

[4] Quick Sort dan Branching

Quick Sort

- ✧ Quick sort merupakan algoritma pengurutan data yang menggunakan implementasi rekursif.
- ✧ Sehingga, algoritma ini akan membagi *array* data yang tidak terurut menjadi beberapa bagian.
- ✧ Data dapat dibagi menjadi dua bagian atau bahkan lebih tergantung dari kondisi data pada setiap bagian yang telah dibagi.
- ✧ Konsep dari *quick sort* ini adalah membuat partisi/membagi array menjadi kumpulan elemen array yang nilainya \leq pivot dan kumpulan elemen array yang nilainya $>$ pivot.

Algoritma:

- ✧ Inisialisasi beberapa variabel sebagai berikut.
 - **current index** adalah -1
 - **swap marker** adalah -1
 - **pivot**, dapat berupa.
 - elemen pertama;
 - elemen terakhir;
 - memilih elemen secara random; dan
 - memilih elemen median data.
- ✧ Memanggil function **partisi**; susun pada array bagian sekarang.
 1. current index tambah dengan 1.
 2. Apakah nilai array data pada indeks ke-*current index* \leq nilai pivot?
 - 1.1 Ya, maka swap marker ditambah dengan 1.
 - 1.1.1 Apakah nilai swap marker dengan current index berbeda?
 - 1.1.1.1 Ya, maka tukar posisi nilai array indeks ke-*current index* dan nilai array indeks ke-*swap marker*.
 - 1.1.1.2 Tidak, maka lakukan proses nomor (3).
 - 1.2 Tidak, maka lakukan proses nomor (3).
 3. Ulang proses nomor (1) - (2) hingga current index == panjang array - 1.
 4. Lakukan pemisahan array data dengan membagi data menjadi dua bagian, kiri pivot dan kanan pivot.
- ✧ Memanggil function **quick sort** secara rekursif pada partisi kiri dan kanan.
 - Rekursi berhenti jika tiap partisi hanya menyisakan 1 elemen saja

Ada dua function dalam quick sort:

--partition

--quick sort

Berikut contoh program dengan implementasi algoritma quick sort:

```
public class QuickSort {
    int partition(int arr[], int low, int high){
        int pivot = arr[high];
        int i = (low-1); // index of smaller element
        for (int j=low; j<high; j++) {
            // If current element is smaller than the pivot
            if (arr[j] <= pivot) {
                i++;
                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        // swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;

        return i+1; // kembalikan posisi pivot sekarang
    }

    void quickSort(int A[], int low, int high) {
        if(low < high) {
            //menyimpan posisi elemen pivot
            int pivotIndex = partition(A, low, high);

            // pada baris ini array A telah berubah akibat partition //

            quickSort(A, low, pivotIndex-1); //menyortir sisi kiri pivot
            quickSort(A, pivotIndex+1, high) ; //menyortir sisi kanan pivot
        }
    }

    public static void main(String[] args) {
        int A[] = {9,7,8,3,2,1};
        QuickSort qs = new QuickSort();
        System.out.println("Sebelum quick sort: ");
        for(int a: A) {
            System.out.print(a);
        }
        qs.quickSort(A, 0, A.length-1);

        System.out.println("\n\nSesudah quick sort: ");
        for(int a: A) {
            System.out.print(a);
        }
    }
}
```

Kelebihan

- ✧ Cenderung yang tercepat dibandingkan algoritma sorting lain.
- ✧ Melakukan *in-place sorting*, tidak memerlukan memori tambahan untuk melakukan sorting.
- ✧ Efisien karena bekerja dengan baik untuk data yang banyak.

Kekurangan

- ✧ Sedikit kesalahan dalam penulisan program dapat membuat quick sort tidak akan berhenti atau hasilnya salah.
- ✧ Bukan algoritma sorting yang stabil. Apabila algoritma ini dipakai untuk keadaan yang cocok, maka *quick sort* berjalan dengan cepat, efisien, dan efektif.
- ✧ Karena penerapannya secara rekursif, jika terjadi *worst case* (kasus terburuk), algoritma ini dapat menghabiskan memori dan membuat program *hang*. Misalnya, data yang digunakan sudah terurut atau hampir terurut atau banyak elemen data yang duplikat.

Solusi

- ✧ ***Randomised Quick Sort***
Pilih pivot secara acak untuk menghindari pola terurut yang menyebabkan partisi tidak seimbang.
- ✧ ***Median-of-Three Partitioning***
Pilih pivot sebagai median dari elemen pertama, tengah, dan terakhir untuk meningkatkan peluang mendapatkan partisi yang lebih seimbang.

Contoh:

[29, 10, 14, 37, 13]

- Elemen pertama: 29
- Elemen tengah: 14
- Elemen terakhir: 13

Dari tiga elemen ini: 13, 14, 29, median adalah 14. Maka gunakanlah 14 sebagai nilai pivot.

Saudara dapat *scan QR Code* berikut untuk dapat memahami lebih dalam mengenai tahapan pengoperasian algoritma *quick sort*.

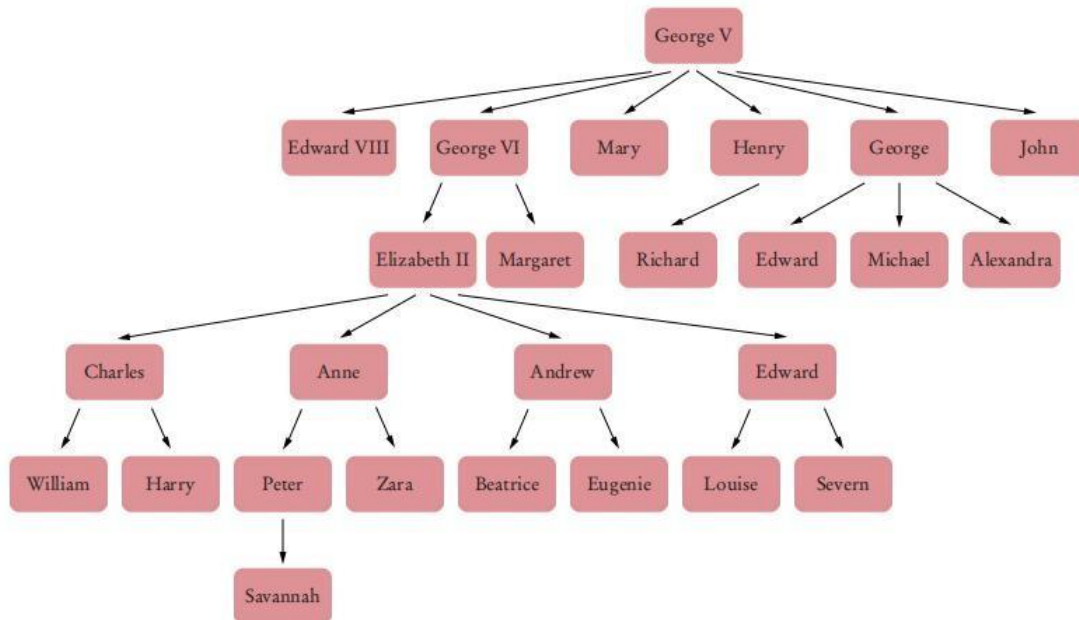


Tree

Konsep Dasar Tree

Tree adalah struktur data non-linear bersifat hierarkis yang tersusundari kumpulan node.

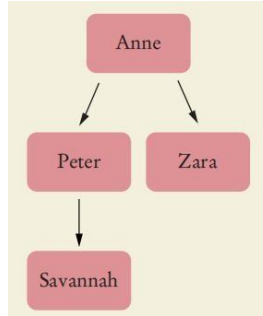
Perhatikan contoh bagan *tree* berikut.



Gambar 1. Family tree

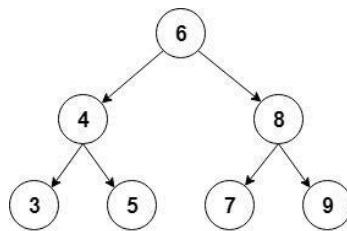
Berikut ini daftar istilah dalam *tree* dan contoh penggunaannya sesuai bagan diatas.

Istilah	Definisi	Contoh (menggunakan Gambar 1)
Node	<ul style="list-style-type: none">✧ Komponen dasar <i>tree</i>✧ <i>Tree</i> tersusun atas kumpulan <i>node</i> yang terhubung	<i>Tree</i> pada Gambar 1 memiliki 26 node
Root	<i>Node</i> yang tidak memiliki <i>parent</i>	George V
Child	Semua <i>node</i> kecuali <i>root</i> adalah <i>child node</i>	Anak (<i>children</i>) dari Elizabeth II adalah Charles, Anne, Andrew, dan Edward
Leaf	<i>Node</i> yang tidak memiliki anak (<i>child node</i>)	<i>Tree</i> pada Gambar 1 memiliki 16 daun (<i>leaves</i>), contohnya: William, Harry, dan Savannah
Parent	<i>Node</i> yang memiliki ≥ 1 anak (<i>child</i>)	Elizabeth II adalah orang tua (<i>parent</i>) dari Charles
Sibling	<i>Node</i> dengan <i>parent</i>	Charles dan Anne

	yang sama	adalah saudara kandung (<i>sibling</i>)
Subtree	bagian dari struktur pohon (<i>tree</i>) yang terdiri dari suatu simpul (<i>node</i>) dan semua keturunannya dalam hierarki pohon tersebut.	<i>Subtree</i> dengan <i>root</i> Anne adalah:  <pre> graph TD Anne[Anne] --> Peter[Peter] Anne --> Zara[Zara] Peter --> Savannah[Savannah] </pre>

Binary Tree

Binary tree merupakan struktur data yang menggunakan konsep rekursif, tiap nodenya maksimal memiliki 2 *child*. Salah satu jenis *binary tree* yang umum digunakan adalah *binary search tree*, yaitu *binary tree* yang *subtree* sebelah kirinya memiliki nilai yang lebih kecil dari *subtree* sebelah kanan. Perhatikan contoh berikut.



Contoh program:

```

public class BinarySearchTree {
    class Node {
        int value;
        Node left, right;

        // constructor
        Node(int value) {
            this.value = value;
            left = right = null;
        }
    }

    Node root;

    // constructor
    BinarySearchTree() {
        root = null;
    }

    void insert(int value) {
        root = insertRecursive(root, value);
    }
}

```

```

}

/* A recursive function to insert a new value in BST */
private Node insertRecursive(Node current, int value) {
    /* If the tree is empty, return a new node */
    if (current == null) {
        return new Node(value);
    }

    /* Otherwise, recur down the tree */
    if (value < current.value) {
        current.left = insertRecursive(current.left, value);
    } else if (value > current.value) {
        current.right = insertRecursive(current.right, value);
    }

    /* return the (unchanged) node pointer */
    return current;
}

void delete(int value) {
    root = deleteRecursive(root, value);
}

/* A recursive function to delete a value in BST */
private Node deleteRecursive(Node current, int value) {
    /* Base Case: If the tree is empty */
    if (current == null)
        return current;

    /* Otherwise, recur down the tree */
    if (value < current.value) {
        current.left = deleteRecursive(current.left, value);
    } else if (value > current.value) {
        current.right = deleteRecursive(current.right, value);
    }

    // if key is same as root's key, then This is the node
    // to be deleted
    else {
        // node with only one child or no child
        if (current.left == null)
            return current.right;
        else if (current.right == null)
            return current.left;

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        current.value = minValue(current.right);

        // Delete the inorder successor
        current.right = deleteRecursive(current.right, current.value);
    }
}

```

```

        return current;
    }

    int minValue(Node current) {
        int minValue = current.value;
        while (current.left != null) {
            minValue = current.left.value;
            current = current.left;
        }

        return minValue;
    }

    void inorder() {
        inorderRecursive(root);
    }

    // A utility function to do inorder traversal of BST
    void inorderRecursive(Node current) {
        if (current != null) {
            inorderRecursive(current.left);
            System.out.print(current.value + " ");
            inorderRecursive(current.right);
        }
    }

    public static void main(String args[]) {
        BinarySearchTree tree = new BinarySearchTree();

        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.insert(70);
        tree.insert(60);
        tree.insert(80);

        System.out.println("Inorder traversal of the given tree");
        tree.inorder();

        System.out.println("\nDelete 20");
        tree.delete(20);
        System.out.println("Inorder traversal of the modified tree");
        tree.inorder();

        System.out.println("\nDelete 30");
        tree.delete(30);
        System.out.println("Inorder traversal of the modified tree");
        tree.inorder();

        System.out.println("\nDelete 50");
        tree.delete(50);
    }

```

```
        System.out.println("Inorder traversal of the modified tree");
        tree.inorder();
    }
}
```

- ✧ Proses pembacaan *binary tree* disebut sebagai ***traversal***.
- ✧ Proses *traversal* melakukan kunjungan pada tiap *node* pada suatu *binary tree* tepat satu kali.
- ✧ Dengan melakukan kunjungan secara lengkap, kemudian akan didapatkan urutan informasi secara linier yang tersimpan dalam *binary tree*.
- ✧ Terdapat tiga cara *traversal* pada *binary tree*, yakni sebagai berikut.

1. Traversal Preorder (Depth First Order)

Yaitu dengan mencetak isi *node* yang dikunjungi lalu melakukan kunjungan ke-*subtree* kiri dan selanjutnya ke-*subtree* kanan. Berikut tahapan algoritma tersebut.

- ✧ Jika *tree* kosong, maka keluar.
- ✧ Proses *node root*.
- ✧ *Traverse subtree* kiri secara *preorder*.
- ✧ *Traverse subtree* kanan secara *preorder*.

2. Traversal Inorder (Symmetric Order)

Yaitu dengan melakukan kunjungan ke *subtree* kiri, mencetak isi *node* yang dikunjungi, lalu melakukan kunjungan ke *subtree* kanan. Berikut tahapan algoritma tersebut.

- ✧ Jika *tree* kosong, maka keluar.
- ✧ *Traverse subtree* kiri secara *inorder*.
- ✧ Proses *node root*.
- ✧ *Traverse subtree* kanan secara *inorder*.

3. Traversal Postorder

Yaitu dengan melakukan kunjungan ke *subtree* kiri, lalu ke *subtree* kanan, dan selanjutnya mencetak isi *node* yang dikunjungi. Berikut tahapan algoritma tersebut.

- ✧ Jika *tree* kosong, maka keluar.
- ✧ *Traverse subtree* kiri secara *postorder*.
- ✧ *Traverse subtree* kanan secara *postorder*.
- ✧ Proses *node root*.

Saudara dapat *scan* QR Code berikut untuk dapat memahami lebih dalam mengenai *tree* atau *binary search tree*.

