1. 스택

* 계산기 문제
* DFS(스택 or 재귀) 방식.
* 백트래킹은 DFS에서 발전한 형태.
* 미로문제(백트래킹으로 해결) 마찬가지로 (스택 or 재귀) 방식 존재.
* Linked List Stack(https://www.youtube.com/watch?v=LOQf1SNPTF0)

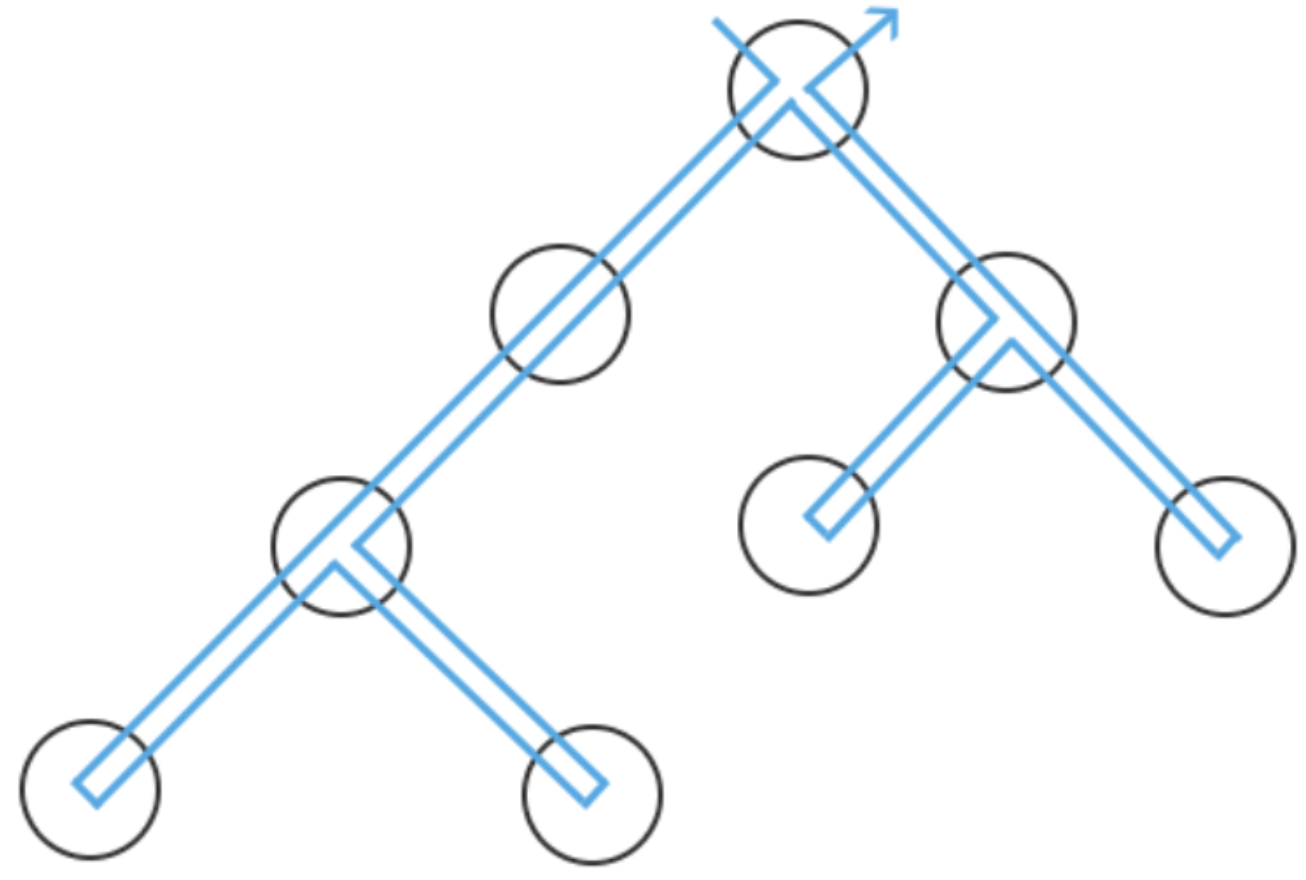
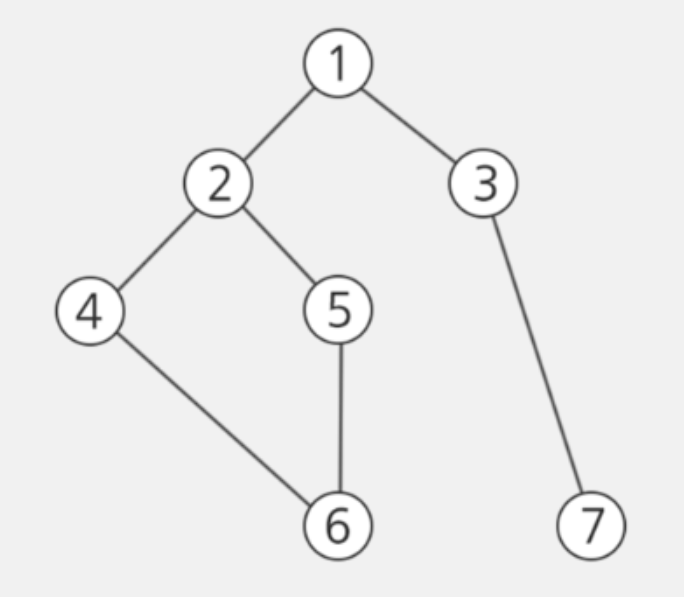
스택의 구현

비선형 구조인 그래프는 그래프로 표현된 모든 자료를 빠짐없이 검색하는 것이 중요.

DFS(깊이 우선 탐색), BFS(너비 우선 탐색) 방식이 있다.

DFS – 트리나 그래프 경로 탐색 방법. 스택(or 재귀!)를 이용하여 탐색.

- 스택을 이용하여 되돌아갈 정점을 표시.



1 2 4 6 5 3 7

-----

Visited[], Stack[] 초기화

DFS(v)

Visited[v] 🡨 True; V 방문;

Do{

If(v의 인접 정점 중 방문 안 한 W 찾기){

Push(v); //마치 함수의 현재 소멸되는 변수를 저장하듯이

While(w){

Visited[w] 🡨 True; W 방문;

Push(w);

V🡨w;

V의 인접 정점 중 방문 안 한 w 찾기.

}

}

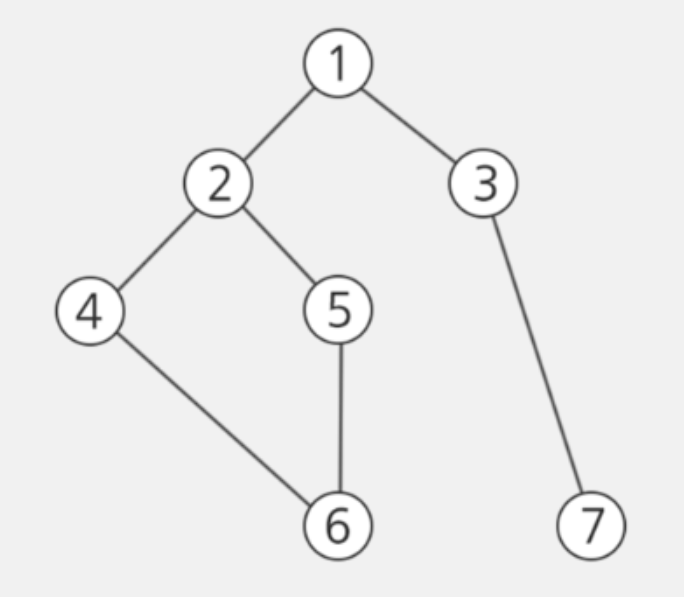
V 🡨 pop(stack);

}while(v)

End DFS()

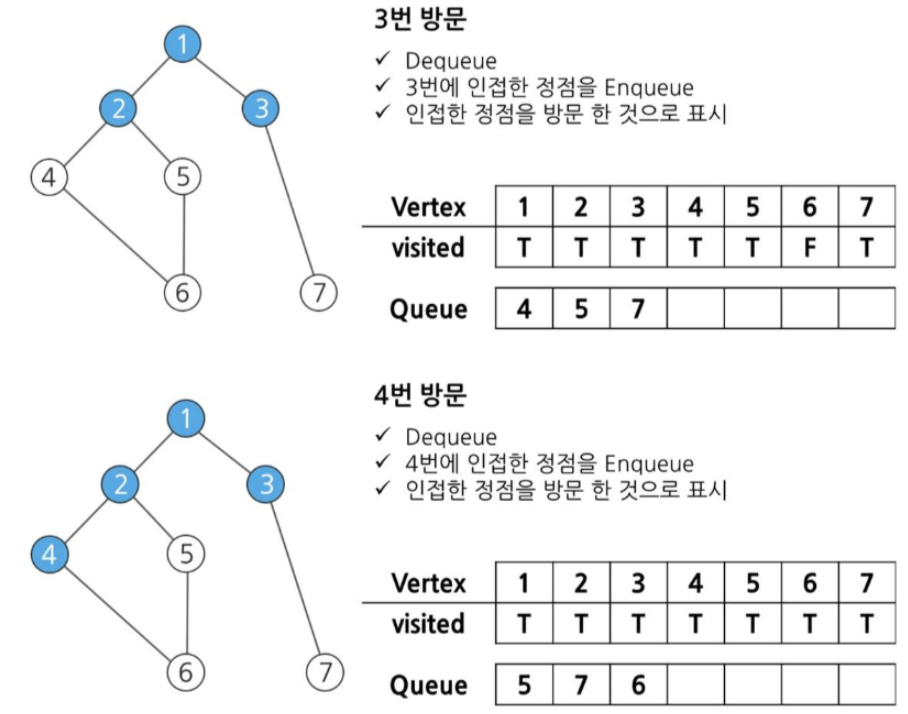
BFS – 트리나 그래프 경로 탐색 방법. 큐를 이용하여 탐색.

- 큐!를 이용하여 앞으로 방문할 정점을 표시



1 2 3 4 5 7 6

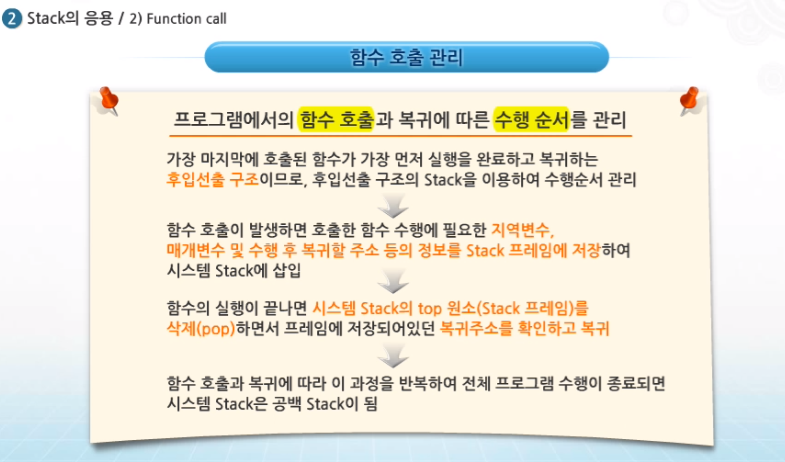
BFS 탐색 예시)

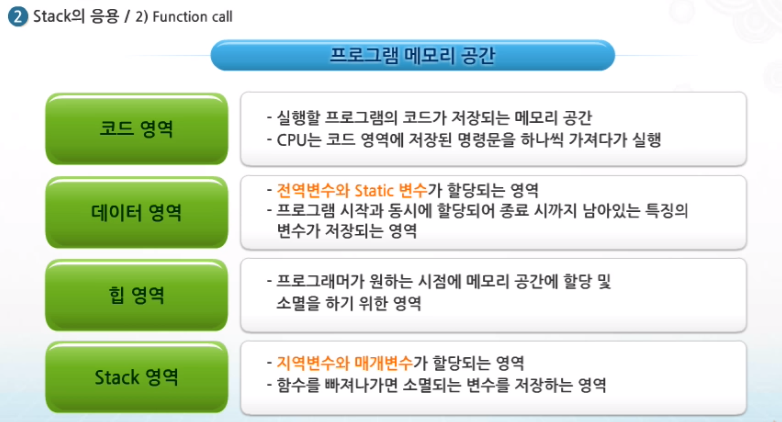


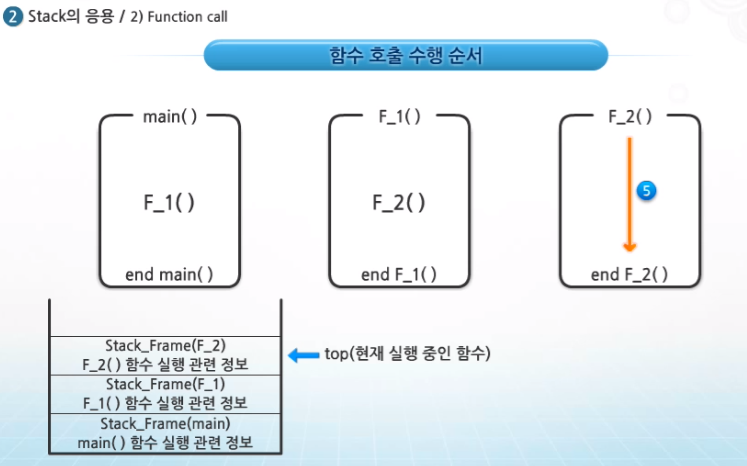
3을 방문하면서 deQueue( ), 7을 enQueue(7).

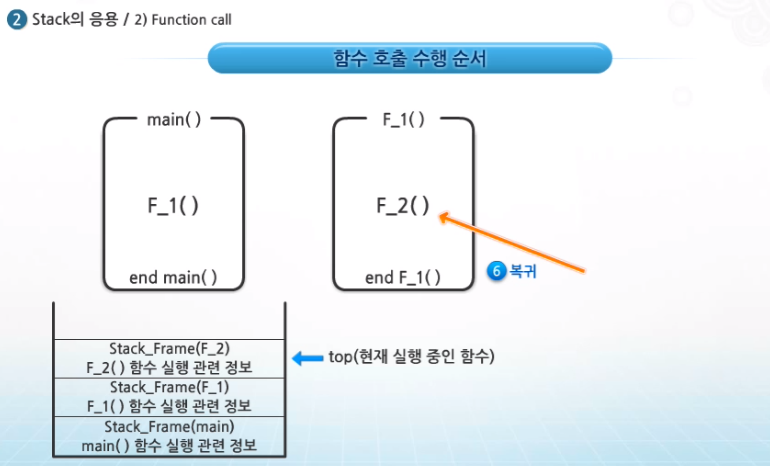
4를 방문하면서 deQueue( ), 6을 enQueue(6)

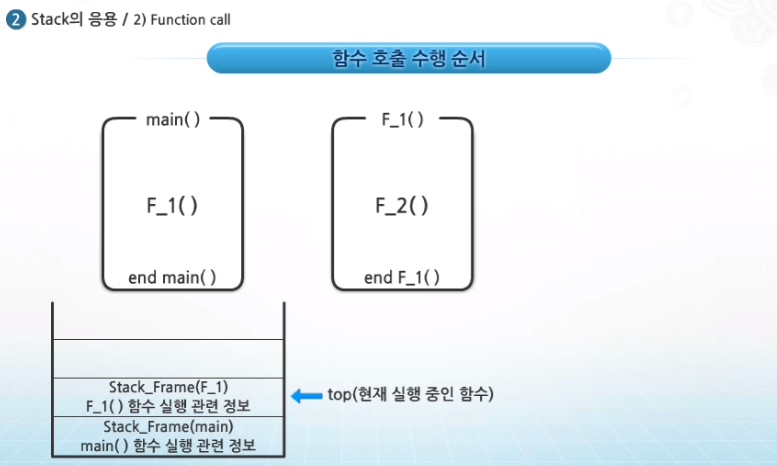
5를 방문하면서 deQueue( ), 이때 6을 다시 enQueue(6) 하지 않는다.

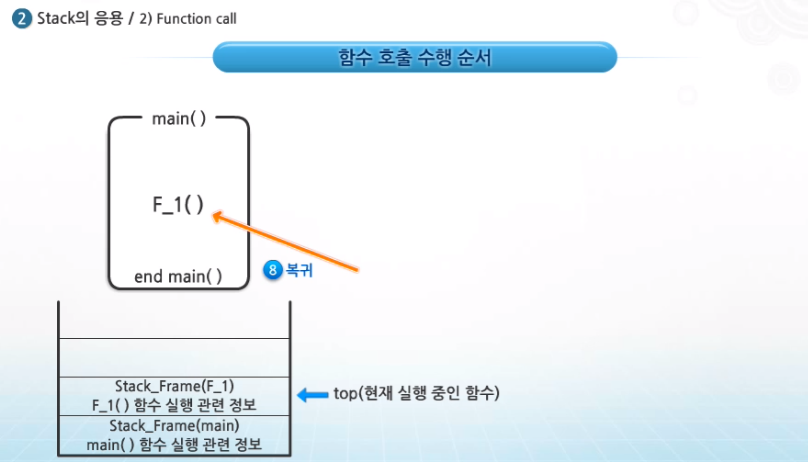


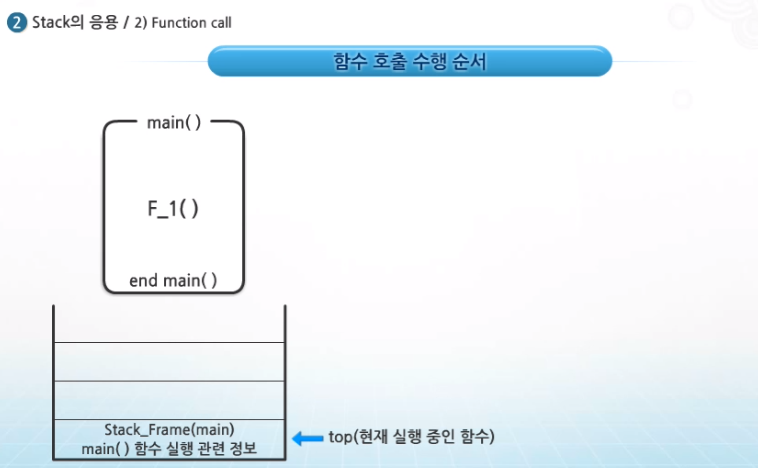


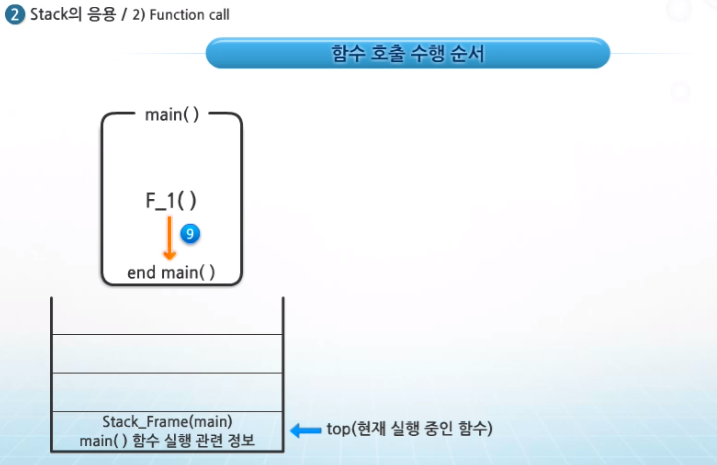












Memorization : 기억하기, 암기하기

Memoization : 기억되어야 할 것. 메모리에 넣기.



#define TRUE 1

#define FALSE 0

int g\_map[100][100];

char g\_resultmap[100][100];

PAIR g\_stack[10000];//최대 스택은 1만개 각각 x,y 값을 저장하는 구조체

int g\_top = 0;

typedef struct{

int x, y;

} PAIR;

int main(){

int T;

int start\_x, start\_y, now\_x, now\_y;

int W, H, i, j;

char str[101];

scanf("%d", &T);

while(T--){

is\_findpath = FALSE;

g\_top = 0;

scanf("%d %d", &W, &H);

for(i=0; i<H ;i++){

scanf("%s", str);

for(j=0;j<W;j++){

if(str[j] == '2'){

start\_x = j;//시작 x

start\_y = i;//시작 y

}

g\_map[i][j] = str[j] = '0';

}

}

//시작점 push

g\_stack[g\_top].x = start\_x;

g\_stack[g\_top].y = start\_y;

g\_top++;

while(g\_top){

now\_x = g\_stack[g\_top - 1].x;

now\_y = g\_stack[g\_top - 1].y;

if(g\_map[now\_y][now\_x] == 3){

if\_findpath = TRUE;

break;

}

g\_map[now\_y][now\_x] = 1;

if(now\_x > 0 && g\_map[now\_y][now\_x-1] != 1){

g\_stack[g\_top].x = now\_x -1;

g\_stack[g\_top].y = now\_y;

g\_top++;

}

else if(now\_x < W - 1 && g\_map[now\_y][now\_x+1] != 1){

g\_stack[g\_top].x = now\_x +1;

g\_stack[g\_top].y = now\_y;

g\_top++;

}

else if(now\_y> 0 && g\_map[now\_y-1][now\_x] != 1){

g\_stack[g\_top].x = now\_x;

g\_stack[g\_top].y = now\_y - 1;

g\_top++;

}

else if(now\_y < H - 1 && g\_map[now\_y+1][now\_x] != 1){

g\_stack[g\_top].x = now\_x;

g\_stack[g\_top].y = now\_y + 1;

g\_top++;

}

else{//갈 곳이 없는 경우

g\_top--;

}

}

if(is\_findpath == FALSE)

printf("Can not find path.\n");

else{

for(i=0;i<H;i++)

for(j=0;j<W;j++)

g\_resultmap[i][j] = 'X'

}

while(g\_top){

g\_top--;

now\_x = g\_stack[g\_top].x;

now\_y = g\_stack[g\_top].y;

g\_resultmap[now\_y][now\_x] = '.';

}

for(i=0;i<H;i++){

for(j=0;j<W;j++){

printf("%c", g\_resultmap[i][j]);

}

printf("\n");

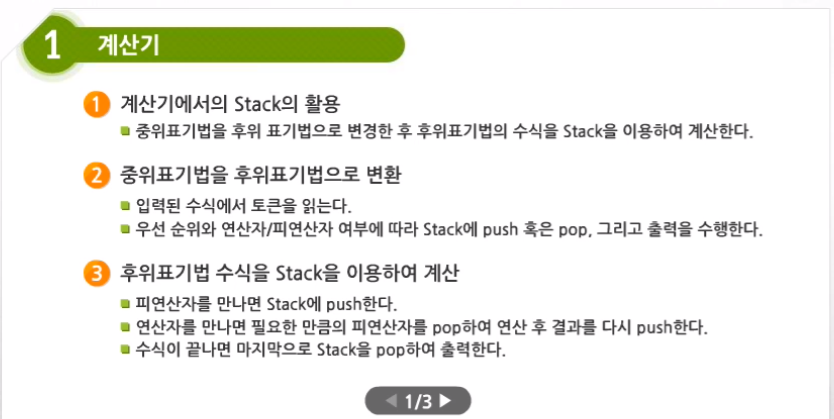
}

}

return 0;

}

* 복잡하다. Dfs는 재귀로 푼다.



1. 큐

큐의 구현. 당연히, 원형 큐(Circler Queue)

링크드 리스트 큐(Linked List Queue)

우선순위 큐(Priority Queue)

* 시뮬레이션 시스템, 네트워크 트래픽 제어, 운영체제의 테스크 스케줄링 등에 적용
* 1) 배열의 기반으로 구현하는 방법(X)
* 2) 연결리스트를 기반으로 구현하는 방법(X)
* 3) 힙(Heap)을 이용하는 방법(O)

링크드 리스트 덱(Deque)

1. 트리

자식과 부모의 관계로 이루어진 계층적 구조.

3-1 힙

최대값, 최소값을 빠르게 찾아낼 수 있는 트리형 자료구조

완전 이진 트리 형식(노드번호 n번까지 빈자리가 없는 이진트리)

힙의 삽입: 부모 노드와 대소관계 확인

힙의 삭제: 힙의 가장 마지막 원소를 삭제된 원소 위치에 넣어준 후. 대소관계 확인

힙은 Dijkstra 알고리즘, Prim 알고리즘으로 이어진다.

1. 문자열

1) Brute Force

2) 카프-라빈 알고리즘

3) KMP 알고리즘

4) 보이어-무어 알고리즘

5. 그래프

- 인접행렬로 모델링

- 인접 리스트로 모델링

<http://sarah950716.tistory.com/12>

<http://lasthere.tistory.com/15>

6. 해싱

해싱은 임의의 길이의 데이터(Key)를 고정된 길이의 데이터(Hash Value)로 변환해 작은 크기의 해시 테이블로 대응(Mapping)시켜 식별하는 기법.

해시 테이블은 M개의 버킷으로 이루어져 있으며, 이 글에서 다루는 해시 값은 해당 키가 저장될 버킷번호(해시 테이블의 인덱스)를 나타낸다.

키에서 해시 값을 추출하는 일련의 과정을 해시함수(Hash Function)이라고 한다. 해시 함수는 같은 키에 대해서는 동일한 해시 값을, 다른 키에 대해서는 다른 해시 값을 추출한다. 하지만 일반적으로 해싱에서 해시 값의 범위는 키의 범위보다 작기 때문에 비둘기집 원리에 의해 서로 다른 두 키가 같은 해시 값을 가질 수 있다 이런 경우 Collision 이라고 한다.

충돌을 제어하는 방법도 다양.

체이닝

- 충돌한 키들을 보존하기 위해 각 버킷을 리스트 형태로 구현. 최초 버킷은 모두 원소가 0개인 리스트의 헤더이며, 해당 버킷에 데이터가 추가될 때 마다 노드를 추가한다. 이때 어떤 키가 해시테이블에 존재하는지 검사하기 위해서는 해당 키의 해시 값에 해당하는 버킷이 가진 노드를 모두 순회해야 한다.

좋은 해시함수란 해시값을 추출하는 연산이 빠르되, 출동이 적고 해시 테이블의 영역을 고르게 사용할 수 있어야 한다.

해싱의 핵심은 값의 식별. 그렇기에 원소의 중복을 허용하지 않는 Set이나 Key:Value쌍에서 중복된 Key가 존재하면 안되는 Map과 같은 자료구조를 구현하는데 사용되기도 한다.

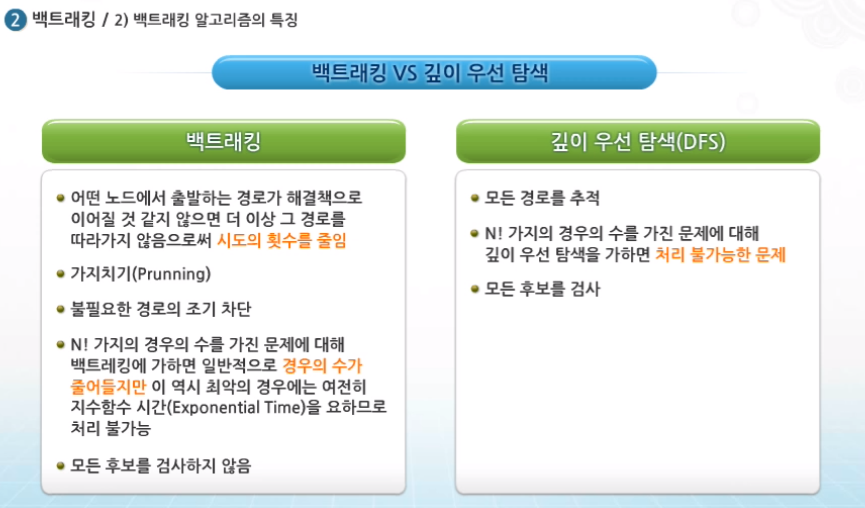
백트래킹

– 최적화(Optimization)문제,

– 결정(Decision)문제 해결 기법

: 문제의 조건을 만족하는 해가 존재하는지 여부를 ‘YES’, ‘NO’로 답하는 문제.

: 미로 찾기, n-Queen 문제, Map coloring, 부분 집합의 합(Subset Sum)문제 등.

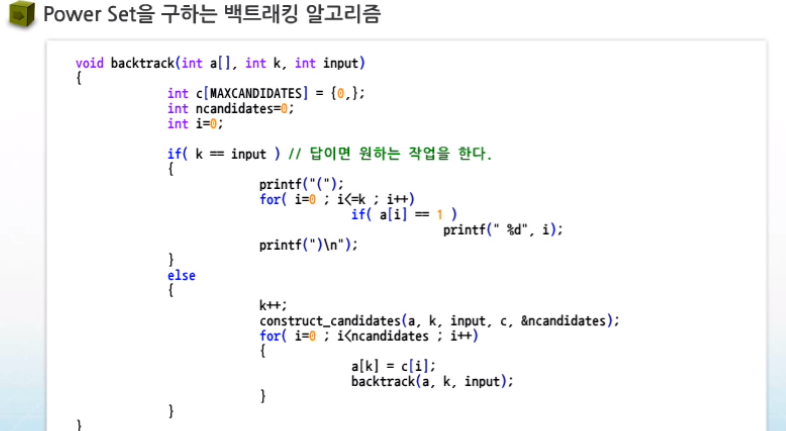


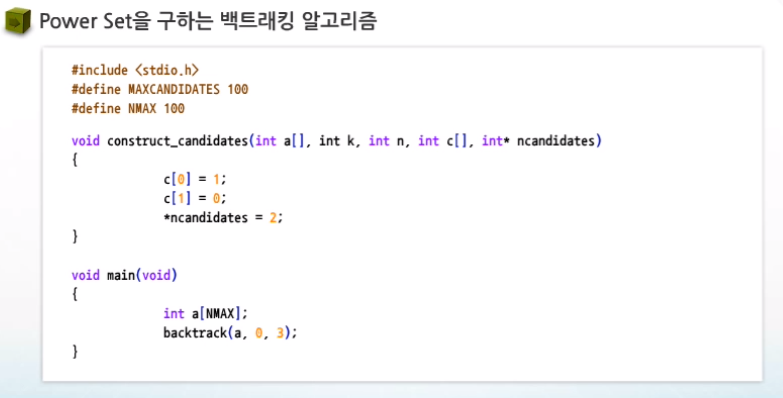
알고리즘 절차

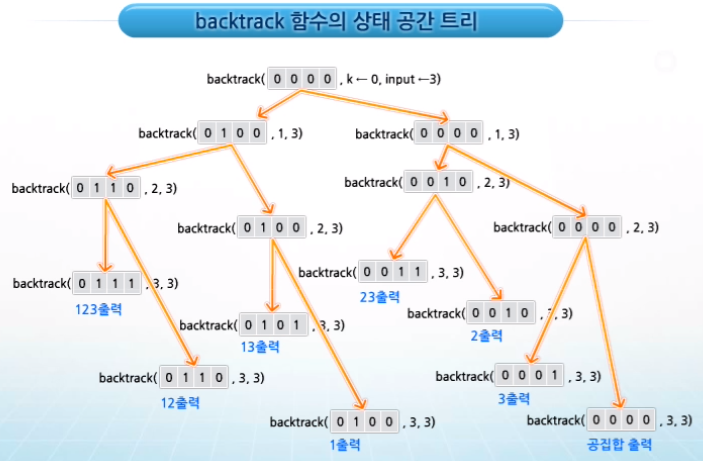
1. 상태 공간 Tree의 깊이 우선 검색을 실시

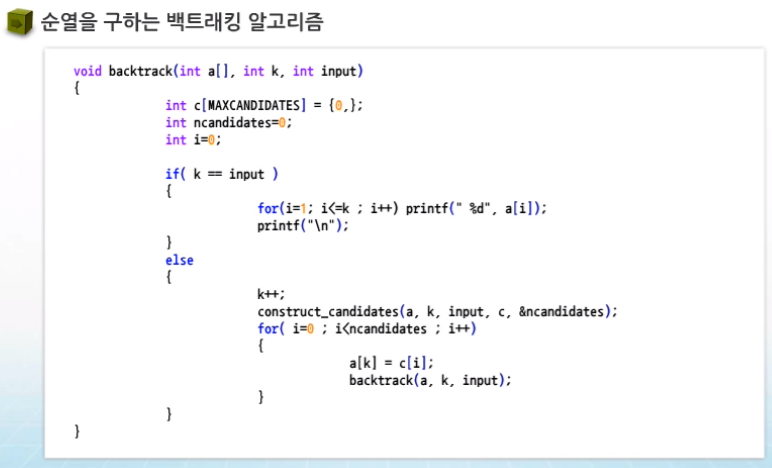
2. 각 노드가 유망한지를 점검

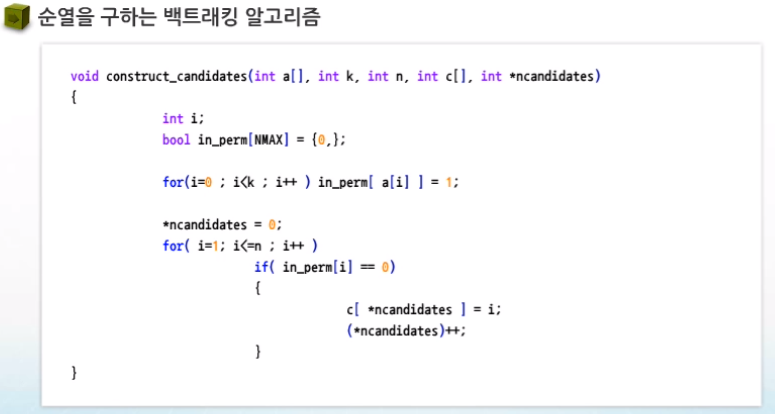
3. 만일 그 노드가 유망하지 않으면, 그 노드의 부모 노드로 돌아가서 검색을 계속.

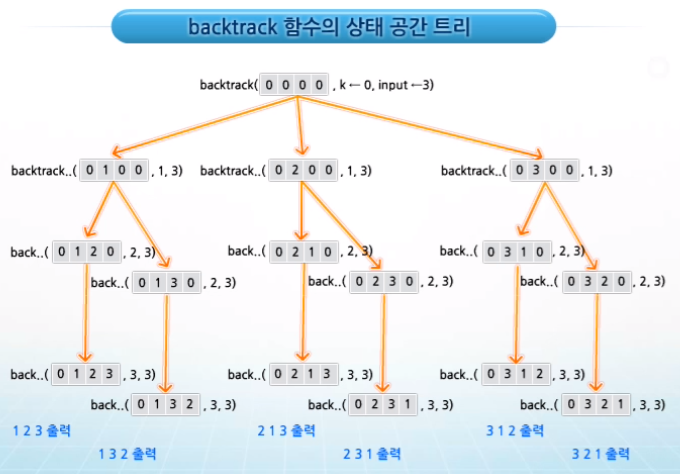












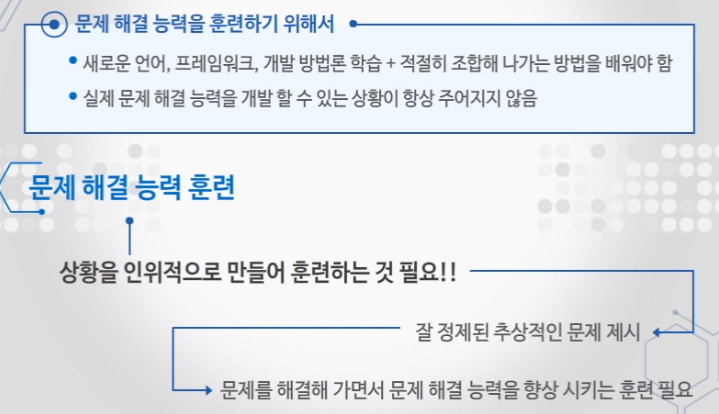
1. Dijkstra 알고리즘

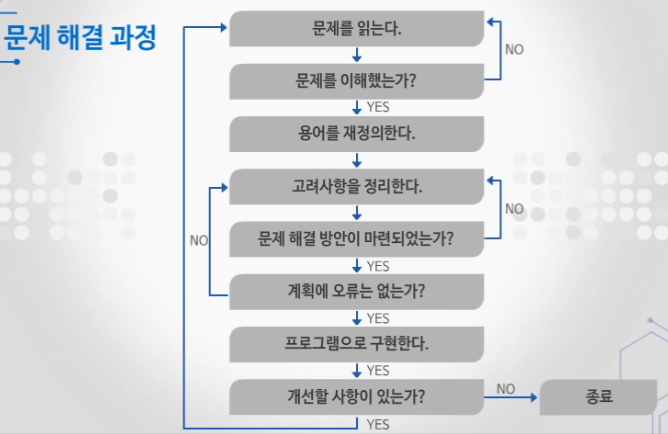
최소 스패닝 트리(Minimum Spanning Tree) 결과를 내는 Prim’s, Kruskal's 알고리즘과는 다르다.

2. LRU Cache

- 가장 최근에 덜 사용된 것을 캐시에서 내 보낸다.

알고리즘





체계적인 접근

1) 비슷한 문제를 풀어본 적이 있던가?

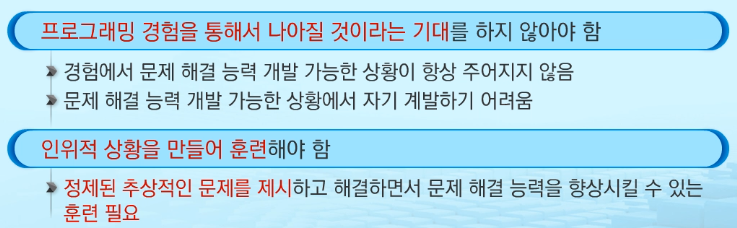
2) 단순한 방법에서 시작할 수 있을까?

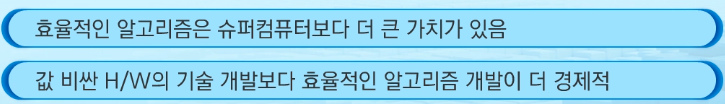
3) 문제를 단순화할 수 있을까? – 수식, 그림 이용

4) 문제를 분해할 수 있을까?

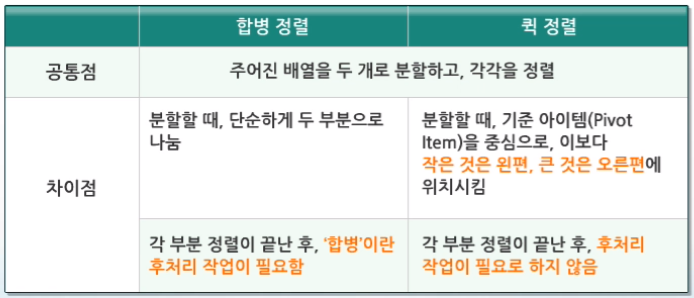
5) 뒤에서부터 생각해서 문제를 풀 수 있을까?

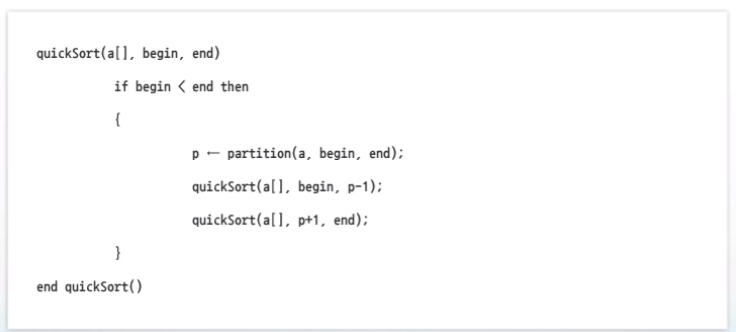
6) 특정 형태의 답 만을 고려할 수 있을까.

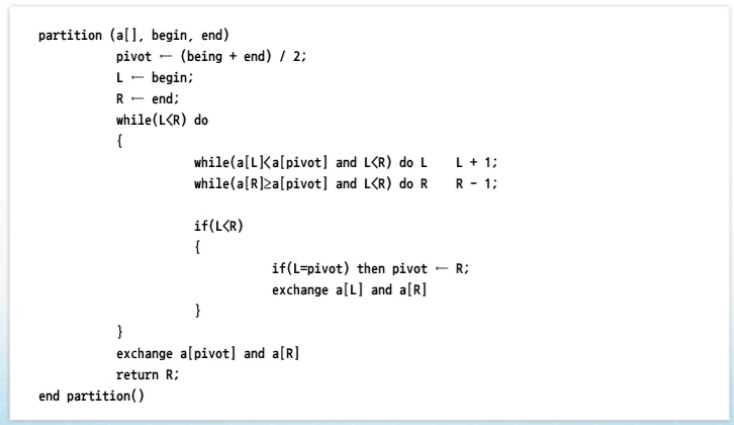




정렬





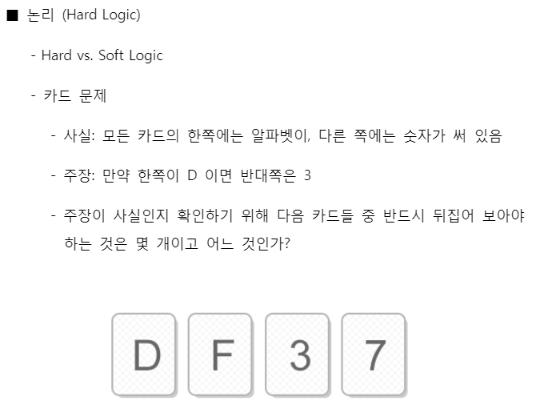


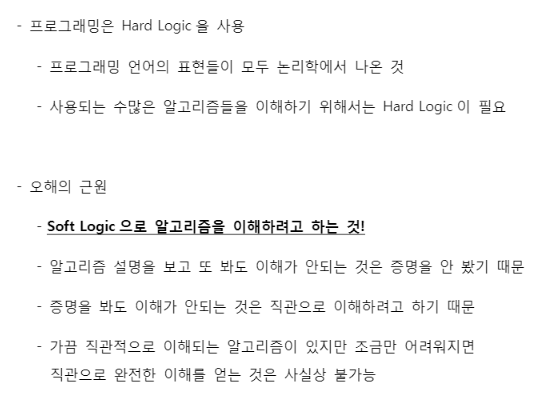


프로그래밍의 어려운 점 두가지

- 프로그래밍 언어 문법과 라이브러리 사용

- 논리(Hard Logic)





사용되는 수많은 알고리즘들을 이해하기 위해서는 Hard Logic 필요

- 직관으로 이해하려 하면 안된다.

- 증명을 봐야 한다.

■ 버전관리는 기본



■ 개발을 하면서 구현에 대한 문서를 동시에 작성한다.

- github/Wiki 에다가 구현/설계 문서를 정리한다.

(팀원들끼리 설계에 대한 내용을 빠르고 쉽게 공유할 수 있다)

(내용이 바뀔 경우 문서로 반영 가능하다)

■ github/Issue로 작업을 관리한다. 혹은 레드마인.

- 팀원에게 일을 줄 수 있다(MileStone과 담당자 지정 가능)

- 기능 분담을 할 수 있다.

**매개변수(parameter)와 전달인자(argument)**

매개변수는 함수의 정의부분에 나열되어 있는 변수들을 의미하며, 전달인자는 함수를 호출할 때 전달되는 실제 값을 의미한다. 이같은 의미를 명확히 하기 위해 매개변수는 **변수(variable)**로, 전달인자는 **값(value)**으로 보는 것이 일반적이다.

각각의 매개변수는 함수의 정의 부분에 포함되어 있는 고유한 특성이다.

대부분의 경우 매개변수는 call by value의 형태로 동작하게 되며, 이 경우 서브루틴 내부에서 매개변수는 전달 인자를 복사한 독립적인 지역변수의 형태로 동작하게 된다. 반면 call by reference의 경우에는 서브루틴 내부에서의 동작이 이를 호출한 부분에서의 전달인자에 까지 영향을 주게 된다.