








Data Structures & Algorithms

# Index	3
 CreatedAt	
 Person	 Ally Hyeseong Kim
 Status	TODO
 Tags	<div>Computer Science</div> <div>Data Structures & Algorithms</div>
 UpdatedAt	

References

면접 시리즈6 - 자료구조, 알고리즘

Java, JPA, Spring을 주로 다루고 공유합니다.

 <https://backtony.github.io/interview/2022-02-14-interview-28/>

References

1. 배열 vs Linked List
2. Stack vs Queue
3. Hash Table
4. Graph, Tree
5. Minimum Spanning Tree (MST)
6. Heap
7. Binary Search Tree
8. AVL
9. B Tree, B+ Tree
10. Red-Black Tree
11. Time Complexity, Space Complexity
12. Time Complexity, 실제 수행 시간
13. Big-O Notation
14. BFS, DFS
15. Kruskal Algorithm

16. Prim Algorithm
17. Dijkstra Algorithm
18. 인접 행렬, 인접 리스트
19. Sorting
20. Divide and Conquer
21. Dynamic Programming
22. Memoization
23. worst 시간 복잡도가 크기만 자주 사용되는 알고리즘
24. Java
25. 손코딩

1. 배열 vs Linked List

- 배열
 - 인덱스를 통한 검색이 용이하다.
 - 연속적이므로 메모리 관리가 편하다.
 - 크기가 고정적이어 컴파일 이후 배열의 크기를 변동할 수 없다.
 - 고정적이기 때문에 요소가 삭제되면 빈 공간을 그대로 남겨두어야 한다.
- 링크드 리스트
 - 크기가 동적이다.
 - 포인터를 통해 다음 데이터를 가르키므로 순차접근을 해야 한다.

2. Stack vs Queue

- 스택
 - 선입후출 구조
 - ex) dfs, 재귀 알고리즘, 함수의 스택프레임 저장
- 큐
 - 선입선출 구조
 - ex) bfs, cpu 스케줄링
- 우선순위 큐
 - 우선순위를 갖고 있는 큐

3. Hash Table

- key-value로 저장되는 자료구조
- 적은 리소스로 많은 데이터를 효율적으로 관리 가능
- 배열 인덱스를 사용하므로 검색, 삽입, 삭제가 빠르다
- 들어온 순서 무시
- 충돌이 많아지면 성능이 낮아지므로 해쉬 함수를 잘 설계해야 한다.

4. Graph, Tree

- 그래프
 - 정점과 간선으로 이루어진 자료구조
- 트리
 - 사이클이 없는 그래프
 - 이진탐색트리 같이 데이터를 빠르게 찾을 때 사용

5. Minimum Spanning Tree (MST)

- 모든 노드를 잇는 신장 트리에서 간선 가중치의 합이 최소값인 트리

6. Heap

- 완전 이진 트리 의 일종으로 최댓값이나 최솟값을 빠르게 찾을 수 있다.
- 중복된 값을 허용
- 우선순위 큐에서 사용
- 삽입
 - 최대 힙: 마지막 노드에 추가하고 부모와 비교해서 부모보다 크면 Swap을 반복한다.
 - 최소 힙: 마지막 노드에 추가하고 부모와 비교해서 부모보다 작으면 Swap을 반복한다.
- 삭제
 - 최대 힙: 루트 노드 데이터를 삭제하고 마지막 노드 데이터를 부모로 가져온 뒤 부모가 자식보다 작으면 자식 중 큰 값과 Swap
 - 최소 힙: 루트 노드 데이터를 삭제하고 마지막 노드 데이터를 부모로 가져온 뒤 부모가 자식보다 크면 자식 중 작은 값과 Swap

7. Binary Search Tree

- 왼쪽 자식은 부모보다 작은값, 오른쪽 자식은 부모보다 큰 값으로 구성
- 중복이 없어야 한다
- 균형 트리인 경우 $O(\log N)$, 편향 트리인 경우 $O(N)$
- 포화이진트리: 리프노드를 제외한 모든 노드가 두 개의 자식을 가지고 있는 트리
- 완전이진트리: 왼쪽부터 차근차근 채워진 이진 트리
- 정이진트리/적정이진트리: 노드들이 자식을 0개 혹은 2개만 갖고 있는 트리

8. AVL

- 자식들의 좌우 높이 차이가 1을 넘지않는 균형 이진 탐색 트리
- 탐색, 삽입, 삭제 $O(\log N)$

9. B Tree, B+ Tree

- B Tree
 - 균형 트리로 하나의 노드에 여러 개의 데이터를 갖는다.
 - 최상위를 root, 중간을 Branch, 마지막을 leaf라고 한다.
 - key들은 항상 오름차순으로 정렬되어 있고 branch 는 key와 data로 구성된다.
 - 한 노드에 최대 M개의 데이터 저장할수 있으면 M차 B-Tree
 - 규칙
 - 노드의 자료수가 N개면 자식 수는 $N+1$
 - 각 노드는 정렬된 상태
 - 루트 노드는 적어도 2개 이상의 자식을 갖어야함
 - 루트 노드를 제외한 모든 노드는 적어도 $M/2$ 개의 자료를 갖고 있어야 함
 - 외부 노드로 가는 경로의 길이는 모두 같음(균형 트리)
 - 입력 자료는 중복 불가
- B+ Tree
 - Branch는 key만 저장되고 key는 항상 오름차순으로 정렬

- key만 담으므로 더 많은 key를 담을 수 있게 되어 트리의 높이가 낮아진다.
- leaf노드는 key와 data를 저장하고 Linked List로 연결되어 있다.
- 풀 스캔 시 B트리의 경우 모든 노드를 확인해야 하지만, B+트리의 경우 리프노드에 연결된 연결리스트로 선형 탐색
- B의 경우 최상 케이스는 루트에서 끝나지만 B+의 경우 실제 데이터가 리프 노드에 있기 때문에 무조건 리프로 내려가야한다.

10. Red-Black Tree

- 균형 이진 탐색 트리
- 조건
 - 루트는 블랙
 - 모든 리프(NIL)노드는 블랙
 - 노드가 레드이면 자식은 반드시 블랙
 - 루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드 수는 모두 같다.
- 위 조건들을 만족하게 되면, 레드-블랙 트리는 가장 중요한 특성을 나타내게 된다.
 - 루트 노드부터 가장 먼 잎노드 경로까지의 거리가, 가장 가까운 잎노드 경로까지의 거리의 두 배 보다 항상 작다.
 - 레드-블랙 트리는 개략적으로 균형이 잡혀 있다. 따라서, 삽입, 삭제, 검색시 최악의 경우에서의 시간복잡도가 트리의 높이(또는 깊이)에 따라 결정되기 때문에 보통의 이진 탐색 트리에 비해 효율적이라고 할 수 있다.

11. Time Complexity, Space Complexity

- 시간 복잡도: 알고리즘을 수행하는데 연산이 몇 번 이루어지는지
- 공간 복잡도: 알고리즘이 필요하는 자원의 양

12. Time Complexity, 실제 수행 시간

- 시간 복잡도는 반복문이 반복하는 횟수로 판단
- 실제 수행에 미치는 요소는 아주 많다

- cpu 클럭 속도, 프로그램 메모리 접근 패턴, 운영체제, 컴파일러 버전 등.

13. Big-O Notation

- 빅오: 최악의 경우
- 빅세타: 평균
- 빅오메가: 최선의 경우

14. BFS, DFS

- DFS
 - 모든 경로를 방문해야 할 경우
 - 스택/재귀함수 사용
 - 시간 복잡도
 - 인접 행렬 : $O(V^2)$
 - 인접 리스트 : $O(V+E)$
- BFS
 - 인접 노드부터 탐색
 - 큐
 - 최소 비용 구하기
 - 시간 복잡도
 - 인접 행렬 : $O(V^2)$
 - 인접 리스트 : $O(V+E)$

15. Kruskal Algorithm

- 간선 위주의 알고리즘
- 정점 개수에 비해 간선이 적은 경우 사용
- 시간 복잡도 : $O(E \log E)$ - 정렬하는데 가장 오래걸림

1. 간선 오름차순 정렬 및 선택
2. 사이클이면 지나침
3. 반복

16. Prim Algorithm

- 최소 비용 신장트리 만드는 알고리즘
- 정점 위주의 알고리즘
- 간선 개수에 비해 정점 개수가 적은 경우 사용
- 시간 복잡도 : $O(E \log V)$

17. Dijkstra Algorithm

- 그래프의 최단 거리를 찾기 위한 알고리즘으로 현재까지의 최단 거리를 계속 갱신하는 방식으로 구성된다.

18. 인접 행렬, 인접 리스트

- 인접 행렬
 - 이차원 배열로 표현
 - 두 정점이 연결되어 있는지 여부는 $O(1)$
- 인접 리스트
 - 리스트로 표현
 - 두 정점이 연결되어 있는지 여부는 $O(V)$

19. Sorting

19.1. Selection Sort

- 앞에서부터 차근차근 비교하는 정렬
- 불안정 정렬
- $O(N^2)$

19.2. Insertion Sort

- 원소가 삽입될 자리를 찾아나가는 정렬
- 안정 정렬
- 최선의 경우 $O(N)$, 최악의 경우 $O(N^2)$

19.3. Bubble Sort

- 인접한 두 원소를 비교하며 정렬하는 방식
- 안정정렬

- 시간 복잡도 $O(N^2)$

19.4. Quick Sort

1. 피벗 선택
2. 오른쪽에서 왼쪽으로 가면서 피벗보다 작은 수를 찾음
3. 왼쪽에서 오른쪽으로 가면서 피벗보다 큰 수를 찾음
4. 각 인덱스에 대한 요소를 교환
5. 반복
6. 두 인덱스가 역전되면 중단하고 피벗과 교환
7. 이제 교환된 피벗 기준으로 왼쪽엔 피벗보다 작은 값, 오른쪽엔 큰 값들만 존재
 - 분할 정복을 통해 정렬을 수행하는 알고리즘으로 왼쪽에는 pivot보다 작거나 같은 것을 모아주고 오른쪽은 pivot보다 크거나 같은 것을 모아주도록 Partitioning을 재귀적으로 진행하여 정렬하는 알고리즘이다.
 - 피벗 값이 최소나 최대값으로 지정되어 파티션이 나뉘지지 않았을 경우 $O(n^2)$ 에 대한 시간 복잡도를 갖고 최선은 $O(n \log n)$ 인 불안정 정렬이다.

19.5. Merge Sort

- 분할정복을 이용한 방식
- 데이터를 분할하여 분할된 여러 개의 부분집합을 하나의 정렬된 집합으로 병합하여 진행
- 안정정렬
- 시간 복잡도
 - 분할 : n 개의 원소를 두 개로 분할 : $O(\log n)$
 - 병합 : 최대 n 번 비교 연산 : $O(n)$
 - 총 $O(n \log n)$

19.6. Heap Sort

- 최대 힙 트리나 최소 힙 트리를 구성해 정렬을 하는 방법
- 총 $O(n \log n)$
- 불안정 정렬

19.7. Binary Search

- 탐색 범위를 두 부분으로 나눠서 진행

- $O(\log n)$

19.8. Stable Sorting Algorithm

- stable: 동일한 Element가 있을 때 정렬 전의 순서와 정렬 후의 순서가 동일함을 보장하는 것

19.9 Sorting Algorithm이 많은 이유

- 정렬 알고리즘마다 기대되는 속도가 다르다.
- Stable 여부에 따라 사용해야할 알고리즘이 다르다.
 - 안정 정렬 : 삽입, 버블, 병합 정렬
 - 불안정 정렬 : 선택, 퀵, 힙 정렬

20. Divide and Conquer

- 큰 문제를 작은 문제로 나눠서 작은 문제를 해결해 합치면서 해를 구하는 것
- 병합 정렬, 퀵 정렬, 이분 탐색

21. Dynamic Programming

- 복잡한 문제를 간단한 여러 개로 나눠서 푸는 것

22. Memoization

- 한 번 계산한 것은 저장해두고 재사용

23. worst 시간 복잡도가 크기만 자주 사용되는 알고리즘

- 퀵 정렬
- 해시

24. Java

- foreach문을 사용할 수 있는 자료구조는 어떤 인터페이스를 구현하는가
 - iterable 인터페이스
- foreach를 사용할 때 다음 데이터를 얻기 위해 내부적으로 호출하는 메서드
 - iterator 메서드를 호출해서 iterator를 가져온다. 루프를 돌 때는 hasNext로 값이 있는지 보고 Next로 가져온다.
- iterable과 iterator의 차이

- Iterable 인터페이스의 역할은 iterator() 메소드를 하위 클래스에서 무조건 구현을 하게 만들기 위한 역할이다. iterator는 컬렉션을 순회하는 기능을 한다.
- iterator를 상속받으면 구현해야 하는 메서드
 - hasNext, next
- Map을 사용해야 하는 경우
 - JSON자료구조를 Map을 이용하여 표현할 수 있다.
- Hashing
 - 키(Key) 값을 해시 함수(Hash Function)라는 수식에 대입시켜 계산한 후 나온 결과를 주소로 사용하여 바로 값(Value)에 접근하게 할 수 하는 방법이다.
- Hash 값의 중복을 피하는 방법
 - 배열의 크기를 크게하거나 hash 함수의 중복값이 나오지 않도록 2차 해싱 등 해시 함수를 잘 짜야한다.
- Hashing의 장점
 - 해싱 알고리즘만 잘 설계되어있다면 데이터를 가져오는데 $O(1)$ 로 가져올 수 있다.
- Queue와 Stack 예시
 - Stack은 함수 스택, 재귀 알고리즘 Queue는 BFS 알고리즘, CPU 스케줄링
- Queue와 Stack의 내부 자료구조
 - Queue는 ArrayList보다 데이터의 추가/삭제가 쉬운 LinkedList로 구현하는 것이 적합하다.
 - Stack은 vector를 상속받아서 구현된다.
- 동적 계획법(메모이제이션)과 분할 정복의 차이
 - 주어진 문제를 더 작은 문제들로 나눈 뒤, 작은 문제들의 답을 구하고 원래 문제에 대한 답을 구한다는 점에서 동적 계획법과 분할 정복은 같다. 동적 계획법은 어떤 부분 문제는 두 개 이상의 문제를 푸는 데 사용될 수 있기 때문에 답을 한 번만 계산하고 그 결과를 여러 번 사용하도록 하는 반면에, 분할 정복은 중복해서 사용하지 않고 재귀 함수로 구현된다.

25. 손코딩

25.1. 재귀를 이용한 피보나치 수열

```
public class Test {
    public static void main(String[] args) {
        Test test = new Test();
        System.out.println(test.fibo(5));
    }

    private int fibo(int n) {
        if (n < 2)
            return n;
        return fibo(n - 1) + fibo(n - 2);
    }
}
```

25.2. Memoization을 이용한 피보나치 수열

```
public class Test {
    public static void main(String[] args) {
        Test test = new Test();
        int n = 5;
        System.out.println(test.solution(n));
    }

    private int solution(int n) {
        int[] memo = new int[n + 1];
        return fibo(n, memo);
    }

    private int fibo(int n, int[] memo) {
        if (memo[n] != 0)
            return memo[n];
        if (n < 2)
            return n;
        return memo[n] = fibo(n - 1, memo) + fibo(n - 2, memo);
    }
}
```

25.3. Dynamic Programming을 이용한 피보나치 수열

```
public class Test {
    public static void main(String[] args) {
        Test test = new Test();
        int n = 5;
        System.out.println(test.solution(n));
    }

    private int solution(int n) {
        int[] dp = new int[n + 1];
        dp[0] = 0;
        dp[1] = 1;
```

```

        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
}

```

25.4. 재귀 팩토리얼

```

int fact(int n) {
    if(n==1)
        return 1;
    return n * fact(n-1);
}

```

25.5. Selection Sort

```

public class Test {
    public static void main(String[] args) {
        Test test = new Test();
        int[] arr = {1, 5, 6, 2, 3, 7, 8, 4, 9, 10};
        test.selectSort(arr);
        for (int i : arr) {
            System.out.println(i);
        }
    }

    private void selectSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            int min = i;
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[min] > arr[j])
                    min = j;
            }
            int tmp = arr[i];
            arr[i] = arr[min];
            arr[min] = tmp;
        }
    }
}

```

25.6. Bubble Sort

```

public class Test {
    public static void main(String[] args) {
        Test test = new Test();
        int[] arr = {1, 5, 6, 2, 3, 7, 8, 4, 9, 10};
    }
}

```

```

        test.bubbleSort(arr);
        for (int i : arr) {
            System.out.println(i);
        }
    }

    private void bubbleSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] > arr[j]){
                    int tmp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = tmp;
                }
            }
        }
    }
}

```

25.7. Insertion Sort

```

public class Test {
    public static void main(String[] args) {
        Test test = new Test();
        int[] arr = {1, 5, 6, 2, 3, 7, 8, 4, 9, 10};
        test.insertSort(arr);
        for (int i : arr) {
            System.out.println(i);
        }
    }

    private void insertSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            int tmp = arr[i];
            for (int j = i - 1; j >= 0; j--) {
                if (tmp < arr[j]){
                    arr[j+1] = arr[j];
                }
                else{
                    arr[j+1] = tmp;
                    break;
                }
            }
        }
    }
}

```

25.8. Quick Sort

```

class Solution {
    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] arr = {1, 5, 6, 2, 3, 7, 8, 4, 9, 10};
        solution.quickSort(arr, 0, arr.length - 1);
        for (int i : arr) {
            System.out.println(i);
        }
    }

    private void quickSort(int[] arr, int left, int right) {
        if (left < right){
            int l = partition(arr, left, right);
            quickSort(arr, left, l-1);
            quickSort(arr, l+1, right);
        }
    }

    private int partition(int[] arr, int left, int right) {
        int pivot = right;
        int l = left;
        for (int r = left; r < right; r++){
            if (arr[r] < arr[pivot]){
                swap(arr, l, r);
                l++;
            }
        }
        swap(arr, l, pivot);
        return l;
    }

    private void swap(int[] arr, int l, int r) {
        int temp = arr[r];
        arr[r] = arr[l];
        arr[l] = temp;
    }
}

```

25.9. Merge Sort

```

public class Test {
    public static void main(String[] args) {
        Test test = new Test();
        int[] arr = {1, 5, 6, 2, 3, 7, 8, 4, 9, 10};
        int[] temp = new int[arr.length];
        test.mergeSort(arr, 0, arr.length - 1, temp);
        for (int i : arr) {
            System.out.println(i);
        }
    }

    private void mergeSort(int[] arr, int left, int right, int[] temp) {
        if (left >= right)

```

```

        return;
    int mid = (left + right) / 2;
    mergeSort(arr, left, mid, temp);
    mergeSort(arr, mid + 1, right, temp);

    int p1 = left;
    int p2 = mid + 1;
    int idx = left;

    while (p1 <= mid && p2 <= right) {
        if (arr[p1] < arr[p2])
            temp[idx++] = arr[p1++];
        else
            temp[idx++] = arr[p2++];
    }
    while (p1 <= mid)
        temp[idx++] = arr[p1++];
    while (p2 <= right)
        temp[idx++] = arr[p2++];

    for (int i = left; i <= right; i++)
        arr[i] = temp[i];
}
}

```

25.10. 최대 공약수, 최소 공배수

```

private int gcd(int a, int b) {
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

private int gcd(int a, int b) {
    return BigInteger.valueOf(a).gcd(BigInteger.valueOf(b)).intValue();
}

private int lcm(int a, int b) {
    return a * b / gcd(a, b);
}

```