INTRODUCTION TO FIBONACCI HEAPS

Fibonacci Heaps are a type of priority queue data structure, an advanced form of heaps that supports a collection of operations efficiently. Invented by Michael L. Fredman and Robert E. Tarjan in 1984, Fibonacci Heaps are particularly noted for their amortized time complexity, which is better than that of other heap structures for several operations. They are especially useful in applications such as Dijkstra's shortest path algorithm and other graph algorithms where multiple priority queue operations are involved.

Structure of a Fibonacci Heap

A Fibonacci Heap is a collection of trees satisfying the minimum-heap property, where the key of a parent node is always less than or equal to the keys of its children. It is more flexible than binary or binomial heaps due to its relaxed constraints, leading to improved performance for some operations.

Key features of Fibonacci Heaps include:

- Collection of trees: Unlike binary heaps, Fibonacci Heaps are composed of a forest of heap-ordered trees.

- Marked nodes: Nodes that have lost a child since they became a child of another node are marked to facilitate certain operations.

- Lazy deletion: Elements are not immediately deleted; rather, nodes are marked for deletion, improving amortized performance.

Fibonacci Heap Operations

Insert Operation

To insert a new element into a Fibonacci Heap, create a new node and add it to the root list. The root list is a doubly linked list of tree roots.

1. Create a new node with the given key.

2. Add the node to the root list.

3. Update the minimum pointer if necessary.

Time Complexity: O(1) amortized.

Example:

Suppose we have an empty Fibonacci Heap and we insert the keys 3, 18, 39.

1. Insert 3: Create a single-node tree with root 3.

2. Insert 18: Create another single-node tree with root 18.

3. Insert 39: Create another single-node tree with root 39.

The heap now contains three separate trees:

Root list: 3 <-> 18 <-> 39

Extract Minimum

The extract-minimum operation removes the minimum node from the root list and consolidates the trees to maintain the heap property.

1. Remove the minimum node (pointed by the min pointer).

2. Add its children to the root list.

3. Consolidate the trees in the root list to ensure each tree has a unique degree.

Time Complexity: O(log n) amortized.

Example:

Using the previous example, let's extract the minimum (3).

1. Remove 3 from the root list.

2. There are no children to add to the root list.

3. Consolidate the trees (no changes needed as the degrees are unique).

The heap now contains:

Root list: 18 <-> 39

Decrease Key

The decrease-key operation decreases the key of a node. If this operation violates the heap property, a cascading cut is performed.

1. Decrease the key of the node.

2. If the new key is less than the parent's key, cut the node and add it to the root list.

3. Perform cascading cuts if necessary.

Time Complexity: O(1) amortized.

Example:

Consider the previous heap with keys 18 and 39, and suppose the node with key 39 has a child with key 45. Now, decrease the key of 45 to 10.

1. Change 45 to 10.

2. Since 10 is less than its parent's key (39), cut the node and add it to the root list.

3. No cascading cuts are needed in this example.

The heap now looks like:

Root list: 18 <-> 39 <-> 10

Delete Operation

The delete operation is performed by decreasing the key of the node to be deleted to negative infinity, making it the minimum element, and then extracting it.

1. Decrease the key to negative infinity.
2. Perform extract-min.

Time Complexity: O(log n) amortized.

Example:

To delete node with key 18:

1. Decrease key of 18 to negative infinity.
2. Extract minimum (negative infinity).

The heap now contains:

Root list: 39 <-> 10

Union Operation

Union of two Fibonacci Heaps involves concatenating their root lists and updating the min pointer accordingly.

1. Concatenate the root lists of the two heaps.
2. Update the min pointer to point to the smaller key of the two heaps.

Time Complexity: O(1) amortized.

Example:

If we have two heaps:

Heap 1: Root list: 3 <-> 17

Heap 2: Root list: 25 <-> 7

Union results in:

Union: Root list: 3 <-> 17 <-> 25 <-> 7

Minimum: 3

Visual Examples

Inserting Nodes:

Insert 5, 2, 8 into an empty heap.

Heap: Root list: 5 <-> 2 <-> 8

Minimum: 2

Extracting Minimum:

Heap: Root list: 2 <-> 8 (2 is minimum)

Extract 2:

Heap after extracting 2: Root list: 8

Minimum: 8

Decreasing Key:

Heap: Root list: 8 <-> 3 (3 has child 5)

Decrease key of 5 to 1:

Heap: Root list: 8 <-> 3 <-> 1 (1 is new minimum)

Applications of Fibonacci Heaps

Fibonacci Heaps are particularly useful in:

- Graph Algorithms: Dijkstra's and Prim's algorithms benefit significantly from Fibonacci Heaps due to efficient decrease-key operations.
- Network Optimization: Used in network flow and shortest path problems.
- Priority Queue Operations: Where multiple decrease-key and delete operations are frequent.

Conclusion

Fibonacci Heaps offer a powerful alternative to traditional heap structures with their efficient amortized time complexities for key operations. Their utility in graph algorithms and other computational problems demonstrates their significance in computer science. Understanding Fibonacci Heaps requires a grasp of their structure and the ability to implement and visualize the various operations, making them a fascinating subject for both theoretical and practical exploration.