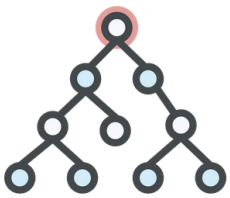
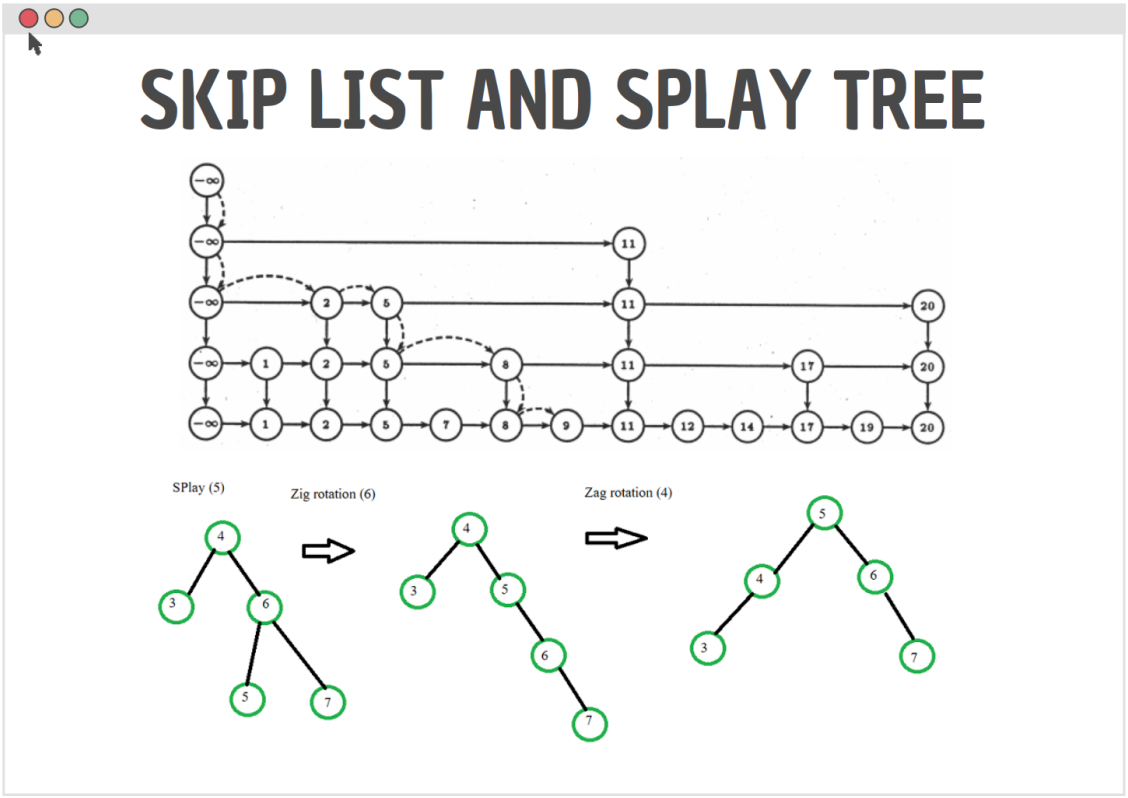


ESTRUCTURAS DE DATOS Y ALGORITMOS con Java



Profesor(a):
Edson Luque Mamani

Estudiantes:
Jorge Luis Mamani Huarsaya
Victor Narciso Mamani Anahua

27 de junio, 2024

Skip List

La clase SkipList

La clase `SkipList<T>` en Java es una implementación de una estructura de datos avanzada conocida como SkipList, diseñada para proporcionar operaciones eficientes de búsqueda, inserción y eliminación. Utiliza múltiples niveles de punteros, permitiendo un acceso más rápido en comparación con las listas enlazadas simples. Esta implementación genérica admite cualquier tipo que implemente la interfaz `Comparable`, asegurando que los elementos puedan ser ordenados y comparados correctamente. A continuación, se describen los componentes y métodos principales de esta implementación.

Definición de la Clase SkipList

La clase `SkipList<T>` implementa una SkipList genérica que admite elementos que implementan la interfaz `Comparable`. La clase contiene una constante `MAX_LEVEL` que define el número máximo de niveles en la lista, un nodo cabeza `head` de tipo `SkipListNode<T>` con el nivel máximo, y un objeto `Random` para generar niveles aleatorios. Además, se mantiene un entero `level` que indica el nivel más alto actualmente en uso. La estructura de datos permite que cada nodo apunte a varios otros nodos a diferentes niveles, proporcionando así una manera rápida de atravesar la lista.

```java
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Random;

public class SkipList<T extends Comparable<? super T>> implements Iterable<T> {
  private static final int MAX_LEVEL = 4;
  private final SkipListNode<T> head = new SkipListNode<>(null, MAX_LEVEL);
  private final Random random = new Random();
  private int level = 0;
```

Definición de la Clase SkipListNode

La clase `SkipListNode<T>` es una clase interna estática que representa un nodo en la SkipList. Cada nodo contiene un valor de tipo `T` y un array de punteros `forward` que apunta a otros nodos en niveles distintos. El constructor de `SkipListNode` inicializa estos punteros según el nivel proporcionado, asegurando que cada nodo pueda tener múltiples referencias hacia adelante, lo que permite saltar varios nodos en una sola operación, mejorando así la eficiencia de las búsquedas.

```java
  private static class SkipListNode<T> {
    final T value;
    final SkipListNode<T>[] forward;

    @SuppressWarnings("unchecked")
    SkipListNode(T value, int level) {
      this.value = value;
      this.forward = new SkipListNode[level + 1];
    }
  }
```

Método contains

El método `contains(T value)` verifica si un valor está presente en la lista. Recorre la lista desde el nivel superior hacia abajo, avanzando en cada nivel hasta encontrar el nodo que contiene el valor o

determinar que no está presente. Esto permite realizar búsquedas de manera rápida, aprovechando los niveles superiores para saltar grandes segmentos de la lista y reducir el número total de comparaciones necesarias.

```
public boolean contains(T value) {
  SkipListNode<T> current = head;
  for (int i = level; i >= 0; i--) {
    while (current.forward[i] != null && current.forward[i].value.compareTo(value)
    ↪  < 0)
      current = current.forward[i];
  }

  current = current.forward[0];
  return current != null && current.value.compareTo(value) == 0;
}
```

Método add

El método `add(T value)` agrega un nuevo valor a la SkipList. Primero, encuentra las posiciones donde se deben actualizar los punteros para mantener la integridad de la lista. Luego, si el valor no está presente, genera un nuevo nivel para el nodo, actualiza los punteros y, si es necesario, incrementa el nivel de la lista. Esta técnica asegura que la lista se mantenga balanceada y que las operaciones de inserción sean eficientes.

```
public void add(T value) {
  SkipListNode<T>[] update = new SkipListNode[MAX_LEVEL + 1];
  SkipListNode<T> current = head;

  for (int i = level; i >= 0; i--) {
    while (current.forward[i] != null && current.forward[i].value.compareTo(value)
    ↪  < 0)
      current = current.forward[i];
    update[i] = current;
  }

  current = current.forward[0];

  if (current == null || !current.value.equals(value)) {
    int newLevel = randomLevel();
    if (newLevel > level) {
      for (int i = level + 1; i <= newLevel; i++)
        update[i] = head;
      level = newLevel;
    }

    SkipListNode<T> newNode = new SkipListNode<>(value, newLevel);
    for (int i = 0; i <= newLevel; i++) {
      newNode.forward[i] = update[i].forward[i];
      update[i].forward[i] = newNode;
    }
  }
}
```

Método remove

El método `remove(T value)` elimina un valor de la SkipList. Similar a `add`, primero encuentra las posiciones de los punteros que deben actualizarse. Si encuentra el nodo con el valor, actualiza los punteros para omitir este nodo y ajusta el nivel de la lista si es necesario. Esto permite eliminar elementos de manera eficiente, ajustando dinámicamente los niveles de la lista para mantener el equilibrio y el rendimiento.

```
public boolean remove(T value) {
  SkipListNode<T>[] update = new SkipListNode[MAX_LEVEL + 1];
  SkipListNode<T> current = head;
  for (int i = level; i >= 0; i--) {
    while (current.forward[i] != null && current.forward[i].value.compareTo(value)
    ↪  < 0)
      current = current.forward[i];
    update[i] = current;
  }
  current = current.forward[0];
  if (current != null && current.value.equals(value)) {
    for (int i = 0; i <= level; i++) {
      if (update[i].forward[i] != current)
        break;
      update[i].forward[i] = current.forward[i];
    }
    while (level > 0 && head.forward[level] == null)
      level--;
    return true;
  }
  return false;
}
```

Método randomLevel

El método `randomLevel()` genera un nivel aleatorio para un nodo nuevo. Incrementa el nivel hasta `MAX_LEVEL` con una probabilidad del 50% en cada incremento. Esta función es crucial para mantener la naturaleza probabilística de la SkipList, asegurando que la distribución de niveles se mantenga equilibrada a largo plazo y proporcionando la base para la eficiencia de las operaciones de búsqueda e inserción.

```
private int randomLevel() {
  int lvl = 0;
  while (lvl < MAX_LEVEL && random.nextInt(2) == 0)
    lvl++;
  return lvl;
}
```

Método iterator

La clase implementa la interfaz `Iterable<T>` y proporciona un iterador que recorre la SkipList de principio a fin. Esto permite utilizar la SkipList en bucles `for-each` y otras construcciones que requieren un iterador, proporcionando una manera sencilla y natural de recorrer los elementos de la lista.

```
@Override
public Iterator<T> iterator() {
```

```
    return new Iterator<T>() {
      private SkipListNode<T> current = head.forward[0];

      @Override
      public boolean hasNext() {
        return current != null;
      }

      @Override
      public T next() {
        if (current == null)
          throw new NoSuchElementException();
        T value = current.value;
        current = current.forward[0];
        return value;
      }
    };
  }
```

Método printList

El método `printList()` imprime la estructura de la SkipList mostrando los nodos en cada nivel, útil para visualizar el contenido y la estructura de la lista. Este método es particularmente útil para depuración y para entender cómo se distribuyen los nodos en diferentes niveles, proporcionando una representación visual clara del estado de la SkipList.

```
  public void printList() {
    System.out.println("SkipList:");

    for (int i = MAX_LEVEL; i >= 0; i--) {
      SkipListNode current = head.forward[i];
      System.out.print("Level " + i + ": ");

      while (current != null) {
        System.out.print(current.value + " ");
        current = current.forward[i];
      }
      System.out.println();
    }
  }
}
```

Splay Tree

Estructura de la Implementación

La implementación se divide en varias partes: la definición de los nodos del árbol, las operaciones básicas del Splay Tree (inserción, eliminación, splay, etc.) y la representación gráfica utilizando GraphStream.

Definición de la Clase Nodo

La clase `TreeNode` es una clase estática anidada dentro de la clase `SplayTree`. Esta clase representa los nodos del árbol.

```
private static class TreeNode<T> {
  T key;
  TreeNode<T> left, right;

  TreeNode(T key) {
    this.key = key;
    this.left = this.right = null;
  }
}
```

Operaciones Básicas del Splay Tree

La clase `SplayTree` implementa las operaciones básicas de un Splay Tree como inserción, eliminación, búsqueda y la operación de splay. A continuación, se describen algunos de estos métodos:

Método Insert

El método `insert` añade un nuevo nodo al árbol y luego aplica la operación de splay para mover dicho nodo a la raíz.

```
@Override
public void insert(T key) {
  root = insert(root, key);
  splay(key);
}

private TreeNode<T> insert(TreeNode<T> node, T key) {
  if (node == null) {
    return new TreeNode<>(key);
  }
  int cmp = key.compareTo(node.key);
  if (cmp < 0) {
    node.left = insert(node.left, key);
  } else if (cmp > 0) {
    node.right = insert(node.right, key);
  }
  return node;
}
```

Método Splay

El método `splay` mueve el nodo con la clave especificada a la raíz del árbol. Esta operación mejora el rendimiento de futuras operaciones en el nodo accedido.

```
@Override
public void splay(T key) {
  root = splay(root, key);
}

private TreeNode<T> splay(TreeNode<T> node, T key) {
  if (node == null) return null;

  int cmp1 = key.compareTo(node.key);
  if (cmp1 < 0) {
    if (node.left == null) return node;
    int cmp2 = key.compareTo(node.left.key);
    if (cmp2 < 0) {
      node.left.left = splay(node.left.left, key);
      node = rotateRight(node);
    } else if (cmp2 > 0) {
      node.left.right = splay(node.left.right, key);
      if (node.left.right != null) {
        node.left = rotateLeft(node.left);
      }
    }
    return node.left == null ? node : rotateRight(node);
  } else if (cmp1 > 0) {
    if (node.right == null) return node;
    int cmp2 = key.compareTo(node.right.key);
    if (cmp2 < 0) {
      node.right.left = splay(node.right.left, key);
      if (node.right.left != null) {
        node.right = rotateRight(node.right);
      }
    } else if (cmp2 > 0) {
      node.right.right = splay(node.right.right, key);
      node = rotateLeft(node);
    }
    return node.right == null ? node : rotateLeft(node);
  } else {
    return node;
  }
}
```

Rotaciones

Las rotaciones son operaciones auxiliares utilizadas durante la operación de splay para reestructurar el árbol. Las siguientes son las implementaciones de rotación a la derecha e izquierda:

```
private TreeNode<T> rotateRight(TreeNode<T> node) {
  TreeNode<T> temp = node.left;
  node.left = temp.right;
  temp.right = node;
```

```
  return temp;
}

private TreeNode<T> rotateLeft(TreeNode<T> node) {
  TreeNode<T> temp = node.right;
  node.right = temp.left;
  temp.left = node;
  return temp;
}
```

Representación Gráfica con GraphStream

La clase `SplayTree` también incluye un método para representar gráficamente el árbol utilizando la librería GraphStream. Este método agrega los nodos y las aristas al grafo y lo muestra visualmente.

Método DisplayTree

El método `displayTree` inicializa el grafo y llama al método `addNodesAndEdges` para agregar los nodos y las aristas.

```
public void displayTree() {
  System.setProperty("org.graphstream.ui", "swing");
  Graph graph = new SingleGraph("Splay Tree");

  addNodesAndEdges(graph, root);

  for (Node node : graph) {
    node.setAttribute("ui.label", node.getId());
  }

  graph.display();
}
```

Método AddNodesAndEdges

El método `addNodesAndEdges` es responsable de agregar los nodos y las aristas al grafo. Para evitar la duplicación de código, se utilizan los métodos auxiliares `addNode` y `addEdge`:

```
private void addNodesAndEdges(Graph graph, TreeNode<T> node) {
  if (node == null) return;

  addNode(graph, node.key.toString());

  if (node.left != null) {
    addNode(graph, node.left.key.toString());
    addEdge(graph, node.key.toString(), node.left.key.toString());
    addNodesAndEdges(graph, node.left);
  }

  if (node.right != null) {
    addNode(graph, node.right.key.toString());
    addEdge(graph, node.key.toString(), node.right.key.toString());
```

```
    addNodesAndEdges(graph, node.right);
  }
}

private void addNode(Graph graph, String key) {
  if (graph.getNode(key) == null) {
    graph.addNode(key);
  }
}

private void addEdge(Graph graph, String parentKey, String childKey) {
  String edgeId = parentKey + "-" + childKey;
  if (graph.getEdge(edgeId) == null) {
    graph.addEdge(edgeId, parentKey, childKey, true);
  }
}
```

Visualización

Esta representación gráfica es útil para entender el comportamiento del Splay Tree y cómo las operaciones de splay mejoran el rendimiento del árbol.

```
$ javac -cp .:jar/gs-algo-2.0.jar:jar/gs-core-2.0.jar:jar/gs-ui-swing-2.0.jar
↪  TestSplayTree.java
$ java -cp .:jar/gs-algo-2.0.jar:jar/gs-core-2.0.jar:jar/gs-ui-swing-2.0.jar
↪  TestSplayTree
10 20 25 30 40 50
```

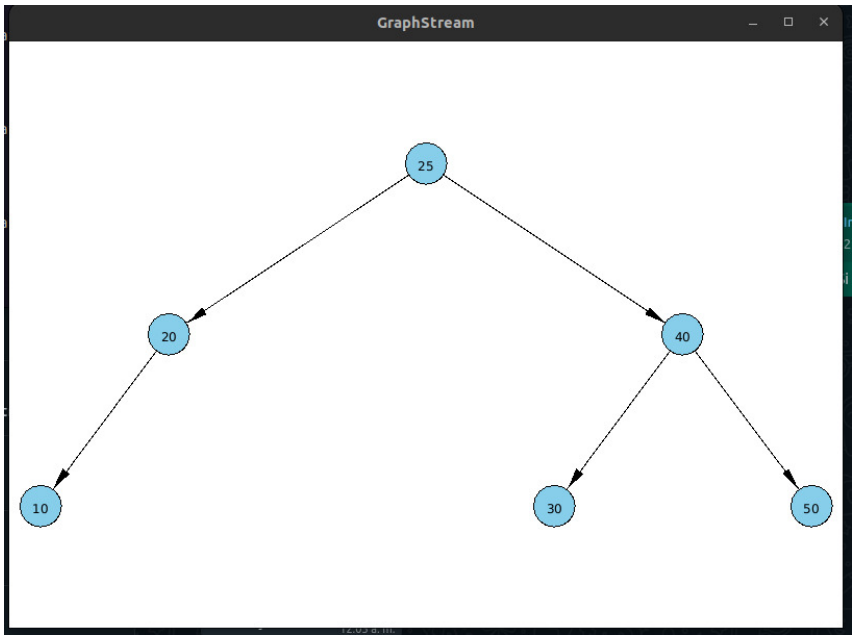


Figure 1: Splay Tree