HEAPS AND PRIORITY QUEUES

DATA STRUCTURES IN JAVA

Sercan Külcü | Data Structures In Java | 10.05.2023

Contents

Heaps

A heap is a data structure that can be used to store a collection of elements in a way that allows for efficient access to the minimum or maximum element. A heap is a complete binary tree, which means that all levels of the tree are filled, except for the last level, which is filled from left to right.

There are two types of heaps: max heaps and min heaps. In a max heap, the value of the root node is always greater than or equal to the values of its children. In a min heap, the value of the root node is always less than or equal to the values of its children.

Heaps can be used to implement priority queues. A priority queue is a data structure that stores a collection of elements, each of which has a priority. The elements in a priority queue are ordered by their priority, with the element with the highest priority at the front of the queue.

Heaps can also be used to implement sorting algorithms. The most common heap sorting algorithm is heap sort. Heap sort works by first building a max heap from the input array. The root node of the max heap is then removed and placed at the end of the array. The heap is then rebuilt without the root node. This process is repeated until the array is sorted.

Heaps are a powerful data structure that can be used for a variety of tasks. They are efficient for access to the minimum or maximum element, and they can be used to implement priority queues and sorting algorithms.

Operations on Heaps

There are a number of operations that can be performed on heaps. Some of the most common operations include:

- Insert: An element can be inserted into a heap by adding it to the end of the tree and then heapifying the tree.

- Delete: An element can be deleted from a heap by removing the root node and then heapifying the tree.
- GetMin: The minimum element in a heap can be obtained by accessing the root node.
- GetMax: The maximum element in a heap can be obtained by accessing the root node.
- Heapify: A heap can be heapified by recursively sorting the subtrees of the root node.

Implementation of Heaps in Java

Heaps can be implemented in Java using a variety of data structures. One common implementation is to use an array. The elements of the array are stored in level order, with the root node at index 0 and the leaves at the bottom of the array.

The following code shows how to implement a max heap in Java:

```
public class MaxHeap {


   private int[] data;

   private int size;


   public MaxHeap(int capacity) {

      data = new int[capacity];

   }


   public void insert(int element) {

      data[size++] = element;

      heapify(size - 1);
```

```
}

public int getMax() {
    return data[0];
}

public void deleteMax() {
    data[0] = data[size - 1];
    size--;
    heapify(0);
}

private void heapify(int index) {
    int left = 2 * index + 1;
    int right = 2 * index + 2;
    int largest = index;

    if (left < size && data[left] > data[largest]) {
        largest = left;
    }

    if (right < size && data[right] > data[largest]) {
```

```
            largest = right;
        }

        if (largest != index) {
            int temp = data[index];
            data[index] = data[largest];
            data[largest] = temp;
            heapify(largest);
        }
    }
}
```

This implementation of a max heap supports the following operations:

- Insert: An element can be inserted into the heap by calling the insert() method.
- DeleteMax: The maximum element can be deleted from the heap by calling the deleteMax() method.
- GetMax: The maximum element in the heap can be obtained by calling the getMax() method.

Heaps are a powerful data structure that can be used for a variety of tasks. They are efficient for access to the minimum or maximum element, and they can be used to implement priority queues and sorting algorithms.

Binary Heaps

A binary heap is a data structure that can be used to store a collection of elements in a way that allows for efficient access to the minimum or maximum element. A binary heap is a complete binary tree, which means that all levels of the tree are filled, except for the last level, which is filled from left to right.

There are two types of binary heaps: max heaps and min heaps. In a max heap, the value of the root node is always greater than or equal to the values of its children. In a min heap, the value of the root node is always less than or equal to the values of its children.

Binary heaps can be used to implement priority queues. A priority queue is a data structure that stores a collection of elements, each of which has a priority. The elements in a priority queue are ordered by their priority, with the element with the highest priority at the front of the queue.

Binary heaps can also be used to implement sorting algorithms. The most common binary heap sorting algorithm is heap sort. Heap sort works by first building a max heap from the input array. The root node of the max heap is then removed and placed at the end of the array. The heap is then rebuilt without the root node. This process is repeated until the array is sorted.

Binary heaps are a powerful data structure that can be used for a variety of tasks. They are efficient for access to the minimum or maximum element, and they can be used to implement priority queues and sorting algorithms.

Operations on Binary Heaps

There are a number of operations that can be performed on binary heaps. Some of the most common operations include:

- Insert: An element can be inserted into a binary heap by adding it to the end of the tree and then heapifying the tree.

- Delete: An element can be deleted from a binary heap by removing the root node and then heapifying the tree.
- GetMin: The minimum element in a binary heap can be obtained by accessing the root node.
- GetMax: The maximum element in a binary heap can be obtained by accessing the root node.
- Heapify: A binary heap can be heapified by recursively sorting the subtrees of the root node.

Fibonacci Heaps

A Fibonacci heap is a data structure that can be used to store a collection of elements in a way that allows for efficient access to the minimum or maximum element. A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a lazy manner, postponing the work for later operations.

In computer science, a Fibonacci heap is a data structure for priority queue operations, consisting of a collection of heap-ordered trees. It has a better amortized running time than many other priority queue data structures including the binary heap and binomial heap. Michael L. Fredman and Robert E. Tarjan developed Fibonacci heaps in 1984 and published them in a scientific journal in 1987. Fibonacci heaps are named after the Fibonacci numbers, which are used in their running time analysis.

Operations on Fibonacci Heaps

There are a number of operations that can be performed on Fibonacci heaps. Some of the most common operations include:

- Insert: An element can be inserted into a Fibonacci heap by adding it to the end of the tree and then consolidating the heap.
- DeleteMin: The minimum element in a Fibonacci heap can be obtained by removing the root node and then consolidating the heap.

- DecreaseKey: The key of an element in a Fibonacci heap can be decreased by updating the element's key and then consolidating the heap.
- Merge: Two Fibonacci heaps can be merged into a single Fibonacci heap.

Skew Heaps

A skew heap is a data structure that can be used to store a collection of elements in a way that allows for efficient access to the minimum or maximum element. A skew heap is a heap, which means that the elements are stored in a way that the minimum or maximum element can be found quickly. Skew heaps are also binomial heaps, which means that they can be merged efficiently.

Skew heaps were invented by Vadim V. Makeev in 1985. They are a more efficient data structure than binary heaps for some operations, such as decrease-key and delete-min. Skew heaps are also more efficient than Fibonacci heaps for some operations, such as merge.

Operations on Skew Heaps

There are a number of operations that can be performed on skew heaps. Some of the most common operations include:

- Insert: An element can be inserted into a skew heap by adding it to the end of the tree and then splaying the tree.
- DeleteMin: The minimum element in a skew heap can be obtained by removing the root node and then splaying the tree.
- DecreaseKey: The key of an element in a skew heap can be decreased by updating the element's key and then splaying the tree.
- Merge: Two skew heaps can be merged into a single skew heap.

Binomial Heaps

A binomial heap is a data structure that can be used to store a collection of elements in a way that allows for efficient access to the minimum or maximum element. A binomial heap is a heap, which means that the elements are stored in a way that the minimum or maximum element can be found quickly. Binomial heaps are also union-find data structures, which means that they can be merged efficiently.

Binomial heaps were invented by Michael L. Fredman and Robert E. Tarjan in 1984. They are a more efficient data structure than binary heaps for some operations, such as decrease-key and delete-min. Binomial heaps are also more efficient than Fibonacci heaps for some operations, such as merge.

Operations on Binomial Heaps

There are a number of operations that can be performed on binomial heaps. Some of the most common operations include:

- Insert: An element can be inserted into a binomial heap by adding it to the end of the tree and then merging the tree with a binomial tree of degree 1.
- DeleteMin: The minimum element in a binomial heap can be obtained by removing the root node and then merging the tree with the binomial trees of all the children of the root node.
- DecreaseKey: The key of an element in a binomial heap can be decreased by updating the element's key and then merging the tree with the binomial trees of all the children of the element.
- Merge: Two binomial heaps can be merged into a single binomial heap.

Union-Find

A union-find data structure is a data structure that can be used to track the connected components of a set. A connected component is a subset of a set in which any two elements can be reached from each other by following edges.

Union-find data structures are often used to solve problems in graph theory, such as finding the connected components of a graph and finding the minimum spanning tree of a graph.

Operations on Union-Find Data Structures

There are a number of operations that can be performed on union-find data structures. Some of the most common operations include:

- Union: The union operation combines two connected components into a single connected component.
- Find: The find operation finds the connected component that contains a given element.
- Count: The count operation returns the number of connected components in the data structure.

Implementation of Union-Find Data Structures in Java

Union-find data structures can be implemented in Java using a variety of data structures. One common implementation is to use an array. The elements of the array are stored in index order, with the root node of each connected component at index 0 and the leaves at the bottom of the array.

The following code shows how to implement a union-find data structure in Java:

```
public class UnionFind {
```

```
private int[] data;
private int[] rank;

public UnionFind(int n) {
    data = new int[n];
    rank = new int[n];
    for (int i = 0; i < n; i++) {
        data[i] = i;
        rank[i] = 0;
    }
}

public void union(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);
    if (rootX == rootY) {
        return;
    }
    if (rank[rootX] < rank[rootY]) {
        data[rootX] = rootY;
    } else {
        data[rootY] = rootX;
```

```
        if (rank[rootX] == rank[rootY]) {
            rank[rootX]++;
        }
    }
}

public int find(int x) {
    if (data[x] == x) {
        return x;
    } else {
        return data[x] = find(data[x]);
    }
}

public boolean connected(int x, int y) {
    return find(x) == find(y);
}

public int count() {
    int count = 0;
    for (int i = 0; i < data.length; i++) {
        if (data[i] == i) {
```

```
            count++;
        }
    }
    return count;
  }
}
```

Union-find data structures are a powerful data structure that can be used for a variety of tasks. They are efficient for finding connected components and minimum spanning trees, and they can be used to solve a variety of problems in graph theory.

Priority Queue

A priority queue is a data structure that can be used to store a collection of elements in a way that allows for efficient access to the element with the highest priority. The priority of an element is a value that is associated with the element, and it is used to determine the order in which the elements are processed.

Priority queues are often used to solve problems in scheduling, such as finding the shortest path between two nodes in a graph and finding the maximum flow in a network.

Operations on Priority Queues

There are a number of operations that can be performed on priority queues. Some of the most common operations include:

- Enqueue: The enqueue operation adds an element to the priority queue.
- Dequeue: The dequeue operation removes the element with the highest priority from the priority queue.
- Peek: The peek operation returns the element with the highest priority from the priority queue without removing it.
- IsEmpty: The isEmpty operation returns true if the priority queue is empty and false otherwise.

Implementation of Priority Queues in Java

Priority queues can be implemented in Java using a variety of data structures. One common implementation is to use a heap. A heap is a data structure that stores elements in a way that allows for efficient access to the element with the highest priority.

The following code shows how to implement a priority queue in Java using a heap:

```
public class PriorityQueue {
```

```
private Node[] data;
private int size;

public PriorityQueue() {
    data = new Node[0];
    size = 0;
}

public void enqueue(int element) {
    Node newNode = new Node(element);
    data = Arrays.copyOf(data, size + 1);
    data[size] = newNode;
    size++;
    heapify(size - 1);
}

public int dequeue() {
    if (size == 0) {
        return -1;
    }
    int min = data[0].element;
```

```
    data[0] = data[size - 1];
    size--;
    heapify(0);
    return min;
}

public int peek() {
    if (size == 0) {
        return -1;
    }
    return data[0].element;
}

public boolean isEmpty() {
    return size == 0;
}

private void heapify(int index) {
    while (index < size / 2) {
        int leftChild = 2 * index + 1;
        int rightChild = 2 * index + 2;
        int largest = index;
```

```
        if (leftChild < size && data[leftChild].element > data[largest].element) {
            largest = leftChild;
        }
        if (rightChild < size && data[rightChild].element > data[largest].element) {
            largest = rightChild;
        }
        if (largest != index) {
            swap(index, largest);
            heapify(largest);
        }
    }
  }

  private void swap(int index1, int index2) {
    Node temp = data[index1];
    data[index1] = data[index2];
    data[index2] = temp;
  }
}
```

Priority queues are a powerful data structure that can be used for a variety of tasks. They are efficient for access to the element with the

highest priority, and they can be used to solve a variety of problems in scheduling and graph theory.