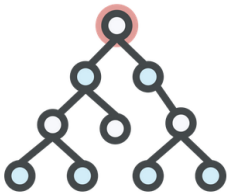
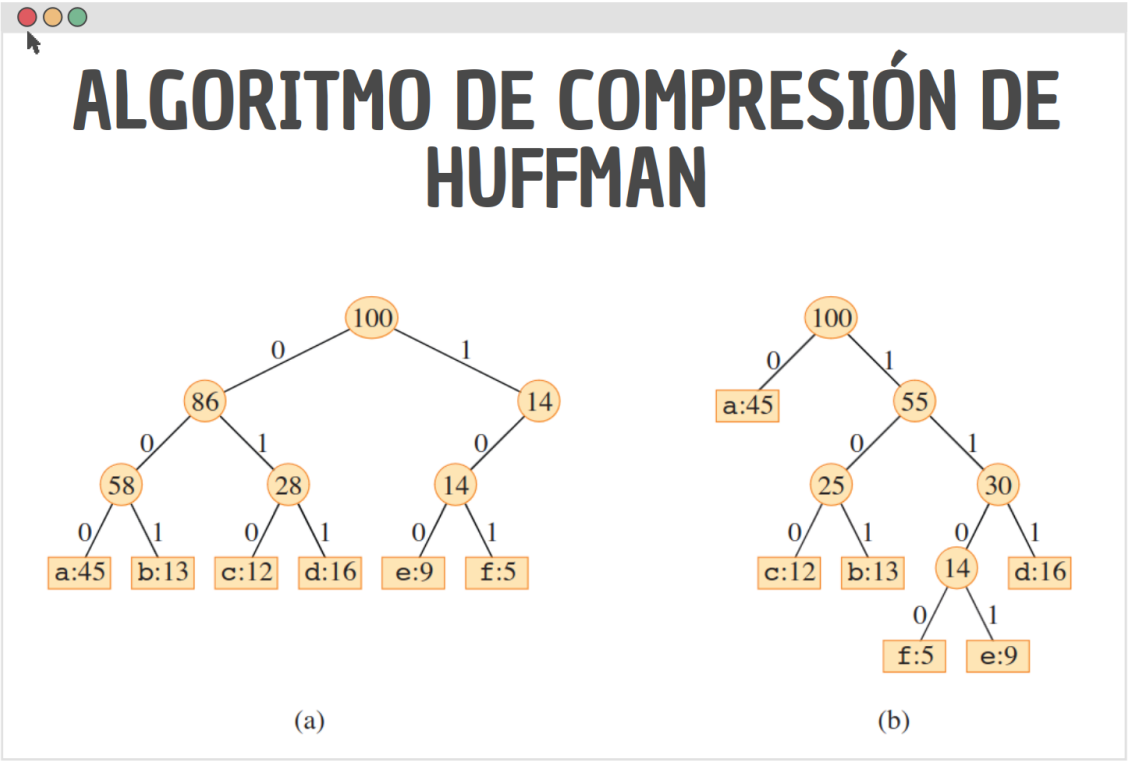


ESTRUCTURAS DE DATOS Y ALGORITMOS con Java



Profesor(a):
Edson Luque Mamani

Estudiantes:
Jorge Luis Mamani Huarsaya
Victor Narciso Mamani Anahua

11 de julio, 2024

Algoritmo de Compresión de Huffman

El algoritmo de Huffman es un método eficiente para la compresión de datos, asignando códigos de longitud variable a cada caracter según su frecuencia de aparición en el texto original. Esta técnica permite una compresión sin pérdida, donde los caracteres más frecuentes son representados con códigos más cortos, y los menos frecuentes con códigos más largos.

Implementación del Algoritmo de Huffman

Para implementar el algoritmo de Huffman en Java, se utilizan dos estructuras principales: la clase `Node` para representar los nodos del árbol de Huffman y la clase `HuffmanCompression` que contiene métodos para construir el árbol de Huffman, generar los códigos Huffman y comprimir y descomprimir textos.

Clase `Node`

La clase `Node` representa un nodo del árbol de Huffman. Cada nodo puede ser un nodo hoja que contiene un caracter y su frecuencia, o un nodo interno que contiene la suma de las frecuencias de sus hijos.

```java
package huffman;

public class Node implements Comparable<Node> {
  public char character;
  public int frequency;
  public Node left;
  public Node right;

  // Constructores y método compareTo omitidos por brevedad
}
```

Clase `HuffmanCompression`

La clase `HuffmanCompression` contiene los métodos principales del algoritmo de Huffman. A continuación se explica cada uno de ellos:

- **Construcción del Árbol de Huffman:** El método `buildHuffmanTree` construye el árbol de Huffman a partir de un mapa de frecuencias de caracteres utilizando una cola de prioridad.

```java
public static Node buildHuffmanTree(Map<Character, Integer> frequencyMap) {
  PriorityQueue<Node> priorityQueue = new PriorityQueue<>();

  for (Map.Entry<Character, Integer> entry : frequencyMap.entrySet()) {
    priorityQueue.add(new Node(entry.getKey(), entry.getValue()),
    ↪  entry.getValue());
  }

  while (priorityQueue.size() > 1) {
    Node left = priorityQueue.poll();
    Node right = priorityQueue.poll();
    Node mergedNode = new Node(left.frequency + right.frequency, left, right);
    priorityQueue.add(mergedNode, mergedNode.frequency);
  }

```

```
  return priorityQueue.poll();
}
```

- **Generación de Códigos Huffman:** El método `generateHuffmanCodes` genera los códigos Huffman recorriendo el árbol de Huffman en preorden.

```
public static Map<Character, String> generateHuffmanCodes(Node root) {
  Map<Character, String> huffmanCodes = new HashMap<>();
  generateCodesHelper(root, "", huffmanCodes);
  return huffmanCodes;
}

private static void generateCodesHelper(Node node, String code, Map<Character,
 ↪  String> huffmanCodes) {
  if (node == null) return;

  if (node.left == null && node.right == null) {
    huffmanCodes.put(node.character, code);
  }

  generateCodesHelper(node.left, code + "0", huffmanCodes);
  generateCodesHelper(node.right, code + "1", huffmanCodes);
}
```

- **Compresión y Descompresión de Texto:** Los métodos `compress` y `decompress` utilizan los códigos Huffman generados para comprimir y descomprimir texto, respectivamente.

```
public static String compress(String text, Map<Character, String> huffmanCodes)
 ↪  {
  StringBuilder compressedText = new StringBuilder();
  for (char c : text.toCharArray()) {
    compressedText.append(huffmanCodes.get(c));
  }
  return compressedText.toString();
}

public static String decompress(String compressedText, Node root) {
  StringBuilder decompressedText = new StringBuilder();
  Node currentNode = root;
  for (char bit : compressedText.toCharArray()) {
    currentNode = (bit == '0') ? currentNode.left : currentNode.right;

    if (currentNode.left == null && currentNode.right == null) {
      decompressedText.append(currentNode.character);
      currentNode = root;
    }
  }
  return decompressedText.toString();
}
```

Implementación de la Cola de Prioridad

La cola de prioridad es una estructura de datos esencial en el algoritmo de Huffman para organizar y acceder eficientemente a los nodos con menor frecuencia. A continuación, se detalla la implementación de la clase `PriorityQueue`, que gestiona los elementos según su prioridad.

Clase `PriorityQueueNode`

La clase `PriorityQueueNode` representa un nodo individual en la cola de prioridad. Cada nodo contiene un dato genérico y una prioridad asociada, que determina su posición en la cola.

```java
package priorityqueue;

public class PriorityQueueNode<T> {
  public T data;
  public int priority;

  public PriorityQueueNode(T data, int priority) {
    this.data = data;
    this.priority = priority;
  }
}
```

En esta implementación, `PriorityQueueNode` es una estructura simple que encapsula el dato y su prioridad, permitiendo la manipulación ordenada dentro de la cola.

Clase `PriorityQueue`

La clase `PriorityQueue` implementa la cola de prioridad utilizando un arreglo dinámico de nodos y un comparador para mantener la estructura ordenada según la prioridad de los nodos.

Constructor

El constructor de `PriorityQueue` inicializa el arreglo dinámico y el comparador que ordenará los nodos según su prioridad.

```java
public class PriorityQueue<T> {
  private ArrayList<PriorityQueueNode<T>> heap;
  private Comparator<PriorityQueueNode<T>> comparator;

  public PriorityQueue() {
    this.heap = new ArrayList<>();
    this.comparator = Comparator.comparingInt(node -> node.priority);
  }
}
```

La clase utiliza `ArrayList` de Java para almacenar los nodos y proporciona métodos para agregar elementos, eliminar el elemento con la prioridad más alta y verificar si la cola está vacía.

Método `add`

El método `add` agrega un nuevo elemento a la cola de prioridad manteniendo la propiedad de orden de la cola.

```java
public void add(T data, int priority) {
  PriorityQueueNode<T> newNode = new PriorityQueueNode<>(data, priority);
  heap.add(newNode);
  heapifyUp(heap.size() - 1);
}
```

Este método inserta un nuevo nodo al final del arreglo y luego lo ajusta hacia arriba (*heapify up*) para mantener el orden de la cola.

Método `poll`

El método `poll` elimina y devuelve el elemento con la prioridad más alta de la cola.

```java
public T poll() {
  if (heap.isEmpty()) {
    throw new NoSuchElementException("Priority queue is empty");
  }

  T result = heap.get(0).data;
  PriorityQueueNode<T> lastNode = heap.remove(heap.size() - 1);
  if (!heap.isEmpty()) {
    heap.set(0, lastNode);
    heapifyDown(0);
  }

  return result;
}
```

Este método extrae el primer nodo (con la prioridad más alta) del arreglo, ajusta el arreglo y luego lo ajusta hacia abajo (*heapify down*) para mantener la estructura de la cola.

Método `heapifyUp`

El método **`heapifyUp`** se utiliza para mantener la propiedad de la cola de prioridad después de agregar un nuevo elemento. Asegura que el nuevo elemento se coloque en la posición correcta para mantener el orden de prioridad ascendente.

```java
private void heapifyUp(int index) {
  while (index > 0) {
    int parentIndex = (index - 1) / 2;
    if (comparator.compare(heap.get(index), heap.get(parentIndex)) >= 0) {
      break;
    }

    swap(index, parentIndex);
    index = parentIndex;
  }
}
```

La función **`heapifyUp`** se ejecuta después de agregar un nuevo nodo al final de la cola de prioridad. Comienza comparando el nodo recién insertado con su padre. Si el nodo tiene una prioridad menor que su padre (según el comparador), intercambia estos nodos. Luego, se mueve hacia arriba en la cola, repitiendo el proceso hasta que el nodo esté en la posición correcta donde su prioridad es mayor o igual que la de sus hijos.

Método `heapifyDown`

El método **`heapifyDown`** se utiliza para mantener la propiedad de la cola de prioridad después de extraer el elemento de mayor prioridad. Asegura que los elementos restantes mantengan el orden de prioridad ascendente.

```
private void heapifyDown(int index) {
  int leftChild, rightChild, smallest;

  while ((leftChild = 2 * index + 1) < heap.size()) {
    rightChild = leftChild + 1;
    smallest = index;

    if (comparator.compare(heap.get(leftChild), heap.get(smallest)) < 0) {
      smallest = leftChild;
    }

    if (rightChild < heap.size() &&
        comparator.compare(heap.get(rightChild), heap.get(smallest)) < 0) {
      smallest = rightChild;
    }

    if (smallest == index) {
      break;
    }

    swap(index, smallest);
    index = smallest;
  }
}
```

La función **`heapifyDown`** se ejecuta después de eliminar el nodo raíz (elemento de mayor prioridad) de la cola de prioridad. Comienza comparando el nodo actual con sus hijos izquierdo y derecho para encontrar el nodo más pequeño según el comparador. Si el nodo actual no es el más pequeño, intercambia el nodo con el hijo más pequeño y continúa hacia abajo en la cola, repitiendo el proceso hasta que el nodo esté en la posición correcta donde su prioridad es menor o igual que la de sus hijos.

Estos métodos son fundamentales para mantener la estructura de la cola de prioridad, garantizando operaciones eficientes de inserción y extracción según la prioridad de los elementos.

Método `swap`

El método **`swap`** se utiliza para intercambiar dos elementos dentro del arreglo de la cola de prioridad. Esto es crucial para mantener el orden y la estructura de la cola después de las operaciones de inserción y extracción.

```
private void swap(int i, int j) {
  PriorityQueueNode<T> temp = heap.get(i);
  heap.set(i, heap.get(j));
  heap.set(j, temp);
}
```

La función **`swap`** toma dos índices como parámetros e intercambia los nodos en estos índices dentro del arreglo **`heap`**. Primero, guarda el nodo en el índice **`i`** en una variable temporal **`temp`**. Luego, asigna el nodo en el índice **`j`** al índice **`i`** y el nodo guardado en **`temp`** al índice **`j`**. Este intercambio garantiza

que la estructura de la cola de prioridad se mantenga correcta después de ajustar los nodos durante las operaciones de `heapifyUp` y `heapifyDown`.

El método `swap` es esencial para mantener la propiedad de orden y prioridad en la cola de prioridad, asegurando operaciones eficientes y correctas en la estructura de datos.

Método `isEmpty`

El método `isEmpty` verifica si la cola de prioridad está vacía.

```
1 public boolean isEmpty() {
2   return heap.isEmpty();
3 }
```

Este método es esencial para verificar si hay elementos pendientes en la cola de prioridad antes de realizar operaciones de extracción.