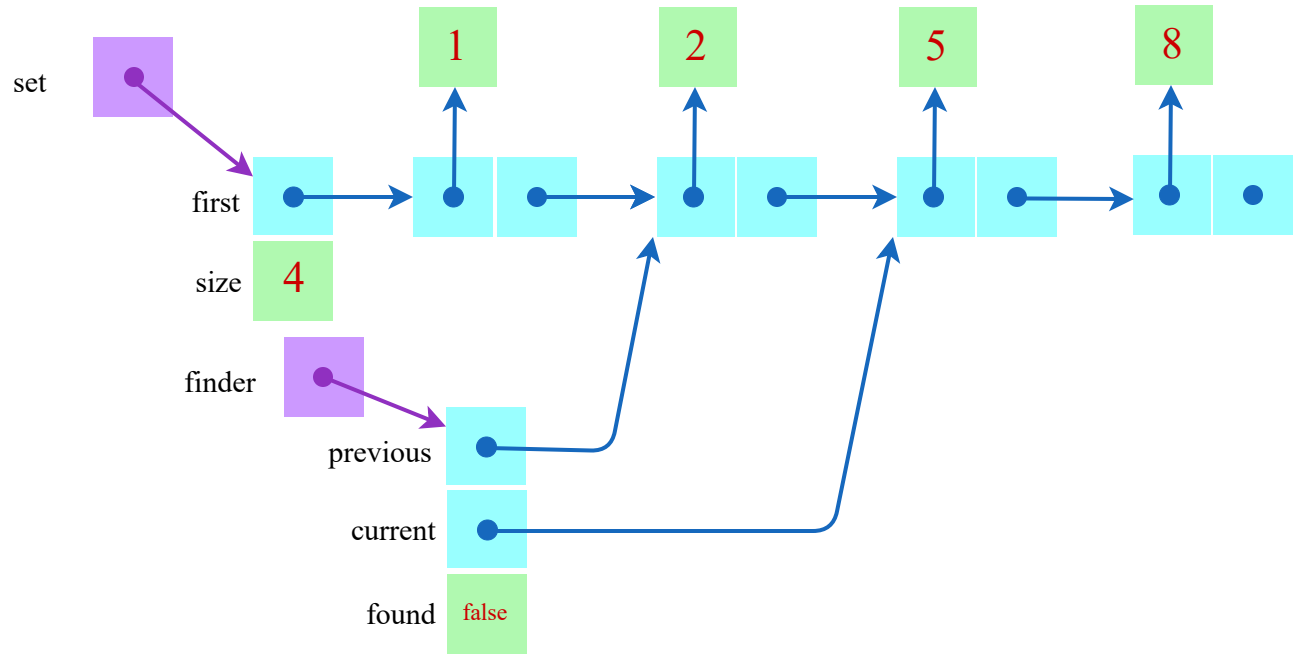
La clase de utilidad `Finder` para `SortedLinkedSet` (II)

- Resultado de una búsqueda exitosa del elemento 5:



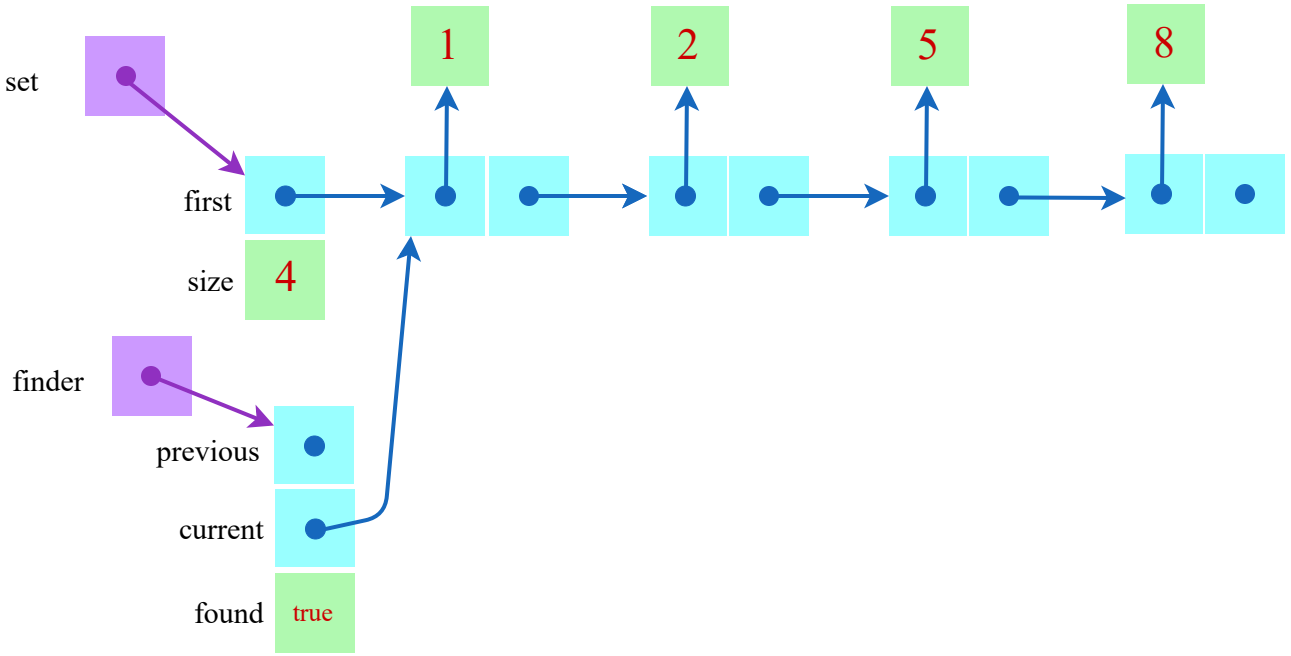
La clase de utilidad `Finder` para `SortedLinkedSet` (III)

- Resultado de una búsqueda fallida del elemento 4:



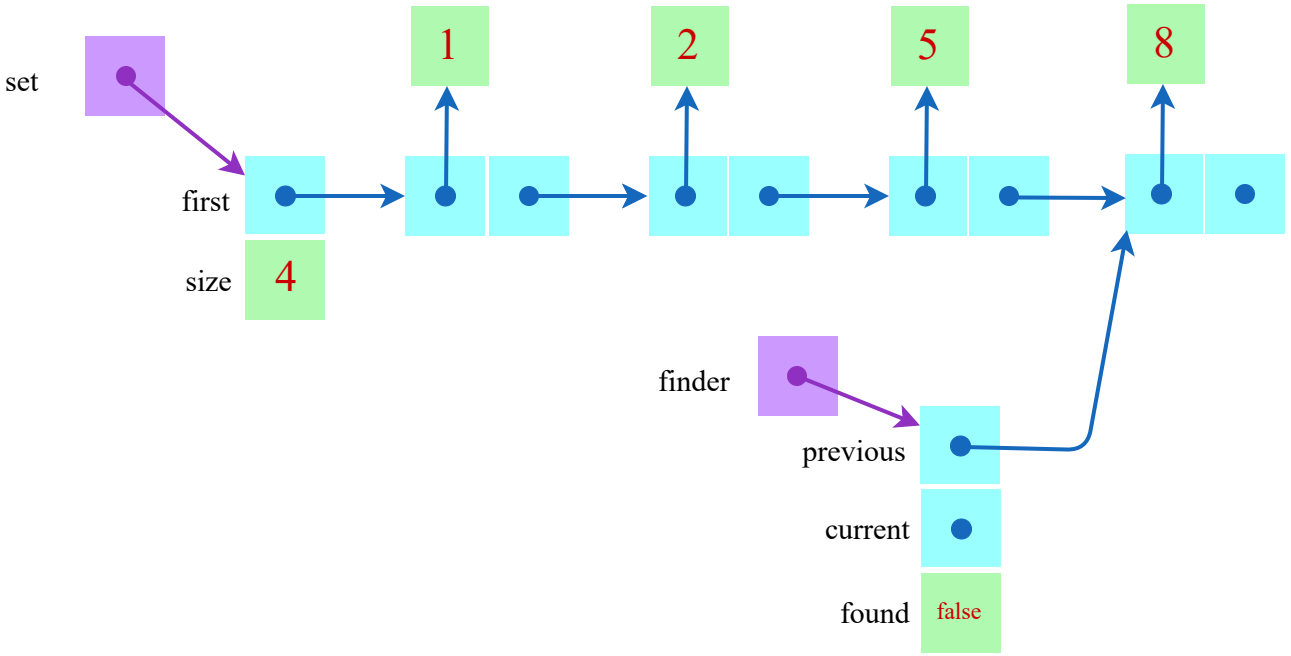
La clase de utilidad `Finder` para `SortedLinkedSet` (IV)

- Resultado de una búsqueda exitosa del elemento 1:



La clase de utilidad `Finder` para `SortedLinkedSet` (V)

- Resultado de una búsqueda fallida del elemento 9:



Búsqueda, inserción y eliminación en `SortedLinkedSet`

- En todos los casos, la clase `Finder` se utiliza para encontrar la posición del elemento de destino en la estructura enlazada ordenada.
- **Buscando**:
  - El método `contains` utiliza la clase `Finder` para determinar si el elemento de destino está en el conjunto.
- **Insertar**:
  - El método `insert` utiliza la clase `Finder` para encontrar la osición donde debe insertarse el elemento de destino.
  - Si el elemento no está en el conjunto, se crea un nuevo nodo con el objetivo. El elemento se inserta entre los nodos `anterior` y `actual`.
  - Tenga en cuenta que, si `anterior` es `nulo`, la referencia `primera` es la que debe actualizarse.
- **Eliminando**:
  - El método `delete` utiliza la clase `Finder` para encontrar la posición del elemento objetivo.
  - Si el elemento está en el conjunto, se actualiza el nodo "anterior" para señalar el nodo `current.next`, eliminando efectivamente el nodo `current` de la estructura enlazada.
  - Tenga en cuenta que, si `anterior` es `nulo`, la referencia `primera` es la que debe actualizarse.

`unión`, `intersección` y `diferencia` en `SortedArraySet`

- Las implementaciones de estos métodos pueden *aprovechar* el orden de clasificación de los conjuntos para optimizar las operaciones.
- Los conjuntos de entrada ordenados permiten construir el resultado agregando elementos secuencialmente al final, lo que garantiza que la salida permanezca ordenada.
- Sin embargo, agregar al final es costoso (O(n)) debido a que `SortedArraySet` solo tiene una referencia al `primer` nodo.
- Se puede utilizar una clase interna auxiliar (`SortedLinkedSetBuilder`) con referencias a los nodos `first` y `last` para construir
  una nueva estructura enlazada de una manera más eficiente (agregando al final de la estructura se convierte en O(1)).
- Una vez construida la nueva estructura enlazada, se puede crear un nuevo objeto `SortedLinkedSet` con el `primer` nodo del `SortedLinkedSetBuilder` auxiliar.

La clase interna `SortedLinkedSetBuilder` para `SortedArrayList`

```
private static final class SortedLinkedSetBuilder<T> {
  Node<T> first, last; // references to the first and last nodes of the linked structure being built
  int size;
  Comparator<T> comparator;

  SortedLinkedSetBuilder(Comparator<T> comparator) { // SortedLinkedSetBuilder constructor
    this.first = null;
    this.last = null;
    this.size = 0;
    this.comparator = comparator;
  }

  void append(T element) {  // appends a new element to the end of the linked structure
    assert first == null || comparator.compare(element, last.element) > 0;

    Node<T> node = new Node<>(element, null);
    if (first == null) {
      first = node;
    } else {
      last.next = node;
    }
    last = node;
    size++;
  }

  SortedLinkedSet<T> toSortedLinkedSet() { // creates a new SortedLinkedSet from the builder
    return new SortedLinkedSet<>(this);
  }
}

private SortedLinkedSet(SortedLinkedSetBuilder<T> builder) {  // copies fiirst, size and comparator
  this.first = builder.first;
  this.size = builder.size;
  this.comparator = builder.comparator;
}
```

Complejidad computacional de las operaciones de `SortedArraySet`

Operation	Cost
`empty`	O(1)
`insert`	O(n)
`delete`	O(n)
`contains`	O(n)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(1)

Comparación experimental entre SortedArraySet y SortedLinkedSet

- Medimos el tiempo de ejecución al realizar 50000 operaciones aleatorias (`insertar`, `eliminar` o `contiene`) en un conjunto inicialmente vacío.
- Usando una CPU Intel i7 860 y JDK 22:
- `SortedArraySet` fue aproximadamente 6,80 veces más rápido que `SortedLinkedList`.

El TAD Diccionario

- Un *diccionario* es una colección que almacena elementos como *pares clave-valor* (*entradas*).
- Cada clave del diccionario es *única* (sin repeticiones) y se utiliza para acceder al valor asociado.
- **Operaciones**:
  - `insert` : inserta un par clave-valor en el diccionario. Si ya existe una entrada con esa clave en el diccionario, se actualiza su valor.
  - `delete` : elimina un par clave-valor del diccionario. Si la clave no está en el diccionario, permanece inalterada.
  - `valueOf` : Devuelve el valor asociado con una clave en el diccionario. Si la clave no está en el diccionario, devuelve `null` .
  - `valueOfOrDefault` : Devuelve el valor asociado con una clave en el diccionario. Si la clave no está en el diccionario, devuelve un valor predeterminado.
  - `isDefinedAt` : comprueba si una clave está en el diccionario.
  - `isEmpty` : Comprueba si el diccionario está vacío.
  - `size` : Devuelve el número de pares clave-valor en el diccionario.
  - `clear` : elimina todos los pares clave-valor del diccionario.
  - `keys` : Devuelve un `Iterable` para recorrer las claves del diccionario.
  - `values` : Devuelve un `Iterable` para recorrer los valores de el diccionario.
  - `entries` : Devuelve un `Iterable` para recorrer los pares clave-valor del diccionario.

El diccionario en Java

- El interfaz `Dictionary<K, V>` define un diccionario con clave de tipo `K` y valor de tipo `V`.

```
package dataStructures.dictionary;

public interface Dictionary<K, V> extends Iterable<Dictionary.Entry<K, V>> {
  record Entry<K, V>(K key, V value) { // record to represent key-value pairs
    public static <K, V> Entry<K, V> of(K key, V value) { return new Entry<>(key, value); }

    public static <K, V> Entry<K, V> withKey(K key) { return new Entry<>(key, null); }

    public boolean equals(Object obj) { // equality only depends on the key
      return obj instanceof Entry<?, ?> entry && key.equals(entry.key);
    }

    public int hashCode() { return key.hashCode(); } // hash code only depends on the key

    public String toString() { return "Entry(" + key + ", " + value + ")"; }

    static <K, V> Comparator<Entry<K, V>> onKeyComparator(Comparator<K> keyComparator) {
      return (entry1, entry2) -> keyComparator.compare(entry1.key, entry2.key);
    }
  }

  void insert(K key, V value);
  void insert(Entry<K, V> entry);
  void delete(K key);
  boolean isDefinedAt(K key);
  V valueOf(K key);
  V valueOfOrDefault(K key, V defaultValue);
  boolean isEmpty();
  int size();
  void clear();
  Iterable<K> keys();
  Iterable<V> values();
  Iterable<Entry<K, V>> entries();
}
```

El TAD diccionario ordenado en Java

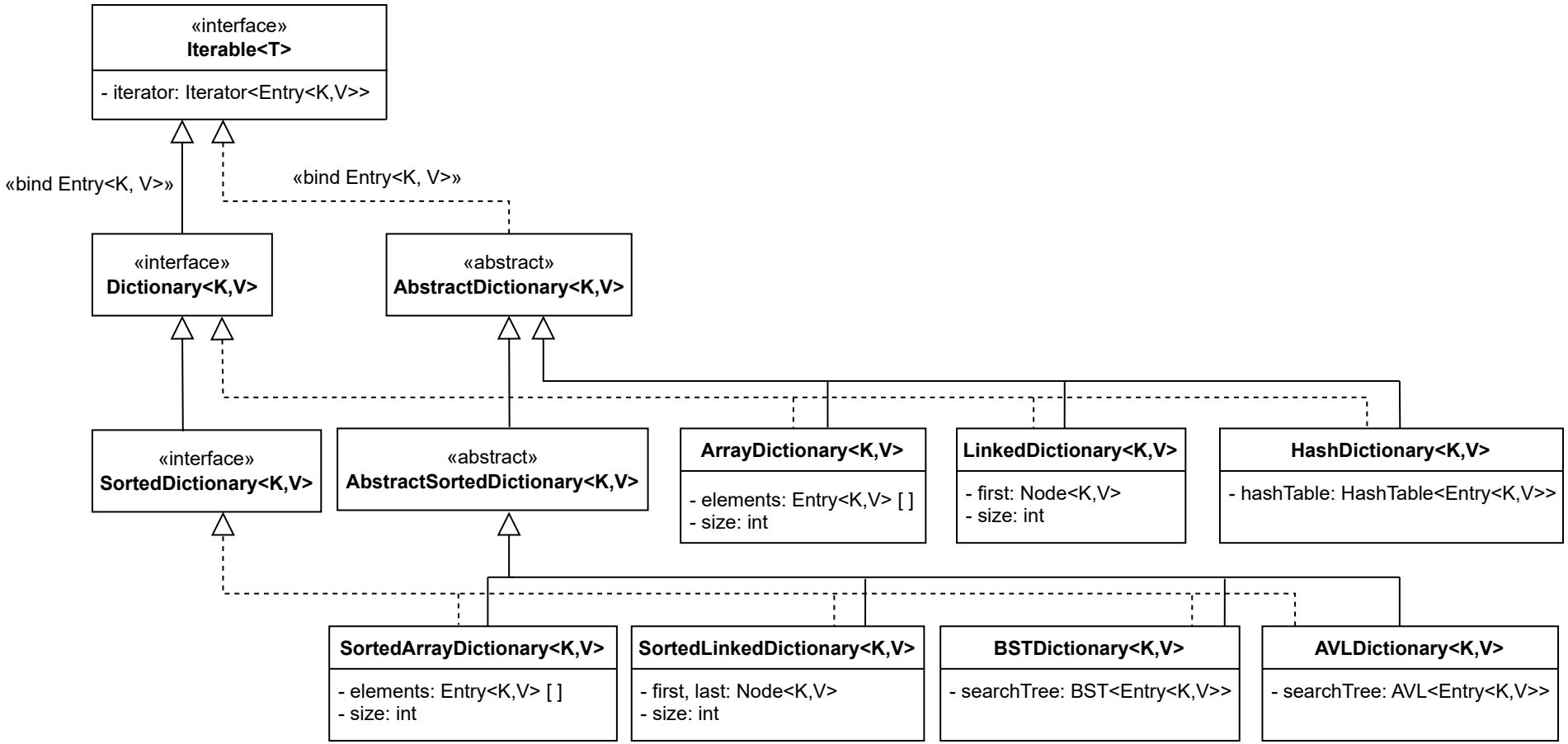
- Un *diccionario ordenado* es un diccionario donde las claves se almacenan en *orden ordenado*.
- Cuando se itera un diccionario ordenado, los elementos se devuelven en orden ordenado.
- **Operaciones**:
  - `comparator`: Devuelve el comparador que define el *igualdad* y *ordenación* de claves dentro del diccionario.
  - `minimum`: devuelve el par clave-valor con la clave más pequeña, según lo determinado por el comparador, del diccionario. Lanza una `NoSuchElementException` si el diccionario está vacío.
  - `maximum`: devuelve el par clave-valor con la clave más grande, según lo determinado por el comparador, del diccionario. Lanza una `NoSuchElementException` si el diccionario está vacío.

```
package dataStructures.dictionary;

public interface SortedDictionary<K, V> extends Dictionary<K, V> {
  Comparator<K> comparator();
  Entry<K, V> minimum();
  Entry<K, V> maximum();
}
```

Implementaciones de TAD de diccionario y SortedDictionary

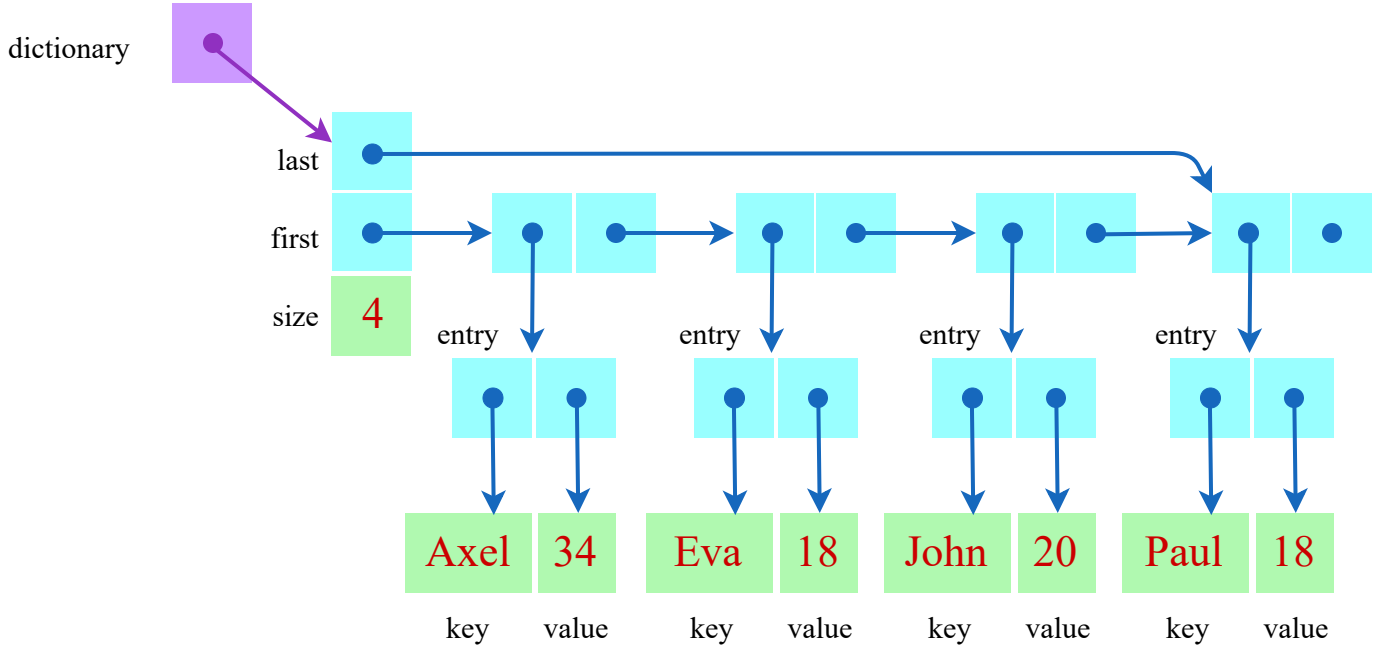
- Un diccionario se puede implementar utilizando diferentes *estructuras de datos*.
- Diferentes *clases* pueden implementar la interfaz `Dictionary<K, V>`:
  - `ArrayDictionary<K, V>`: utiliza un Array sin ordenar para almacenar pares clave-valor.
  - `LinkedDictionary<K, V>`: utiliza una estructura enlazada sin ordenar para almacenar pares clave-valor.
  - `HashDictionary<K, V>`: utiliza una *tabla hash* para almacenar pares clave-valor. La mayoría de las operaciones son O(1).
- Se puede implementar un diccionario ordenado utilizando diferentes *estructuras de datos*.
- Diferentes *clases* pueden implementar la interfaz `SortedDictionary<K, V>`:
  - `SortedArrayDictionary<K, V>`: utiliza un Array ordenada para almacenar pares clave- valor en orden ordenado (según las claves). Puede utilizar la *búsqueda binaria* para acelerar las operaciones.
  - `SortedLinkedDictionary<K, V>`: utiliza un diccionario vinculado ordenado Estructura para almacenar pares clave-valor en orden ordenado.
  - `BSTDictionary<K, V>`: utiliza un *árbol de búsqueda binaria* para Almacenar pares clave-valor en orden ordenado.
  - `AVLDictionary<K, V>`: utiliza un *árbol AVL* para almacenar claves. pares de valores en orden ordenado. La mayoría de las operaciones son O(log n).
- Las clases abstractas base `AbstractDictionary<K, V>` y `AbstractSortedDictionary<K, V>` proporcionan implementación para los métodos `equals`, `hashCode` y `toString`.

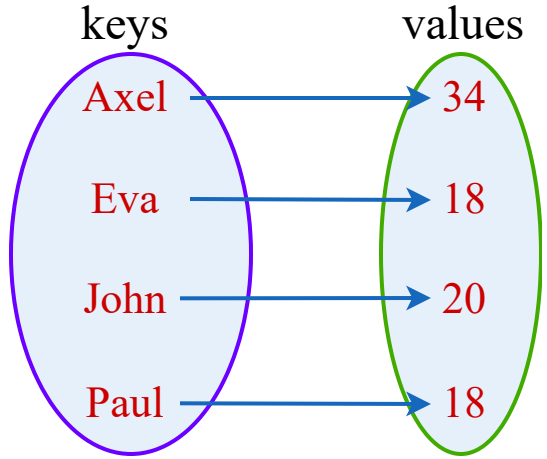
Implementaciones de ADT de diccionarios y SortedDictionary (II)



La clase `SortedLinkedDictionary`

- La clase `SortedLinkedDictionary<K, V>` implementa la interfaz `SortedDictionary<K, V>`.
- De acuerdo con el principio fundamental de un diccionario, cada clave es *distinta*, lo que garantiza que no haya duplicación de claves dentro de la estructura.
- Utiliza un `Comparator<K>`, proporcionado en la construcción, para determinar la igualdad y el orden de las claves.
- Internamente, mantiene una estructura enlazada de nodos, cada uno sosteniendo una `Entrada<K, V>` que representa un par clave-valor.
- La estructura enlazada se mantiene en *orden de clave* ascendente (según el comparador).
- Una referencia `primera` apunta al nodo con la clave más pequeña, mientras que una referencia `última` apunta al nodo con la clave más grande.
- El atributo `size` rastrea el recuento actual de clave-valor pares contenidos en el diccionario.



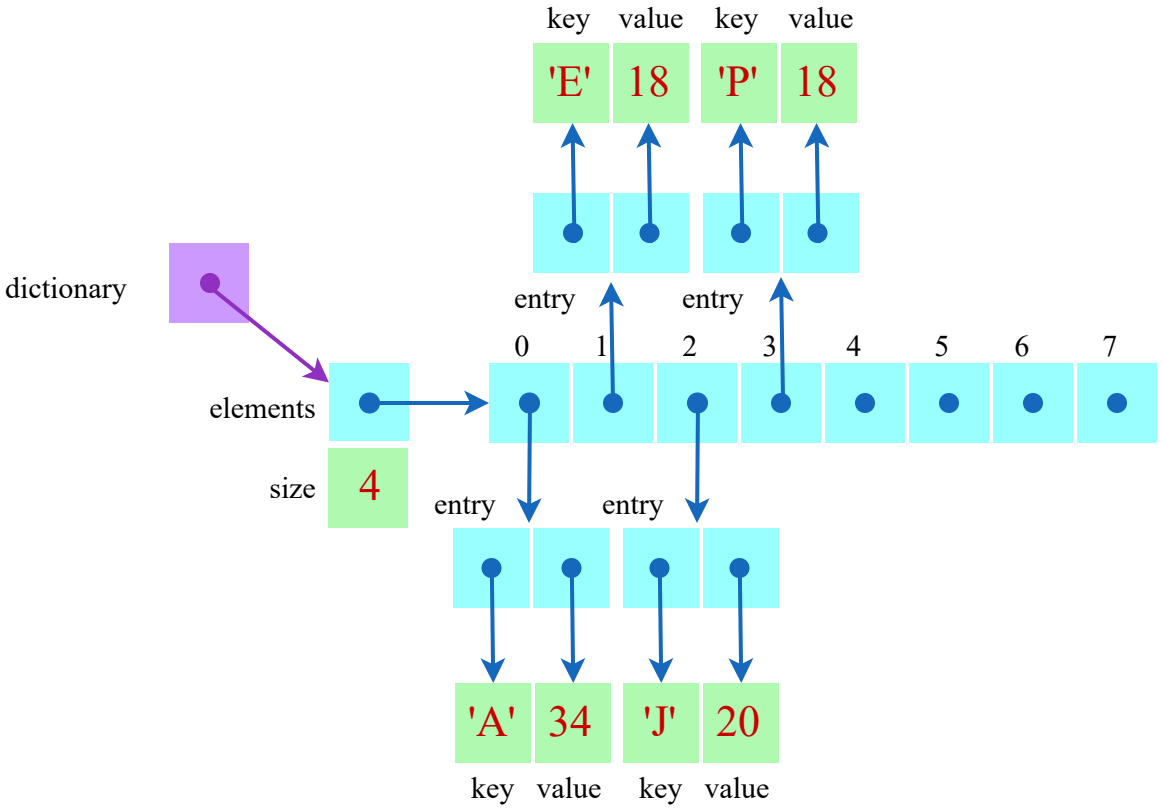


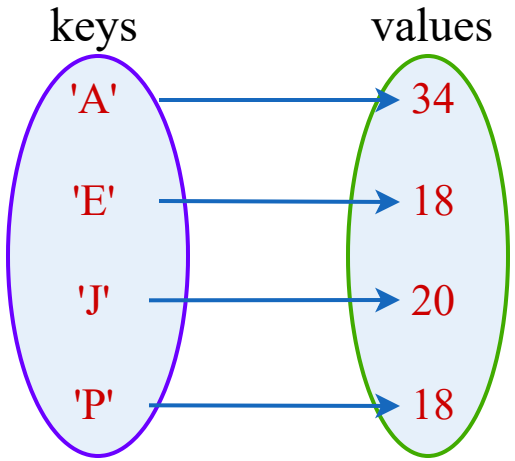
Complejidad computacional de `SortedLinkedDictionary`

Operation	Cost
`empty`	O(1)
`insert`	O(n)
`delete`	O(n)
`isDefinedAt`	O(n)
`valueOf`	O(n)
`valueOfOrDefault`	O(n)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(1)

La clase `SortedArrayDictionary`

- La clase `SortedArrayDictionary<K, V>` implementa la interfaz `SortedDictionary<K, V>`.
- Utiliza un `Comparator<K>`, proporcionado en la construcción, para determinar la igualdad y el orden de las claves.
- Internamente, mantiene un Array de celdas, cada una de las cuales contiene una `Entrada<K, V>` que representa un par clave-valor.
- Inicialmente, la Array tiene un tamaño fijo (la *capacidad* del diccionario), pero puede crecer dinámicamente cuando sea necesario.
- La Array se mantiene en orden de clave ascendente (según al comparador).
- El atributo `size` rastrea el recuento actual de clave-valor pares contenidos en el diccionario.





Complejidad computacional de `SortedArrayDictionary`

Operation	Cost
`empty`	O(1)
`insert`	O(n)
`delete`	O(n)
`isDefinedAt`	O(log n)
`valueOf`	O(log n)
`valueOfOrDefault`	O(log n)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(n)

El TAD cola de prioridad

- Una **cola de prioridad** es una colección que almacena elementos con una *prioridad* asociada.
- Los elementos se ponen en cola en orden arbitrario, pero se quitan de la cola en orden de prioridad.
- **Operaciones:**
  - `enqueue` : Agrega un elemento con una prioridad a la prioridad cola.
  - `dequeue` : elimina el elemento con mayor prioridad.
  - `first` : Devuelve el elemento con mayor prioridad sin quitarlo.
  - `isEmpty` : verifica si la cola de prioridad no contiene elementos.
  - `size` : Devuelve el número de elementos en la prioridad cola.
  - `clear` : purga todos los elementos, limpiando completamente la cola de prioridad.

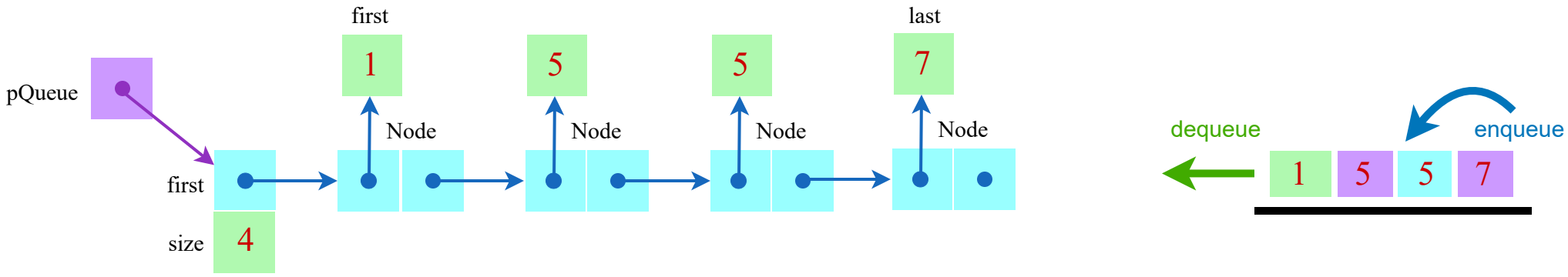
El ADT PriorityQueue en Java

- La interfaz `PriorityQueue<T>` define una cola de prioridad con elementos de tipo `T`.
- Utiliza un `Comparator<T>` para asignar prioridades a los elementos, siguiendo el principio de que se considera que un elemento tiene una *prioridad más alta* si es *más pequeño* según los criterios del comparador. Esto también se conoce como *cola de prioridad mínima*.

```
package dataStructures.priorityQueue;

public interface PriorityQueue<T> {
  Comparator<T> comparator();
  void enqueue(T element);
  void dequeue();
  T first();
  boolean isEmpty();
  int size();
  void clear();
}
```

La clase `SortedLinkedPriorityQueue`

- La clase `SortedLinkedPriorityQueue<T>` implementa la interfaz `PriorityQueue<T>`.
- Utiliza un `Comparator<T>`, proporcionado en la construcción, para determinar la prioridad de los elementos.
- Internamente, mantiene una estructura enlazada de nodos, cada uno que contiene un elemento y una referencia al siguiente nodo.
- La estructura enlazada se mantiene en *elemento ascendente orden* (según el comparador).
- Una referencia `primera` apunta al nodo con el *más pequeño elemento*, que es el elemento con la *prioridad más alta*.
- El atributo `size` rastrea el recuento total de todos los elementos en la cola de prioridad.

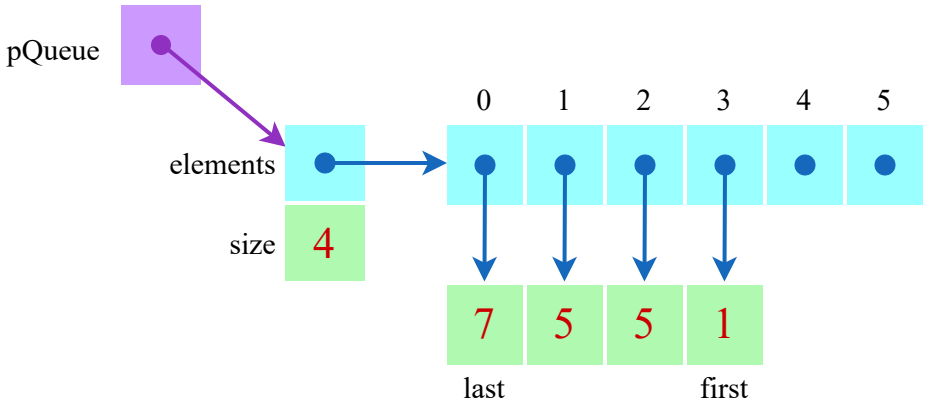


Complejidad computacional de `SortedLinkedPriorityQueue`

Operation	Cost
`empty`	O(1)
`enqueue`	O(n)
`dequeue`	O(1)
`first`	O(1)
`isEmpty`	O(1)
`size`	O(1)

La clase `SortedArrayPriorityQueue<T>`

- Implementa la interfaz `PriorityQueue<T>`, organizando elementos por prioridad.
- Un `Comparator<T>`, proporcionado en la construcción, determina la prioridad de los elementos, donde los *valores más pequeños* indican una *prioridad más alta*.
- Internamente, mantiene un Array de celdas, cada una de las cuales contiene un elemento.
- Comienza con una capacidad fija, pero es capaz de expandirse dinámicamente según sea necesario.
- La Array se *ordena en orden descendente* según el comparador, lo que facilita una gestión eficiente de las prioridades.
- El atributo `size` denota el número total de elementos en la cola.
- El elemento con mayor prioridad se ubica al final de la Array, en el índice `size - 1`, lo que permite una eliminación eficiente simplemente disminuyendo el `size`.

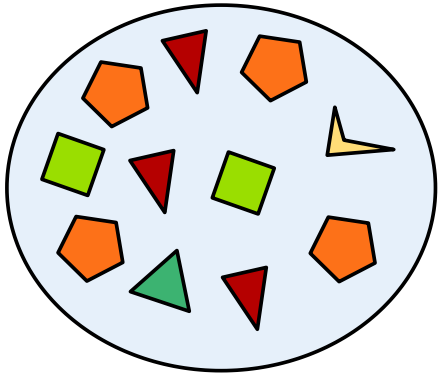


Complejidad computacional de `SortedArrayPriorityQueue` Operaciones

Operation	Cost
`empty`	O(1)
`enqueue`	O(n)
`dequeue`	O(1)
`first`	O(1)
`isEmpty`	O(1)
`size`	O(1)

El TAD bolsa (también conocido como Multiset)

- Una **bolsa** (también conocida como **multiconjunto**) es una colección que permite el almacenamiento de elementos, incluidas múltiples instancias del mismo elemento, denominadas *ocurrencias*.
- **Operaciones:**
  - `insert` : Agrega una nueva instancia de un elemento a la bolsa.
  - `delete` : Elimina una sola instancia de un elemento de la bolsa.
  - `contains` : Verifica si un elemento está presente en la bolsa.
  - `occurrences` : Determina el recuento de instancias de un elemento. Elemento específico dentro de la bolsa (devuelve 0 si está ausente).
  - `isEmpty` : verifica si la bolsa no contiene elementos.
  - `size` : calcula el recuento total de todos los elementos en la bolsa. Bolsa, incluidos duplicados.
  - `clear` : Purga todos los elementos, limpiando la bolsa por completo.
  - `iterator` : Proporciona un `Iterador` para navegar a través de los elementos de la bolsa, iterando varias veces sobre elementos con varias instancias.



Bag of Polygons

El TAD Bag y SortedBag en Java

- La interfaz `Bag<T>` define una bolsa con elementos de tipo `T`.

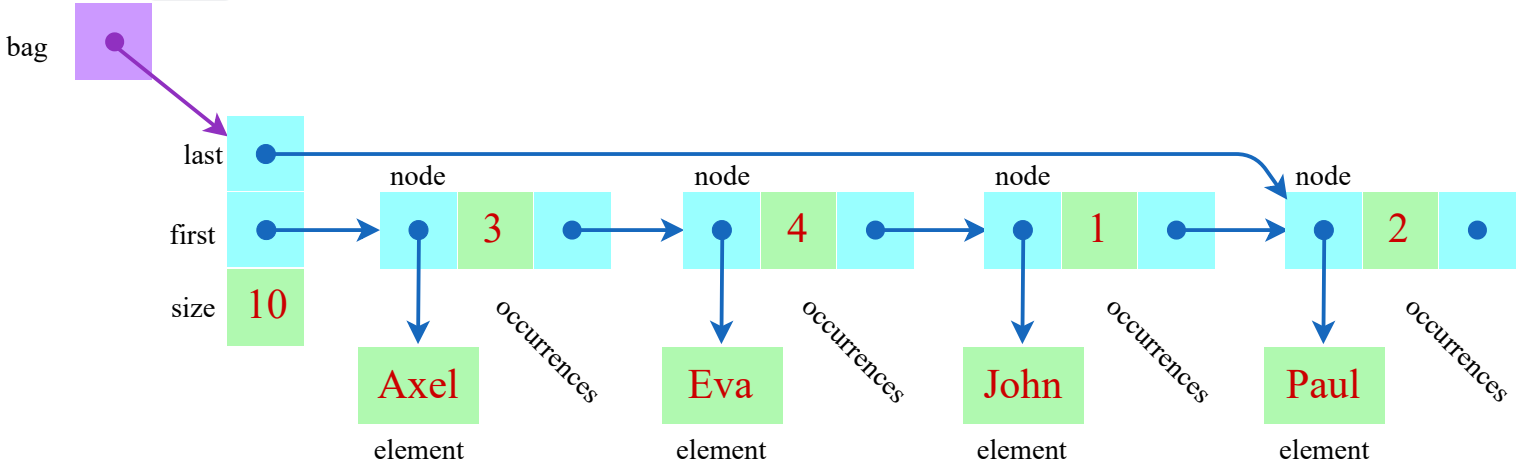
```
package dataStructures.bag;

public interface Bag<T> extends Iterable<T> {
  void insert(T element);
  void delete(T element);
  boolean contains(T element);
  int occurrences(T element);
  boolean isEmpty();
  int size();
  void clear();
}
```

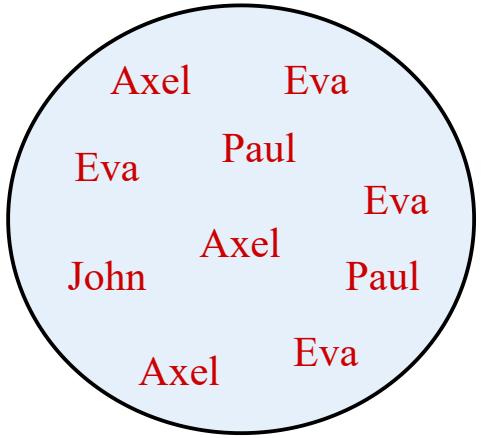
- `SortedBag<T>` define una bolsa ordenada con elementos de tipo `T`.

```
package dataStructures.bag;

public interface SortedBag<T> extends Bag<T> {
  Comparator<T> comparator();
  T minimum();
  T maximum();
}
```

La clase `SortedLinkedBag`

- La clase `SortedLinkedBag<T>` implementa la clase `SortedBag<T>` interfaz.
- Utiliza un `Comparator<T>`, proporcionado en la construcción, para definir la igualdad y el orden de los elementos.
- Internamente, mantiene una estructura enlazada de nodos, cada uno de los cuales contiene un elemento, un recuento de ocurrencias y una referencia al siguiente nodo.
- Cada nodo dentro de la estructura enlazada contiene un *distinto* elemento y un número *positivo* de ocurrencias. Cualquier nodo con cero ocurrencias se elimina de la estructura.
- La estructura enlazada se mantiene en orden ascendente *elemento orden* (según el comparador).
- Una referencia `primera` apunta al nodo con el elemento más pequeño, mientras que una referencia `última` apunta al nodo con el elemento más grande.
- El atributo `size` rastrea el recuento total de todos los elementos en la bolsa, incluidos los duplicados.



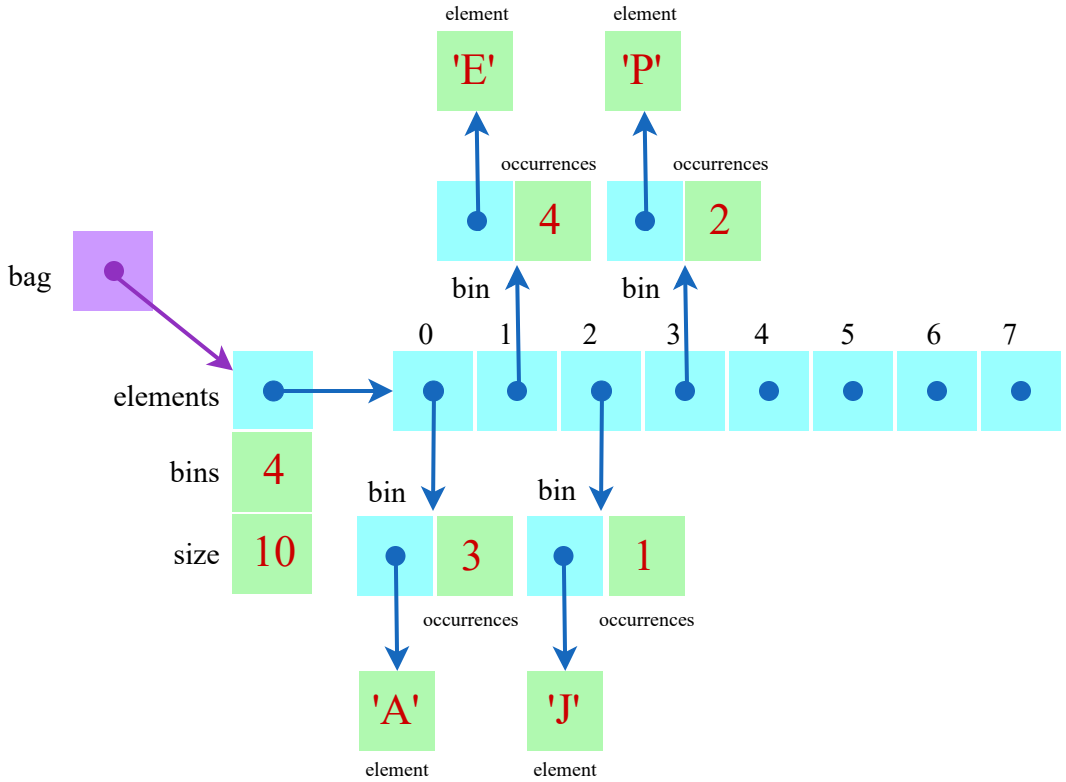


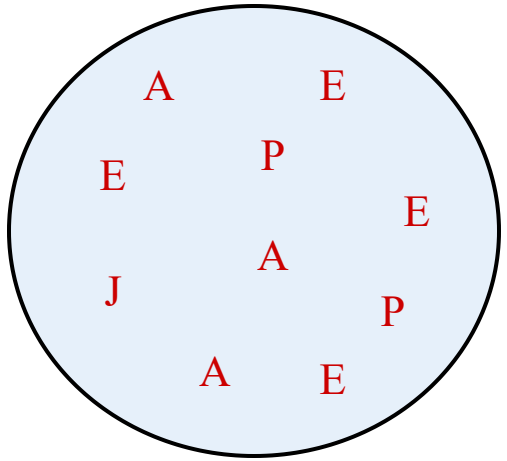
Complejidad computacional de las operaciones de `SortedLinkedBag`

Operation	Cost
`empty`	O(1)
`insert`	O(n)
`delete`	O(n)
`contains`	O(n)
`occurrences`	O(n)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(1)

La clase `SortedArrayBag`

- La clase `SortedArrayBag<T>` implementa `SortedBag<T>` interfaz.
- Utiliza un `Comparator<T>`, proporcionado en la construcción, para definir la igualdad y el orden de los elementos.
- Internamente, la estructura está compuesta por un Array de celdas, y cada celda contiene un `Bin<T>`, que significa un emparejamiento de un elemento y la cantidad de veces que ocurre.
- Cada contenedor de la estructura contiene un elemento *distinto* y un número de ocurrencias *positivo*. Cualquier contenedor con cero ocurrencias se elimina de la estructura.
- Inicialmente, la Array tiene un tamaño fijo (la *capacidad* de la bolsa), pero puede crecer dinámicamente cuando sea necesario.
- La Array se mantiene en *orden de elementos* ascendente (según el comparador).
- El atributo `bins` cuenta la cantidad de celdas utilizadas actualmente en la bolsa.
- El atributo de tamaño suma el número total de elementos en el bolsa, contando cada ocurrencia.





Complejidad computacional de las operaciones de `SortedArrayBag`

Operation	Cost
`empty`	O(1)
`insert`	O(n)
`delete`	O(n)
`contains`	O(log n)
`occurrences`	O(log n)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(n)