**COURSE CODE:** DJS22ITL502 **DATE: 7-10-2024**

**COURSE NAME:** Advanced Data Structures Laboratory **CLASS:** TY B. TECH

**NAME:** Anish Sharma **DIV:** IT 1 **ROLL:** I011

# EXPERIME NT NO. 3

**CO/LO:** Choose appropriate data structure and use it to design algorithm for solving a specific problem

**AIM / OBJECTIVE:** To implement various operations on Binomial Heap.

# DESCRIPTION OF EXPERIMENT:

# Properties of Binomial Heap:

# Collection of Binomial Trees: A heap is made of binomial trees, with each tree structured recursively.

# Heap Property: Each tree obeys the min-heap property, with the smallest key at the root.

# Logarithmic Merging: Merging two heaps takes O(logn) by combining trees of the same order.

# Find Min: Locating the minimum element takes O(logn)) by scanning root nodes.

# Delete Min: Deleting the minimum element and restructuring takes O(logn)

**TECHNOLOGY STACK USED: C, C++, JAVA**

**SOURCE CODE:**

import java.io.\*;

class BinomialHeapNode {

    int key, degree;

    BinomialHeapNode parent;

    BinomialHeapNode sibling;

    BinomialHeapNode child;

    public BinomialHeapNode(int k)

    {

        key = k;

        degree = 0;

        parent = null;

        sibling = null;

        child = null;

    }

    public BinomialHeapNode reverse(BinomialHeapNode sibl)

    {

        BinomialHeapNode ret;

        if (sibling != null)

            ret = sibling.reverse(this);

        else

            ret = this;

        sibling = sibl;

        return ret;

    }

    public BinomialHeapNode findMinNode()

    {

        BinomialHeapNode x = this, y = this;

        int min = x.key;

        while (x != null) {

            if (x.key < min) {

                y = x;

                min = x.key;

            }

            x = x.sibling;

        }

        return y;

    }

    public BinomialHeapNode findANodeWithKey(int value)

    {

        BinomialHeapNode temp = this, node = null;

        while (temp != null) {

            if (temp.key == value) {

                node = temp;

                break;

            }

            if (temp.child == null)

                temp = temp.sibling;

            else {

                node = temp.child.findANodeWithKey(value);

                if (node == null)

                    temp = temp.sibling;

                else

                    break;

            }

        }

        return node;

    }

    public int getSize()

    {

        return (

            1 + ((child == null) ? 0 : child.getSize())

            + ((sibling == null) ? 0 : sibling.getSize()));

    }

}

class BinomialHeap {

    private BinomialHeapNode Nodes;

    private int size;

    public BinomialHeap()

    {

        Nodes = null;

        size = 0;

    }

    public boolean isEmpty() { return Nodes == null; }

    public int getSize() { return size; }

    public void makeEmpty()

    {

        Nodes = null;

        size = 0;

    }

    public void insert(int value)

    {

        if (value > 0) {

            BinomialHeapNode temp

                = new BinomialHeapNode(value);

            if (Nodes == null) {

                Nodes = temp;

                size = 1;

            }

            else {

                unionNodes(temp);

                size++;

            }

        }

    }

    private void merge(BinomialHeapNode binHeap)

    {

        BinomialHeapNode temp1 = Nodes, temp2 = binHeap;

        while ((temp1 != null) && (temp2 != null)) {

            if (temp1.degree == temp2.degree) {

                BinomialHeapNode tmp = temp2;

                temp2 = temp2.sibling;

                tmp.sibling = temp1.sibling;

                temp1.sibling = tmp;

                temp1 = tmp.sibling;

            }

            else {

                if (temp1.degree < temp2.degree) {

                    if ((temp1.sibling == null)

                        || (temp1.sibling.degree

                            > temp2.degree)) {

                        BinomialHeapNode tmp = temp2;

                        temp2 = temp2.sibling;

                        tmp.sibling = temp1.sibling;

                        temp1.sibling = tmp;

                        temp1 = tmp.sibling;

                    }

                    else {

                        temp1 = temp1.sibling;

                    }

                }

                else {

                    BinomialHeapNode tmp = temp1;

                    temp1 = temp2;

                    temp2 = temp2.sibling;

                    temp1.sibling = tmp;

                    if (tmp == Nodes) {

                        Nodes = temp1;

                    }

                    else {

                    }

                }

            }

        }

        if (temp1 == null) {

            temp1 = Nodes;

            while (temp1.sibling != null) {

                temp1 = temp1.sibling;

            }

            temp1.sibling = temp2;

        }

        else {

        }

    }

    private void unionNodes(BinomialHeapNode binHeap)

    {

        merge(binHeap);

        BinomialHeapNode prevTemp = null, temp = Nodes,

                        nextTemp = Nodes.sibling;

        while (nextTemp != null) {

            if ((temp.degree != nextTemp.degree)

                || ((nextTemp.sibling != null)

                    && (nextTemp.sibling.degree

                        == temp.degree))) {

                prevTemp = temp;

                temp = nextTemp;

            }

            else {

                if (temp.key <= nextTemp.key) {

                    temp.sibling = nextTemp.sibling;

                    nextTemp.parent = temp;

                    nextTemp.sibling = temp.child;

                    temp.child = nextTemp;

                    temp.degree++;

                }

                else {

                    if (prevTemp == null) {

                        Nodes = nextTemp;

                    }

                    else {

                        prevTemp.sibling = nextTemp;

                    }

                    temp.parent = nextTemp;

                    temp.sibling = nextTemp.child;

                    nextTemp.child = temp;

                    nextTemp.degree++;

                    temp = nextTemp;

                }

            }

            nextTemp = temp.sibling;

        }

    }

    public int findMinimum()

    {

        return Nodes.findMinNode().key;

    }

    public void delete(int value)

    {

        if ((Nodes != null)

            && (Nodes.findANodeWithKey(value) != null)) {

            decreaseKeyValue(value, findMinimum() - 1);

            extractMin();

        }

    }

    public void decreaseKeyValue(int old\_value,

                                int new\_value)

    {

        BinomialHeapNode temp

            = Nodes.findANodeWithKey(old\_value);

        if (temp == null)

            return;

        temp.key = new\_value;

        BinomialHeapNode tempParent = temp.parent;

        while ((tempParent != null)

            && (temp.key < tempParent.key)) {

            int z = temp.key;

            temp.key = tempParent.key;

            tempParent.key = z;

            temp = tempParent;

            tempParent = tempParent.parent;

        }

    }

    public int extractMin()

    {

        if (Nodes == null)

            return -1;

        BinomialHeapNode temp = Nodes, prevTemp = null;

        BinomialHeapNode minNode = Nodes.findMinNode();

        while (temp.key != minNode.key) {

            prevTemp = temp;

            temp = temp.sibling;

        }

        if (prevTemp == null) {

            Nodes = temp.sibling;

        }

        else {

            prevTemp.sibling = temp.sibling;

        }

        temp = temp.child;

        BinomialHeapNode fakeNode = temp;

        while (temp != null) {

            temp.parent = null;

            temp = temp.sibling;

        }

        if ((Nodes == null) && (fakeNode == null)) {

            size = 0;

        }

        else {

            if ((Nodes == null) && (fakeNode != null)) {

                Nodes = fakeNode.reverse(null);

                size = Nodes.getSize();

            }

            else {

                if ((Nodes != null) && (fakeNode == null)) {

                    size = Nodes.getSize();

                }

                else {

                    unionNodes(fakeNode.reverse(null));

                    size = Nodes.getSize();

                }

            }

        }

        return minNode.key;

    }

    public void displayHeap()

    {

        System.out.print("\nHeap : ");

        displayHeap(Nodes);

        System.out.println("\n");

    }

    private void displayHeap(BinomialHeapNode r)

    {

        if (r != null) {

            displayHeap(r.child);

            System.out.print(r.key + " ");

            displayHeap(r.sibling);

        }

    }

}

public class GFG {

    public static void main(String[] args)

    {

        BinomialHeap binHeap = new BinomialHeap();

        binHeap.insert(12);

        binHeap.insert(8);

        binHeap.insert(5);

        binHeap.insert(15);

        binHeap.insert(7);

        binHeap.insert(2);

        binHeap.insert(9);

        System.out.println("Size of the binomial heap is "

                        + binHeap.getSize());

        binHeap.displayHeap();

        binHeap.delete(15);

        binHeap.delete(8);

        System.out.println("Size of the binomial heap is "

                        + binHeap.getSize());

        binHeap.displayHeap();

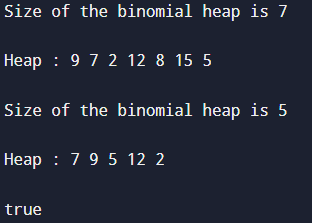
        binHeap.makeEmpty();

        System.out.println(binHeap.isEmpty());

    }

}

**OUTPUT:**

****

**CONCLUSION:** In this experiment we understood the implementation of binomial heap.

# REFERENCES:

1. Peter Brass, “Advanced Data Structures”, Cambridge University Press, 2008
2. Robert Sedgewick & Kevin Wayne, “Algorithms”, 4th Edition, Addison-Wesley Professional, 2011.