자료구조

[자료구조와 알고리즘 2](#_Toc73658643)

[순환 2](#_Toc73658644)

[스택(stack) 2](#_Toc73658645)

[큐(Queue) 3](#_Toc73658646)

[Array vs Linked List 3](#_Toc73658647)

[트리(Tree) 4](#_Toc73658648)

[히프(heap) 5](#_Toc73658649)

[그래프(Graph) 6](#_Toc73658650)

[그래프 탐색 7](#_Toc73658651)

[신장 트리 7](#_Toc73658652)

[최단 경로 8](#_Toc73658653)

[정렬 9](#_Toc73658654)

# 자료구조와 알고리즘

자료구조: 사람들이 사물을 정리하여 저장하는 것과 마찬가지로 프로그램에서도 자료들을 정리하여 보관하는 여러가지 구조.

알고리즘: 어떠한 문제를 해결하기 위해 정해진 일련의 절차나 방법을 공식화 한 형태로 표현

* 종류 : 자연어, 흐름도, 의사 코드, 프로그래밍 언어

프로그램: 자료구조 + 알고리즘.

추상 자료형 (ADT): Abstract data type. 추상적, 수학적으로 자료형을 정의한 것.

* 장점: 캡슐화가 가능, 코드의 재사용성 및 가독성을 증가시킨다, 사용의 유연함을 제공

빅오표기법 : 연산의 횟수를 대략적(점근적)으로 표기한 것 O(n)으로 표기한다.

O(1) < O(logn) < O(n) < O(nlogn) < O() < O() < O(n!)

# 순환

순환 (recursion) : 어떤 알고리즘이나 함수가 자기 자신을 호출하여 문제를 해결하는 기법

* 복귀 주소가 시스템 스택에 저장.
* 반드시 순환 호출에는 순환 호출을 멈추는 문장이 포함되어야 한다.
* 알고리즘이 순환적으로 되어 있는 경우 유리.
  + 팩토리얼 함수, 피보나치 수열(효율 x), 이항계수, 이진 트리 알고리즘, 이진 탐색, 하노이 탑
* 장점: 이해하기 쉽다, 쉽게 프로그램 할 수 있다.
* 단점: 수행 시간과 기억공간의 사용에 있어서 비효율적인 경우가 있다. (ex 피보나치)

# 스택(stack)

* 후입선출(LIFO)구조. 선형 자료구조.
* 나중에 들어간 자료가 먼저 나온다
* 입출력은 맨 위에서만 일어난다.
* 복귀할 주소를 기억하는데 사용
  + 되돌리기(undo), 괄호 검사, 후위 표기 수식, 미로 문제

# 큐(Queue)

* 선입선출(FIFO)구조. 선형 자료구조
* 먼저 들어온 자료가 먼저 나간다.
* 스택과 달리 삽입, 삭제가 양 끝에서 발생한다.
* 선형 큐 – 선형으로 이루어진 큐. 배열의 끝에 도달하면 끝.
* 원형 큐 – 원형으로 되어있는 큐. 배열을 순환시켜 공간이 있는 한 삽입, 삭제가 가능
* 덱(deque) – 큐의 전단과 후단에서 모두 삽입, 삭제가 가능
  + 스케줄링, 버퍼, 은행 서비스

# Array vs Linked List

Array

* 가장 기본적인 자료구조. 논리적 저장 순서화 물리적 저장순서가 일치.
* index로 해당 원소에 접근 가능. => index값을 알면 O(1)으로 접근가능.
* random access가 가능하다.
* 삽입, 삭제과정에서 원소에 접근한 뒤, 작업을 추가적으로 해줘야 하기 때문에 시간이 더 걸린다. O(n)의 시간이 소요.

Linked List

* array의 단점을 해결하기 위한 자료구조.
* +) 삽입, 삭제가 용이하고 연속된 메모리 공간이 필요 없으며 크기 제한이 없다.
* -) 구현이 어렵고 오류가 발생하기 쉽다.
* 각 원소는 자신 다음의 원소만 기억한다.
* 삽입, 삭제 과정에서 바꿔치기만 하면 끝난다. O(1)의 시간이 소요
* 논리적 저장 순서와 물리적 저장 순서가 일치하지 않기 때문에 search과정에서 O(n)의 시간이 소요.

# 트리(Tree)

* 계층적인 자료구조, 비선형 자료구조
* 부모-자식 관계의 노드들로 이루어진다.
* 용어
  + 노드 (node): 트리의 구성 요소.
  + 루트 노드 (root node): 트리 구조에서 최상위 노드
  + 서브 트리 (subtree): 트리에서 루트를 제외한 나머지 트리
  + 간선 (edge): 트리를 구성하기 위해 노드와 노드를 연결하는 선
  + 차수 (degree): 자식 노드의 개수를 의미
  + 레벨 (level): 각 층의 번호를 의미
  + 높이 (height): 트리가 가지고 있는 최대 레벨을 의미
  + 단말 노드(terminal node): 자식이 없는 노드
* 이진 트리 (Binary Tree)
  + 트리 중 가장 많이 쓰이는 트리.
  + 모든 노드가 2개의 서브 트리를 가지고 있다.
  + 포화 이진 트리, 완전 이진 트리, 기타 이진 트리로 나뉜다.
    - 포화 이진 트리: 각 레벨에 노드 꽉 차 있는 이진 트리.
    - 완전 이진 트리: 마지막 레벨은 왼쪽부터 채워지는 이진 트리.
    - 그 외의 트리는 표현할 수는 있으나, 기억공간 낭비가 심해진다.
  + 3가지의 순회 방법이 있다
    - 전위 순회: 자손보다 루트노드를 먼저 방문 (VLR)
    - 중위 순회: 왼쪽 자손, 루트, 오른쪽 자손 순으로 방문 (LVR)
    - 후위 순회: 루트노드보다 자손을 먼저 방문한다. (LRV)
* 스레드 이진 트리
  + 순회 방법에 따른 방문 순서를 유지하는 포인터.
  + 이진 트리의 노드에 NULL 링크들을 효율적으로 사용하기위한 트리
  + 순환호출 사용한다. 트리 높이 커질수록, 노드가 많을수록 비효율.
  + 트리ex) 계층적인 조직 표현, 인공지능의 결정트리, 컴퓨터 디렉토리
* 이진 탐색 트리 (BST, Binary Search Tree)
  + 이진 트리 기반의 탐색을 위한 자료구조.
  + 노드가 n개 있을 때, 평균 시간 복잡도는 O()이다.
    - 정의1 : 모든 원소의 키는 유일한 키를 가진다.
    - 정의2 : 왼쪽 서브 트리 키들은 루트의 키보다 작다.
    - 정의3 : 오른쪽 서브 트리의 키들은 루트의 키보다 크다
    - 정의4 : 왼쪽과 오른쪽 서브 트리도 이진 탐색 트리이다.

# 히프(heap)

* 히프는 완전 이진 트리 기반의 자료구조이다. 여러 개의 값들 중에서 가장 큰 값이나 가장 작은 값을 빠르게 찾아내도록 만들어진 자료구조이다.
* 부모 노드의 키 값이 자식 노드의 키 값보다 항상 큰 이진 트리. (max heap)
* 히프 트리는 중복된 값을 허용한다. (이진 탐색 트리에서는 허용하지 않음)
* 목표는 가장 큰 값을 찾아내기.
* 완전 이진 트리이기 때문에 배열을 사용하여 효율적으로 관리 가능하다 (random access가 가능하다)
* 종류
  + - 최대 히프(max heap): key(부모 노드) >= key(자식 노드)
    - 최소 히프(min heap): key(부모 노드) <= key(자식 노드)
* 삽입, 삭제의 시간 복잡도는 O(). 정렬의 시간 복잡도는 O().
  + - ex) machine scheduling, 허프만 코드, 네트워크 트래픽 제어

# 그래프(Graph)

* 그래프는 객체 사이의 연결 관계를 표현할 수 있는 자료구조.
* 종류
  + 무방향 그래프(undirected graph) : 간선을 통해 양 방향으로 갈 수 있음
  + 방향 그래프(directed graph) : 간선을 통해 한쪽 방향으로만 갈 수 있음
  + 가중치 그래프(weight graph) : 네트워크라고도 불리며 간선에 가중치도 나타냄.
  + 부분 그래프(sub graph) : 그래프의 일부 정점 및 간선으로 이루어진 그래프
* 용어
  + 인접 정점(adjacent vertex) : 간선에 의해 직접 연결된 정점.
  + 간선(edge): 정점들 간의 관계를 의미. 링크(link)라고도 불림
  + 차수(degree) : 정저에 인접한 정점의 수
  + 진입 차수 / 진출 차수 : 외부에서 오는 차수 / 외부로 나가는 차수
  + 경로(path)
    - 단순 경로(simple path) : 경로 중 반복되는 간선이 없는 경로
    - 사이클(cycle) : 단순 경로 중 시작과 종료 정점이 동일한 경로
  + 연결 정도
    - 연결 그래프(connected graph): 모든 정점쌍에 대하여 항상 경로가 존재하는 그래프
    - 완전 그래프(complete graph): 모든 정점이 서로 연결 되어있는 그래프
    - 트리(tree): 사이클을 가지지 않는 연결 그래프.
* 표현 방법
  + 인접 행렬(adjacency matrix) : 2차원 배열을 사용하여 그래프를 표현
    - 밀집 그래프(dense graph)의 표현에 적합
    - 간선의 존재 여부는 O(1), 차수는 O(n), 모든 간선의 수는 O(로 알 수 있음
  + 인접 리스트(adjacency list) : 연결 리스트를 사용하는 그래프를 표현
    - 희소 그래프(sparse graph)의 표현에 적합
    - 정점의 수가 n, 간선의 수가 e개 일 때, 간선의 존재, 차수는 O(n), 모든 간선의 수는 O(n+e)의 시간이 걸림.
  + ex) 지하철 지도, 도로, 미로, 선수과목

# 그래프 탐색

* 가장 기본적인 연산으로, 차례대로 모든 정점들을 한 번씩 방문하는 것. 두 가지의 알고리즘을 기반으로 한다.
* 깊이 우선 탐색(DFS : depth first search)
  + 한 방향으로 가다가 더 이상 갈 구 없게 되면 다시 가장 가까운 갈림길로 돌아가서 다른 방향으로 탐색을 진행하는 방법.
  + 방문했던 정점을 스택으로 표현 가능
  + 정점의 수 : n, 간선의 수 : e 일때, 인접 리스트 – O(n+e), 인접 행렬 – O(
* 너비 우선 탐색(BFS : breath first search)
  + 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법.
  + 방문했던 정점을 큐로 표현 가능
  + 정점의 수 : n, 간선의 수 : e 일때, 인접 리스트 – O(n+e), 인접 행렬 – O(
    - ex) 도로망에서 특정 도시에서 다른 도시로 갈 수 있는지 여부

# 신장 트리

* + 그래프내의 모든 정점을 포함하나, 사이클을 포함하지 않는 트리.
  + n개의 정점을 가지는 신장 트리는 n-1개의 간선을 가짐.
    - ex) 통신망, 도로망, 유통망
  + 최소 비용 신장 트리(MST)
    - 네트워크에 있는 모든 정점들을 가장 적은 수의 간선과 비용으로 연결.
    - ex) 도로건설, 전기회로, 통신, 배관
  + Kruskal’s MST
    - 탐욕적인 방법(greedy method)
    - 각 단계에서 최선을 답을 선택과정을 반복하는 방법.
    - union-find 알고리즘
      * 두 집합들의 합집합으로 만듦.
      * 원소가 어떤 집합에 속하는지 알아냄.
      * Kruskal 뿐만 아니라 널리 사용된다.
      * 간선을 e개 가질 때, 시간 복잡도는 O().
  + Prim’s MST
    - 시작 정점에서부터 신장 트리 집합을 단계적으로 확장하는 방식
    - 신장 트리 집합에 최저 간선으로 연결된 정점을 추가함.
    - 시간 복잡도는 O(
    - vs Kruskal
      * 희소 그래프 - O()인 Kruskal이 유리
      * 밀집 그래프 - O(인 Prim이 유리

# 최단 경로

* + 네트워크에서 정점 사이를 연결하는 경로 중에서 합이 최소가 되는 경로
  + ex) 비용, 거리, 시간
  + Dijksta 알고리즘
    - 하나의 시작 정점에서 다른 정점까지의 최단 경로 계산
    - 시간 복잡도는 O(
    - n번 반복한다면 전체 복잡도는 O(
  + Floyd 알고리즘
    - 모든 정점에서 다른 모든 정점까지의 최단 경로 계산
    - 2차원 배열 A를 이용하여 3중 반복을 하는 루프로 구성
    - 시간 복잡도는 O(이지만 반복 구문이 간결하여 Dijksta보다 빠르다.
  + 위상 정렬(topological sort)
    - 방향 그래프 정점들의 선행 순서를 위배하지 않으면서 모든 점을 나열
    - 선수 과목은 과목들의 선행 관계 표현함

# 정렬

* 크기 순으로 오름차순이나 내림차순으로 나열하는 것
* 단순하지만 비효율적인 방법
  + 삽입 정렬, 선택 정렬, 버블 정렬 등
* 복잡하지만 효율적인 방법
  + 퀵 정렬, 히프 정렬, 합병 정렬, 기수 정렬 등
* 내부 정렬(internal sorting) - 모든 데이터가 주기억장치에 저장되어진 상태에서 정렬
* 외부 정렬(external sorting) - 외부기억장치에 대부분 데이터가 있는 정렬.
* 안정성 – 동일한 키 값을 갖는 레코드들의 상대적인 위치가 정렬 후에도 바뀌지 않음.
* 선택 정렬 (selection sort)
  + 비교횟수 : O(
  + 이동횟수 : 3(n-1)
  + 시간 복잡도 O(
  + 안정성 X
* 삽입 정렬(insertion sort)
  + 정렬되어 있는 부분에 새로운 레코드를 옳은 위치에 삽입
  + 시간 복잡도 : O(
  + 많은 이동 필요 (O(
  + 안정성 O
* 버블 정렬 (bubble sort)
  + 인접한 2개의 레코드를 비교하여 순서대로 되어 있지 않으면 서로 교환
  + 비교 횟수(최상, 평균, 최악 모두 같음) : n(n-1)/2 = O(
  + 이동횟수
    - 역순으로 정렬된 경우(최악의 경우): 3 \* 비교횟수
    - 이미 정렬된 경우(최선의 경우) : 0
    - 평균의 경우 : O(
    - 이동 과다 : 이동연산은 비교연산보다 더 많은 시간이 소요됨
  + 셀 정렬 (Shell sort)
    - 전체 리스트를 일정 간격(gap)의 부분 리스트로 나눔
    - 삽입 정렬의 많은 이동의 단점을 보완하는 정렬.
    - 장점 : 불 연속적인 부분 리스트에서 원거리 자료 이동으로 보다 적은 위치교환으로 제자리 찾을 가능성 증대. 부분 리스트가 점진적으로 정렬된 상태가 되므로 삽입 정렬 속도 증가
    - 시간 복잡도 : 최악 : O(, 평균 : O(
  + 합병 정렬 (merge sort)
    - 리스트를 두 개의 균등한 크기로 분할하고 분할된 부분 리스트를 정렬
    - 정렬된 두 개의 부분 리스트를 합하여 전체 리스트를 정렬함
    - 비교: 크기 n인 리스트를 log(n)개의 패스로 분배, 각 패스에서 n번의 비교.
    - 이동 : 2n\*log(n)번
    - 레코드의 크기가 클 경우, 큰 시간적 낭비.
    - 시간 복잡도 : O()
  + 퀵 정렬 (quick sort)
    - 분할 정복법 사용
    - 평균적으로 가장 빠른 정렬 방법.
    - 리스트를 2개의 부분리스트로 비균등 분할하고, 각각의 부분리스트를 다시 퀵정렬함(재귀호출)
  + 기수 정렬(Radix sort)
    - 레코드를 비교하지 않고 정렬 수행
    - O(dn)의 시간복잡도를 가짐 (d< 10)
    - 버켓은 큐로 구현.
    - 버켓의 개수는 키의 표현 방법과 밀접한 관계.
    - 실수, 한글, 한자로 이루어진 키는 정렬 못함

|  |  |  |  |
| --- | --- | --- | --- |
| 알고리즘 | 최선 | 평균 | 최악 |
| 삽입 정렬 | O( | O( | O( |
| 선택 정렬 | O( | O( | O( |
| 버블 정렬 | O( | O( | O( |
| 셀 정렬 | O( | O( | O( |
| 퀵 정렬 | O() | O() | O( |
| 히프 정렬 | O() | O() | O() |
| 합병 정렬 | O() | O() | O() |
| 기수 정렬 | O( | O( | O( |