



07. Graph

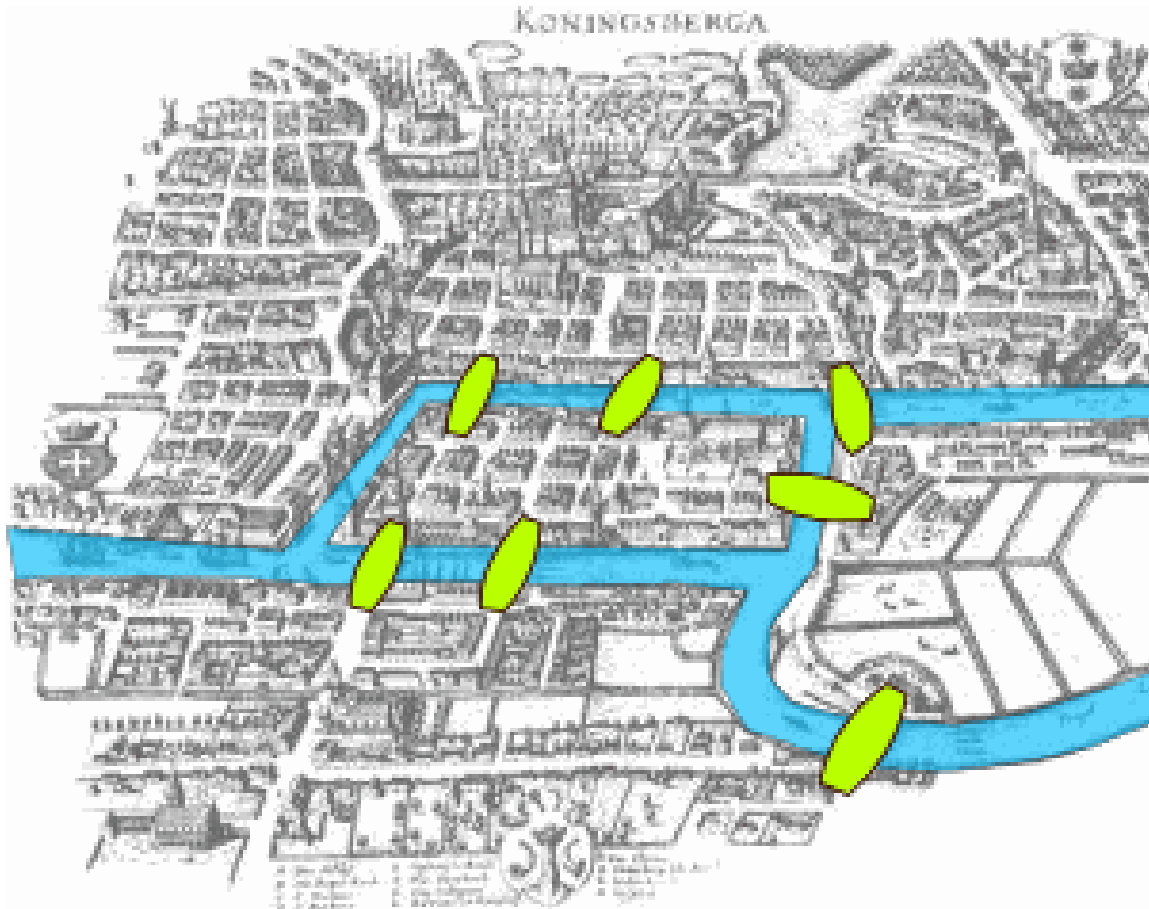
Div. 3 알고리즘 스터디 / 임지환



그래프

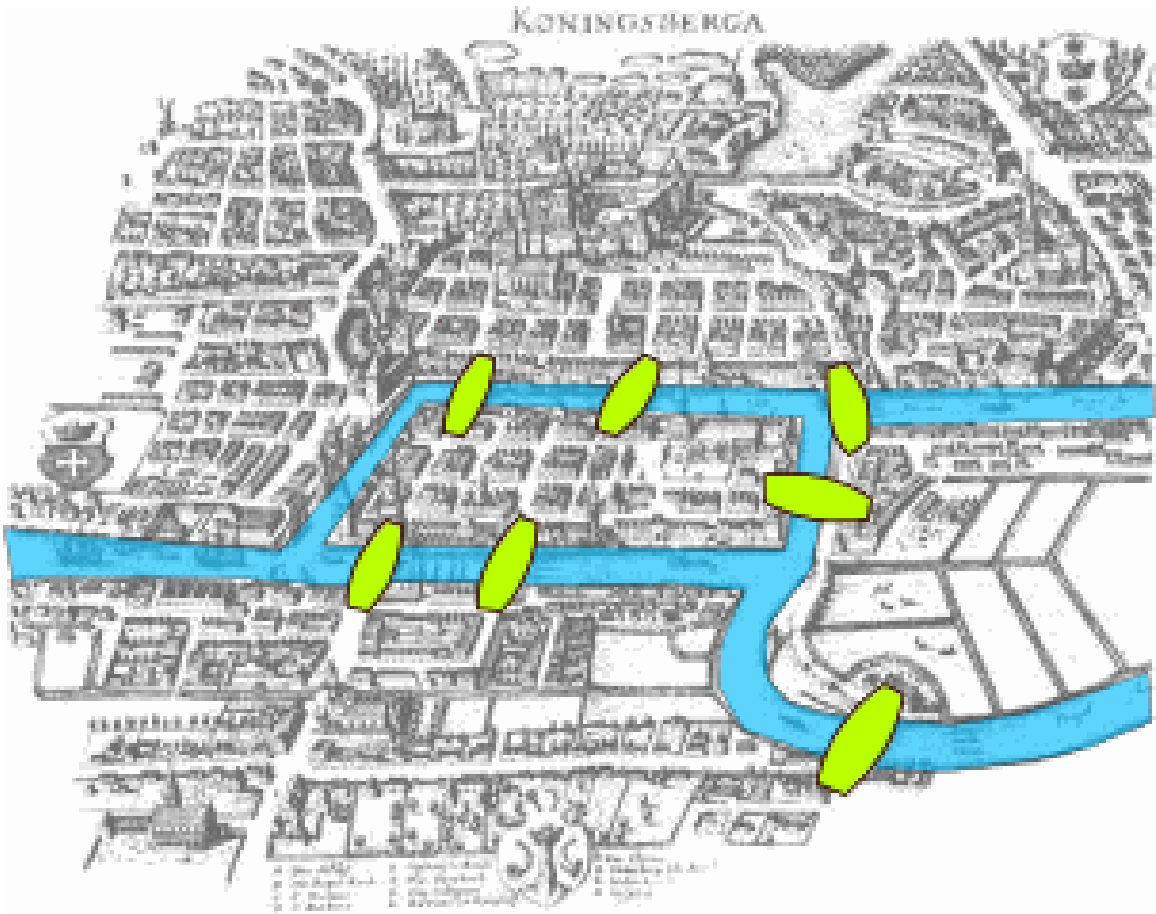


그래프





그래프





선형 자료구조의 한계

- 앞서 본 예시를 선형 자료구조로 관계를 표현할 수 있는가?



선형 자료구조의 한계

- 앞서 본 예시를 선형 자료구조로 관계를 표현할 수 있는가?
 - 힘들걸요.



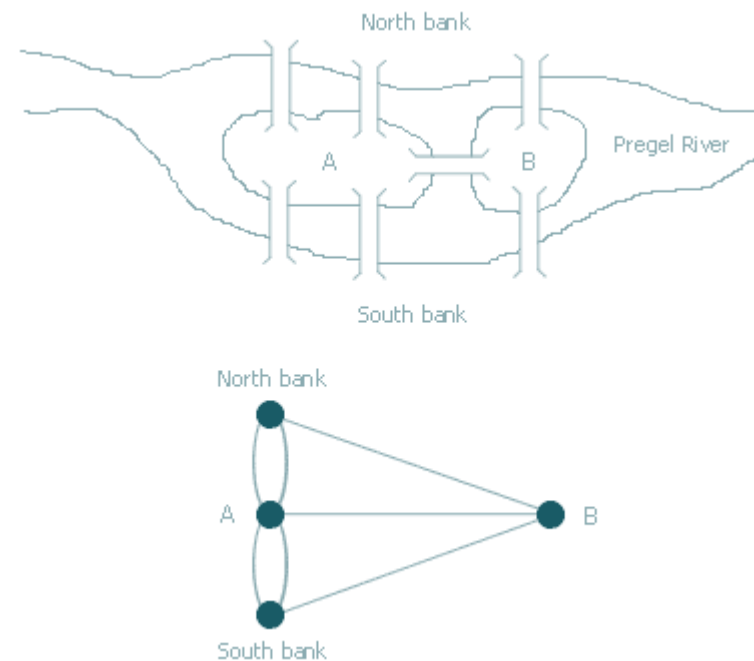
선형 자료구조의 한계

- 앞서 본 예시를 선형 자료구조로 관계를 표현할 수 있는가?
 - 힘들걸요.
- 퀴니히스베르크 다리 문제를 모델링해봅시다.



선형 자료구조의 한계

- 앞서 본 예시를 선형 자료구조로 관계를 표현할 수 있는가?
 - 힘들걸요.
- 쾨니히스베르크 다리 문제를 모델링해봅시다.



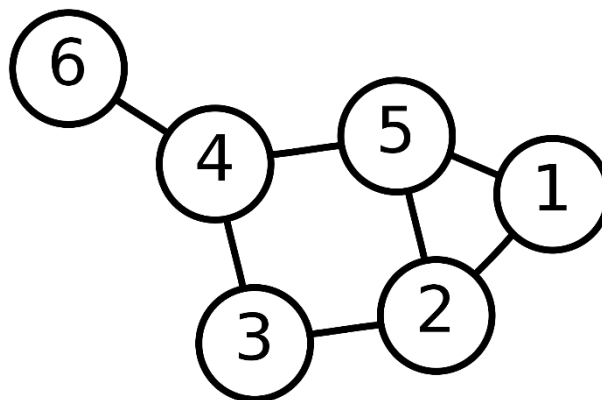


그래프의 정의

G라는 그래프에 대하여 순서쌍 $G = (V, E)$ 로 표현

$V(G) :=$ 정점(vertex)들의 집합

$E(G) :=$ 간선(edge)들의 집합



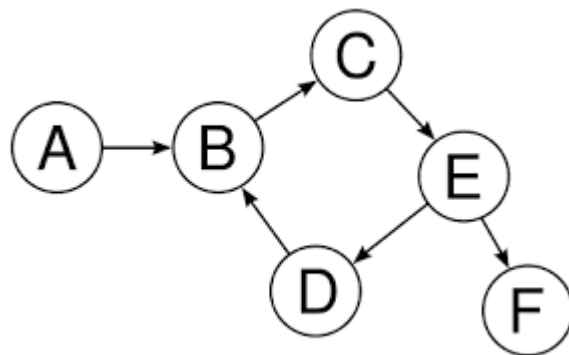
$$V(G) = \{1, 2, 3, 4, 5, 6\}$$

$$E(G) = \{(1, 2), (1, 5), (2, 3), (2, 5), (3, 4), (4, 5), (4, 6)\}$$

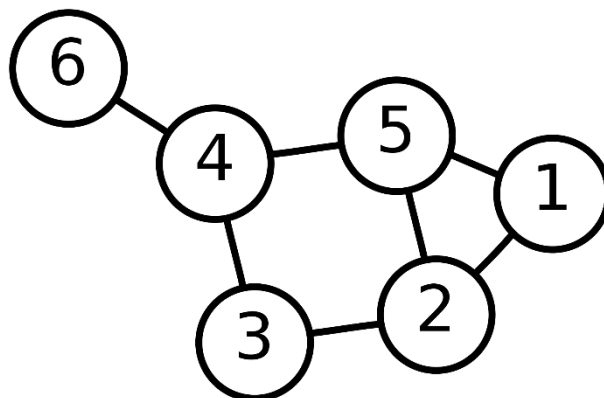


방향그래프와 무방향 그래프

- Directed Graph



- Undirected Graph





Adjacent & incident

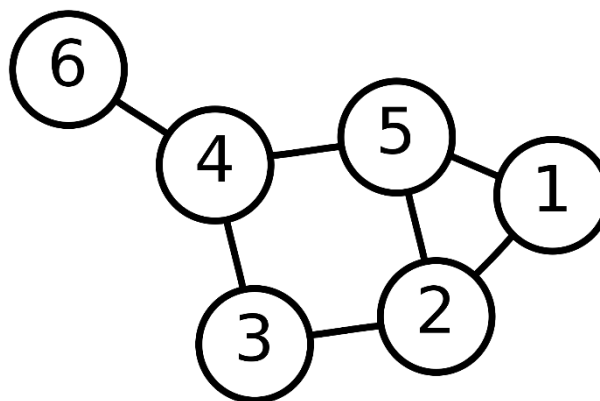
- Adjacent(인접)
 - Undirected Graph에서의 간선 (v_0, v_1) 에 대하여,
 - 두 정점 v_0 과 v_1 는 **adjacent**하다고 한다.
 - Directed Graph에서의 간선 $\langle v_0, v_1 \rangle$ 에 대하여,
 - v_0 is **adjacent to** v_1 이고
 - v_1 is **adjacent from** v_0 이라 한다.
- Incident(부속)
 - Undirected Graph에서의 간선 (v_0, v_1) 은 두 정점 v_0 과 v_1 에 **incident**하다고 한다.
 - Directed Graph에서의 간선 $\langle v_0, v_1 \rangle$ 은 두 정점 v_0 과 v_1 에 **incident**하다고 한다.



경로

- Path

끝과 끝이 서로 연결된 간선들을 순서대로 나열한 것



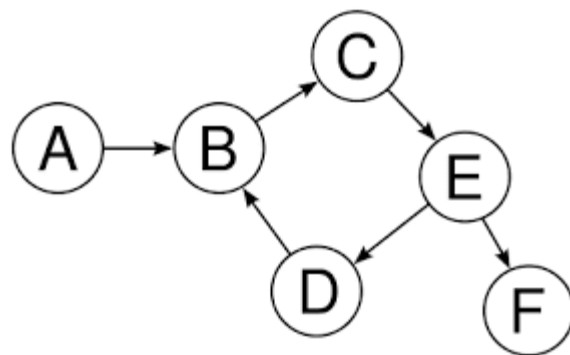
ex) (6,4) (4,3) (3,2) (2,5) (5,1)



경로

- Path

얘도 경로인가???



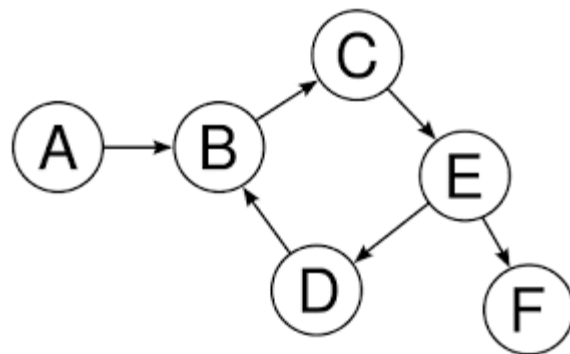
ex) (A,B) (B,D) (D,E) (E,F)



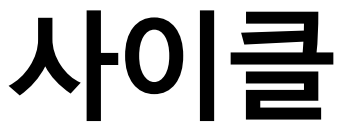
경로

- Path

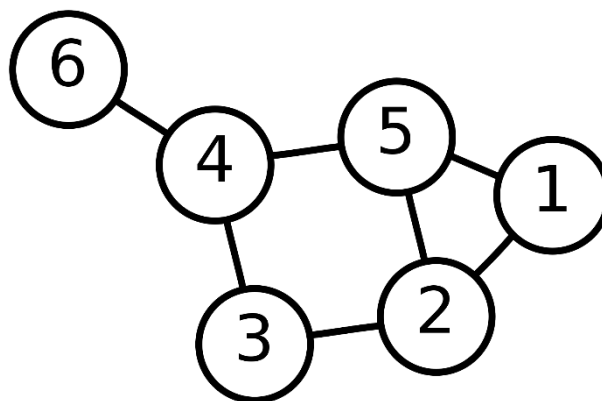
얘도 경로인가??? 아니오.



ex) (A,B) (B,D) (D,E) (E,F)



시작한 점에서 끝나는 경로



ex) (4,3) (3,2) (2,5) (5,4)



생각해봅시다.

문제

민오는 1번부터 N번까지 총 N개의 문제로 되어 있는 문제집을 풀려고 한다. 문제는 난이도 순서로 출제되어 있다. 즉 1번 문제가 가장 쉬운 문제이고 N번 문제가 가장 어려운 문제가 된다.

어떤 문제부터 풀까 고민하면서 문제를 훑어보던 민오는, 몇몇 문제들 사이에는 '먼저 푸는 것이 좋은 문제'가 있다는 것을 알게 되었다. 예를 들어 1번 문제를 풀고 나면 4번 문제가 쉽게 풀린다거나 하는 식이다. 민오는 다음의 세 가지 조건에 따라 문제를 풀 순서를 정하기로 하였다.

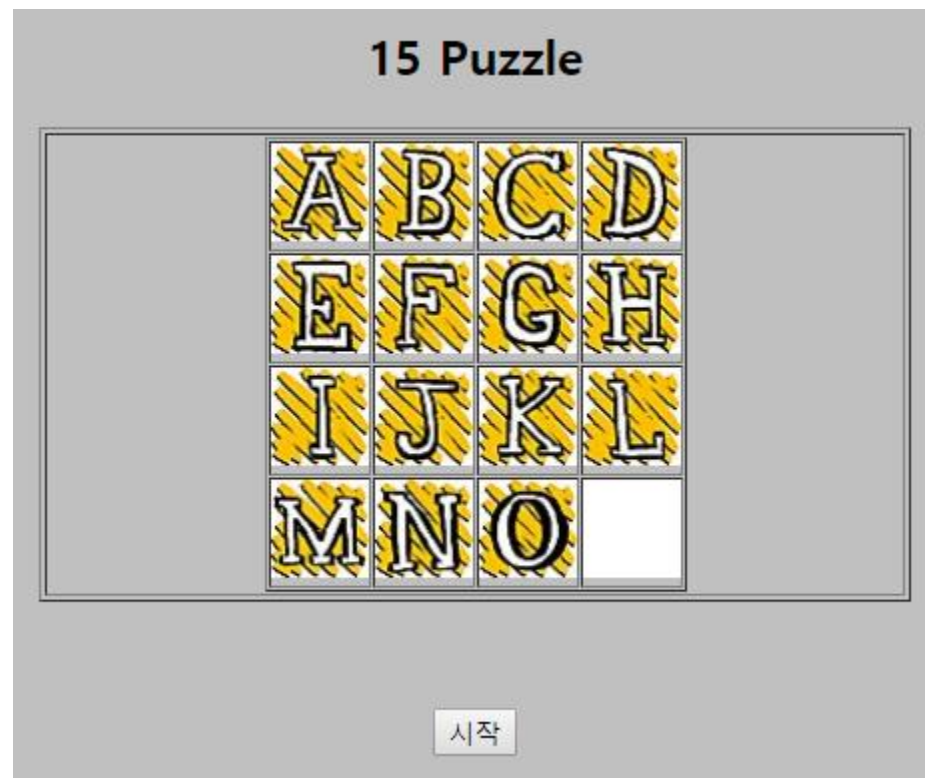
1. N개의 문제는 모두 풀어야 한다.
2. 먼저 푸는 것이 좋은 문제가 있는 문제는, 먼저 푸는 것이 좋은 문제를 반드시 먼저 풀어야 한다.
3. 가능하면 쉬운 문제부터 풀어야 한다.

예를 들어서 네 개의 문제가 있다고 하자. 4번 문제는 2번 문제보다 먼저 푸는 것이 좋고, 3번 문제는 1번 문제보다 먼저 푸는 것이 좋다고 하자. 만일 4-3-2-1의 순서로 문제를 풀게 되면 조건 1과 조건 2를 만족한다. 하지만 조건 3을 만족하지 않는다. 4보다 3을 충분히 먼저 풀 수 있기 때문이다. 따라서 조건 3을 만족하는 문제를 풀 순서는 3-1-4-2가 된다.

문제의 개수와 먼저 푸는 것이 좋은 문제에 대한 정보가 주어졌을 때, 주어진 조건을 만족하면서 민오가 풀 문제의 순서를 결정해 주는 프로그램을 작성하시오.



생각해봅시다.





암시적 그래프(implicit graph)

- 문제집 문제
 - 문제집 간의 순서가 존재했다
- 15-Puzzle
 - 현재 퍼즐의 state
 - 퍼즐의 타일을 움직였을 때 바뀌는 state의 표현



암시적 그래프(implicit graph)

- 문제집 문제

- 문제집 간의 순서가 존재했다



방향 그래프!

- 15-Puzzle

- 현재 퍼즐의 state

- 퍼즐의 타일을 움직였을 때 바뀌는 state의 표현



암시적 그래프(implicit graph)

- 문제집 문제

- 문제집 간의 순서가 존재했다

➔ 방향 그래프!

- 15-Puzzle

- 현재 퍼즐의 state

➔ vertex

- 퍼즐의 타일을 움직였을 때 바뀌는 state의 표현



암시적 그래프(implicit graph)

- 문제집 문제

- 문제집 간의 순서가 존재했다

➔ 방향 그래프!

- 15-Puzzle

- 현재 퍼즐의 state

➔ vertex

- 퍼즐의 타일을 움직였을 때 바뀌는 state의 표현

➔ edge



암시적 그래프(implicit graph)

- 문제집 문제

- 문제집 간의 순서가 존재했다

➔ 방향 그래프!

- 15-Puzzle

- 현재 퍼즐의 state

➔ vertex

- 퍼즐의 타일을 움직였을 때 바뀌는 state의 표현

➔ edge



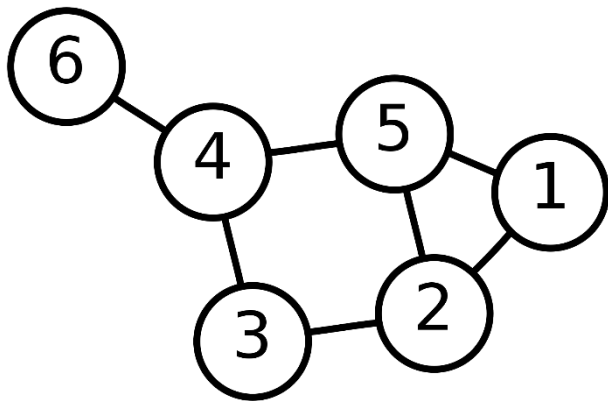
현재 state와 마지막 완성 state간의 최단경로



그래프의 표현

- 인접 행렬(adjacent matrix)

정점 간 연결 여부를 행렬로 판단하는 법

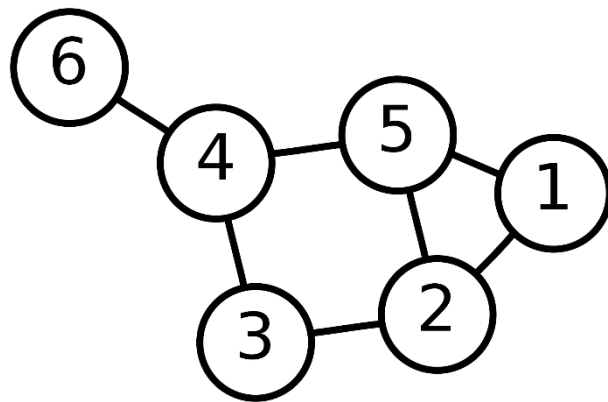




그래프의 표현

- 인접 행렬(adjacent matrix)

정점 간 연결 여부를 행렬로 판단하는 법



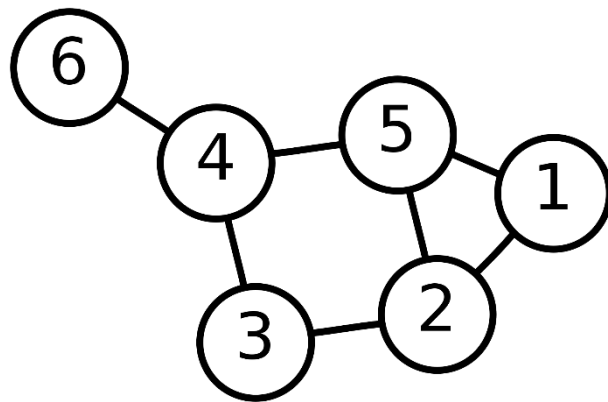
	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	0	1	0
3	0	1	0	1	0	0
4	0	0	1	0	1	1
5	1	1	0	1	0	0
6	0	0	0	1	0	0



그래프의 표현

- 인접 행렬(adjacent matrix)

정점 간 연결 여부를 행렬로 판단하는 법



	1	2	3	4	5	6
1		1			1	
2	1		1		1	
3		1		1		
4			1		1	1
5	1	1		1		
6				1		

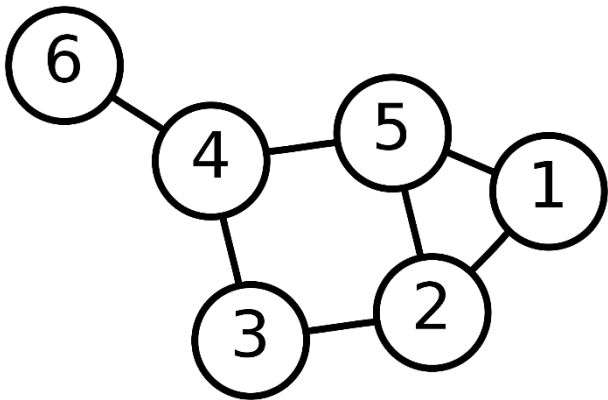


```
int V, E;
int adj_matrix[MAX_V][MAX_V];
int u, v;
for (int i = 0; i < E; i++) {
    scanf("%d%d", &u, &v);
    adj_matrix[u][v] = adj_matrix[v][u] = 1;
}
```



그래프의 표현

- 인접 리스트(adjacent list)
연결된 정점만을 추가한 것



1	→	2		→	5			
2	→	1		→	3		→	5
3	→	2		→	4			
4	→	3		→	5		→	6
5	→	1		→	2		→	4
6	→	4						



```
#include <stdio.h>
#include <stdlib.h>
#define MAX_V 101

typedef struct node* nptr;
typedef struct node {
    int vertex;
    nptr link;
}NODE;

int V, E;
nptr adj_list[MAX_V];
```

```
void push(int u, int v) {
    nptr tmp = (nptr)malloc(sizeof(NODE));
    tmp->vertex = v;
    tmp->link = NULL;
    if (adj_list[u]) tmp->link = adj_list[u];
    adj_list[u] = tmp;
}

int main() {
    int u, v;
    for (int i = 0; i < E; i++) {
        scanf("%d%d", &u, &v);
        push(u, v);
        push(v, u);
    }
}
```



배운 벡터를 써봅시다.

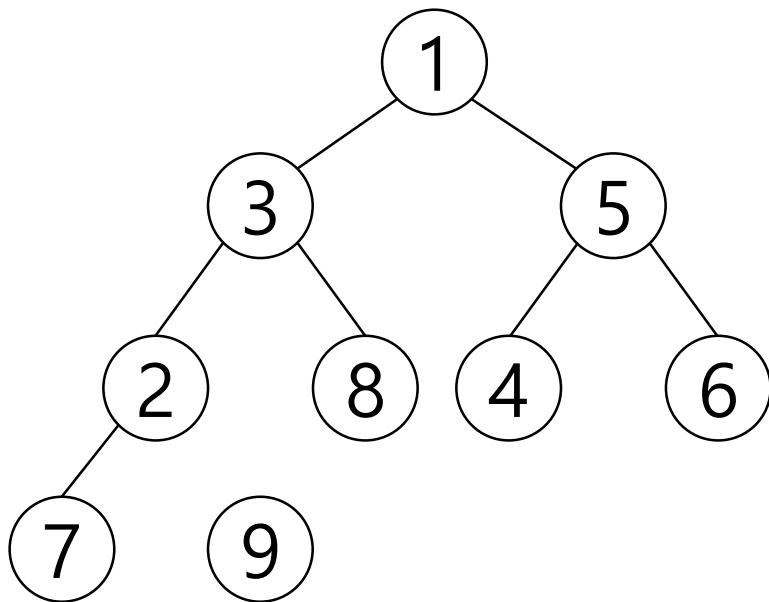
```
#include <vector>
#include <stdio.h>
using namespace std;
#define MAX_V 101

vector<int> adj_list[MAX_V];
int V, E;
int main() {
    int u, v;
    for (int i = 0; i < E; i++) {
        scanf("%d%d", &u, &v);
        adj_list[u].push_back(v);
        adj_list[v].push_back(u);
    }
}
```



그래프 탐색

- 그래프 $G = (V, E)$ 에 있는 $v \in V(G)$ 에 대하여
 v 와 연결되어 있는 정점들의 정보를 알고 싶을 때 그래프 탐색이 필요.







Depth First Search(DFS)

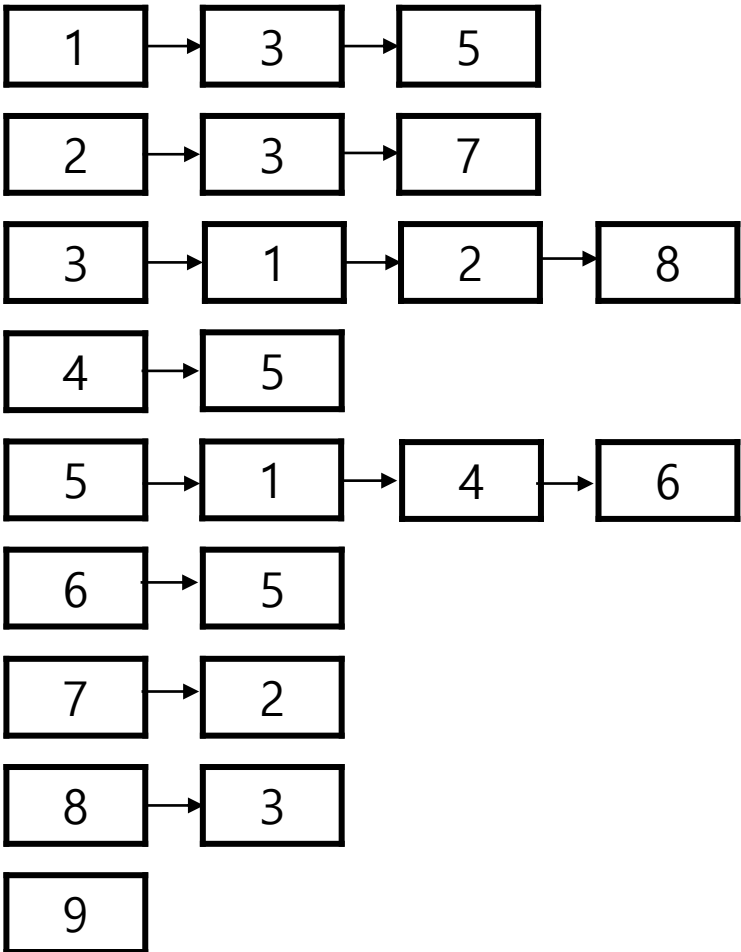
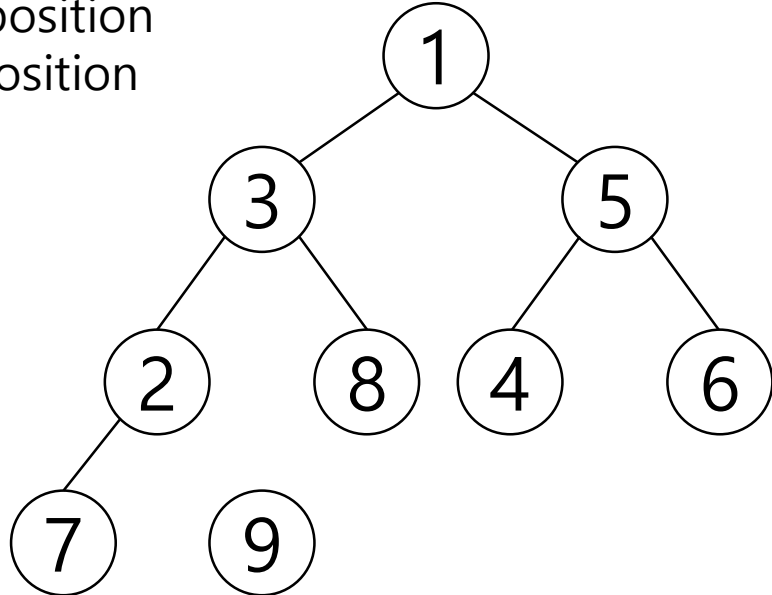
- 한 노드부터 시작하여 탐색할 수 있는 최대 말단까지 탐색을 하는 방법
- 탐색 실패 시 다른 인접한 노드에 대해서 다시 탐색할 수 있는 최대 말단까지 탐색



Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

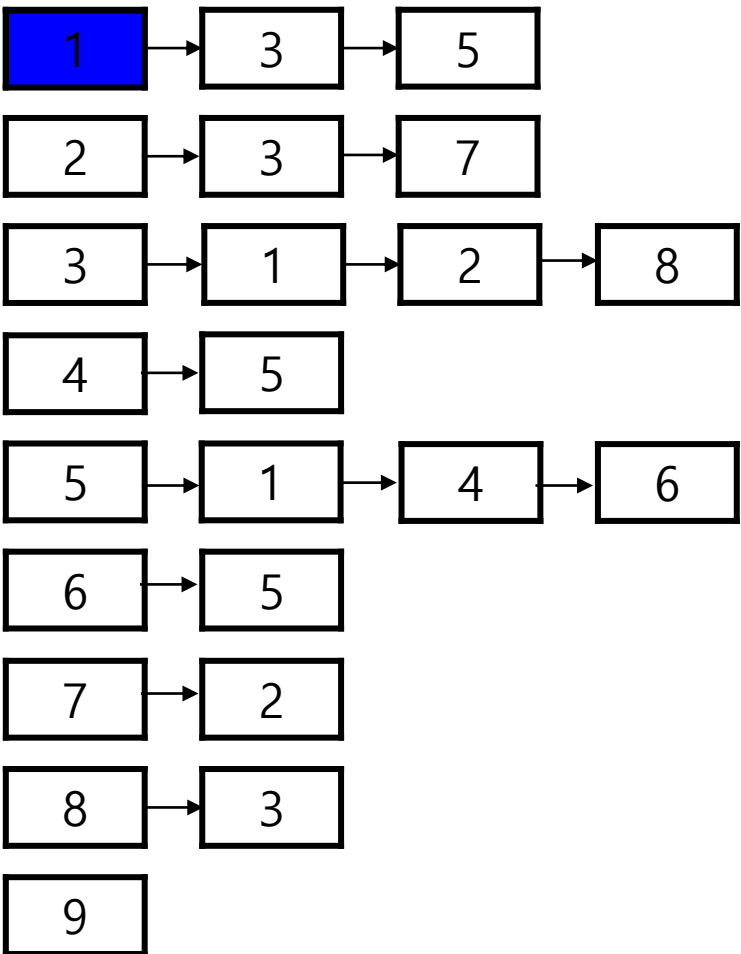
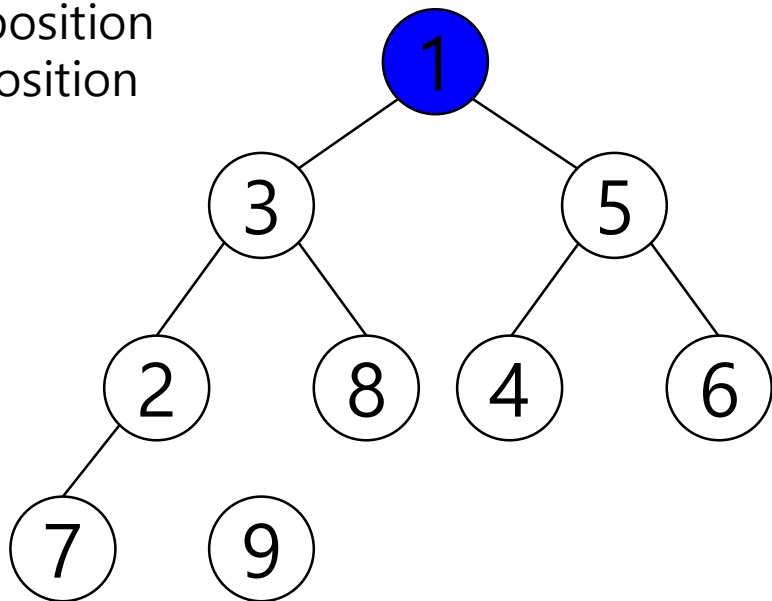




Depth First Search(DFS)

start location : 1
finish location : 6

 : current position
 : visited position





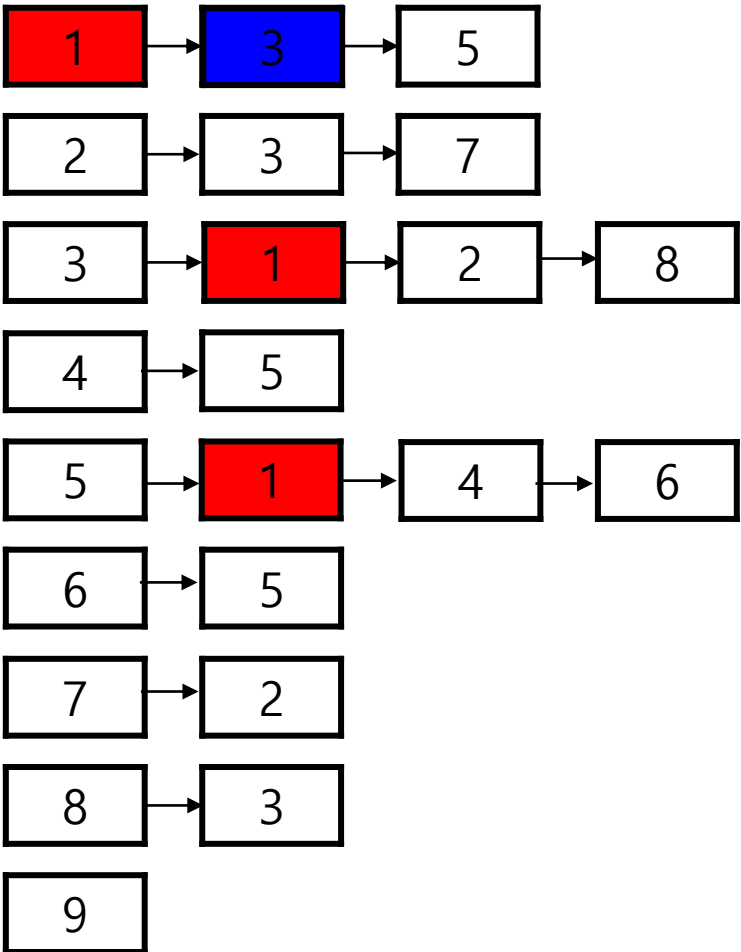
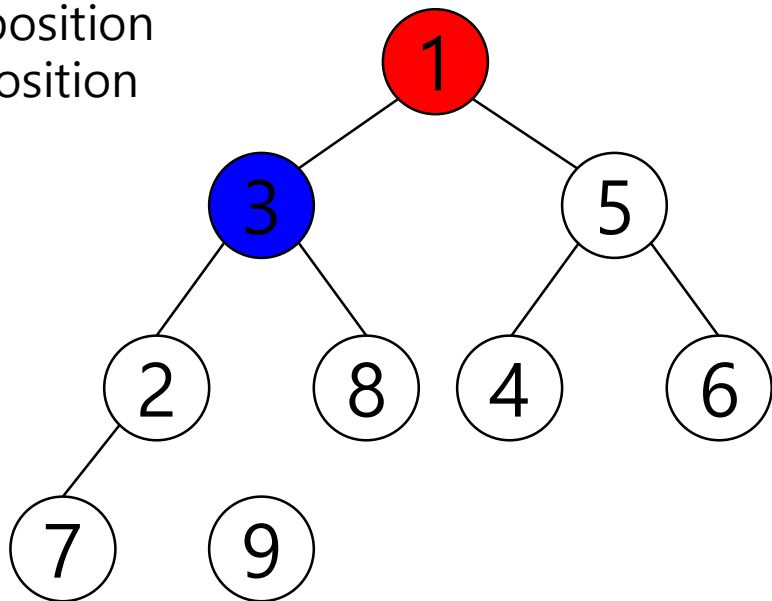


Depth First Search(DFS)

start location : 1

finish location : 6



 : current position
 : visited position

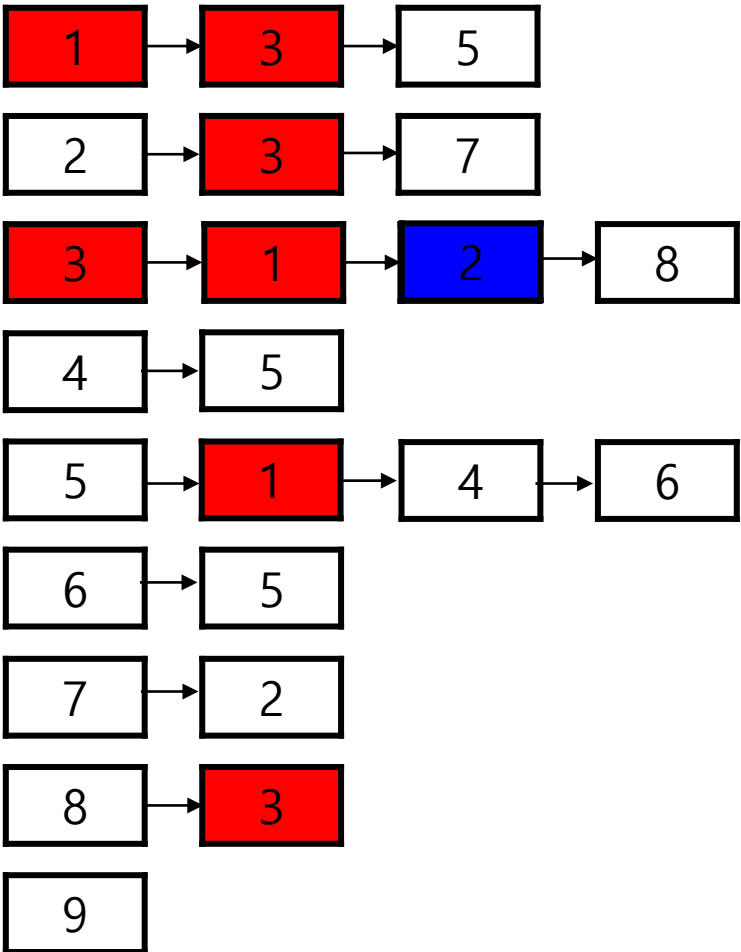
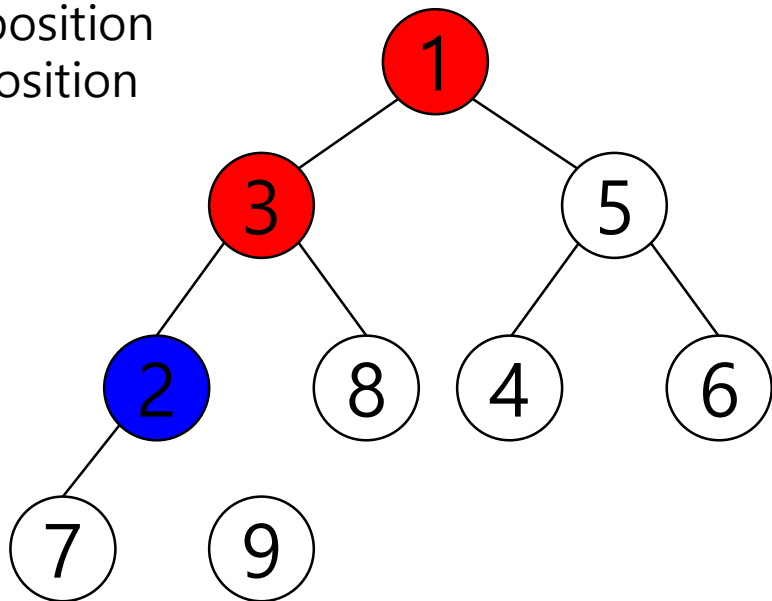




Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

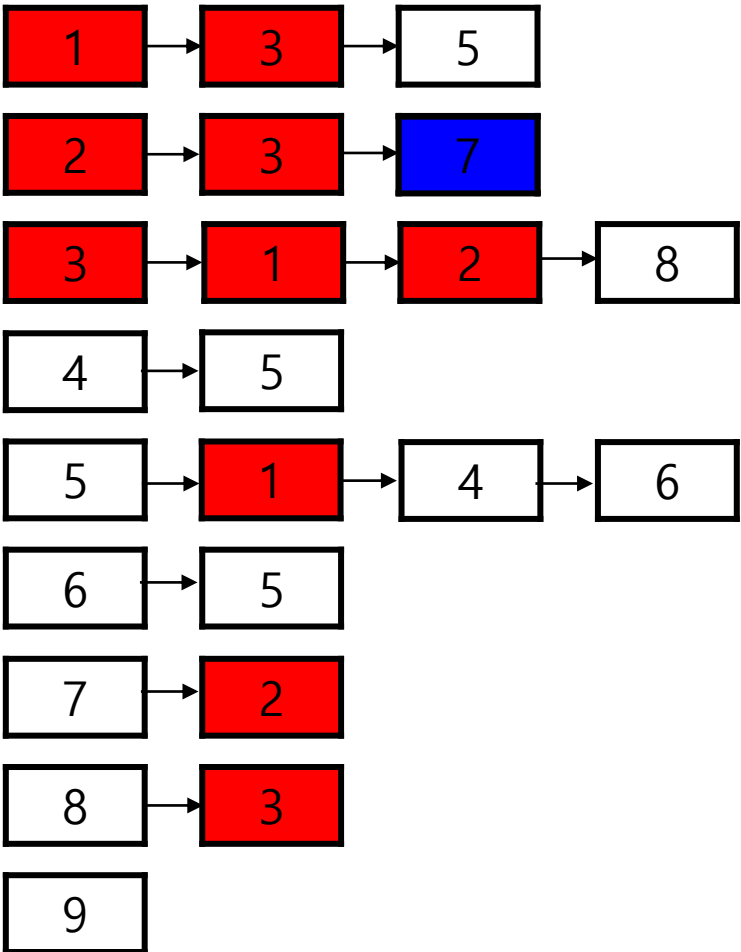
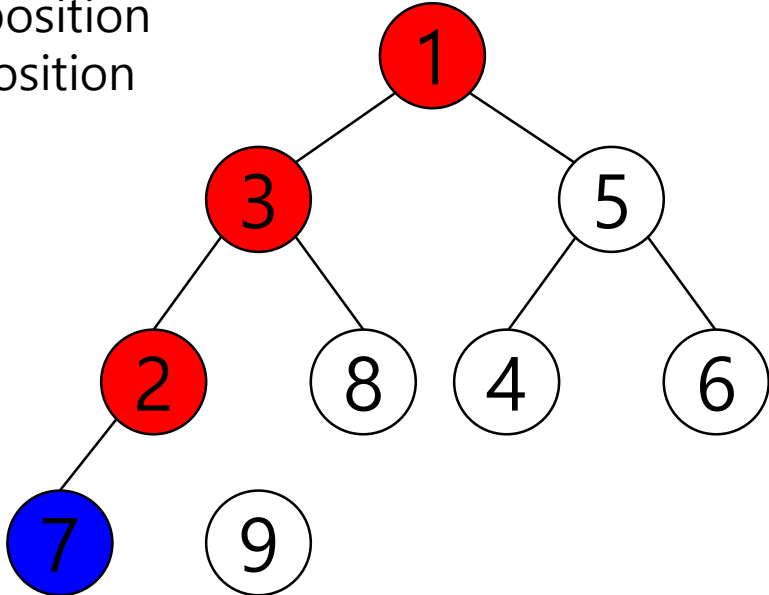




Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

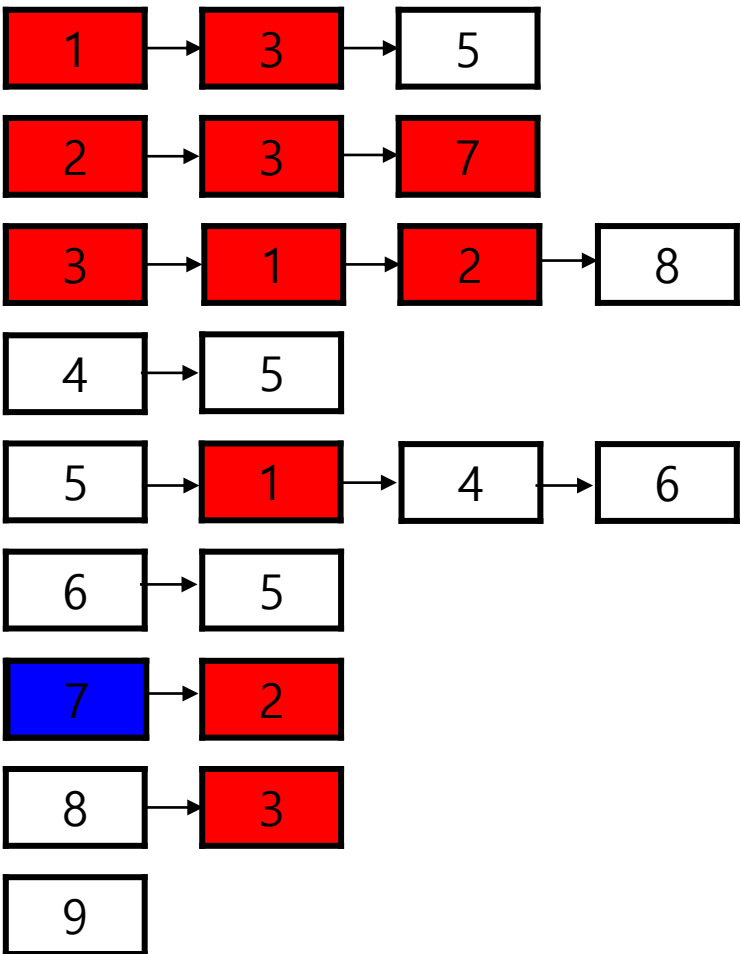
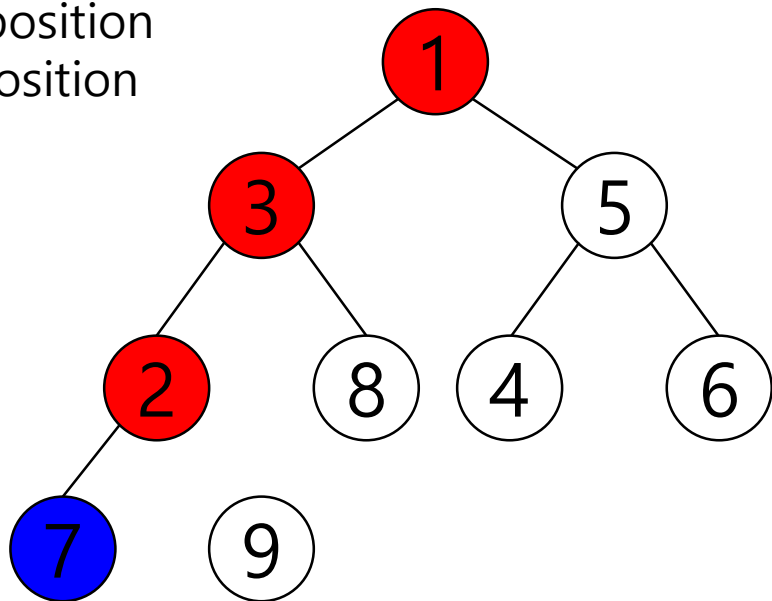




Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

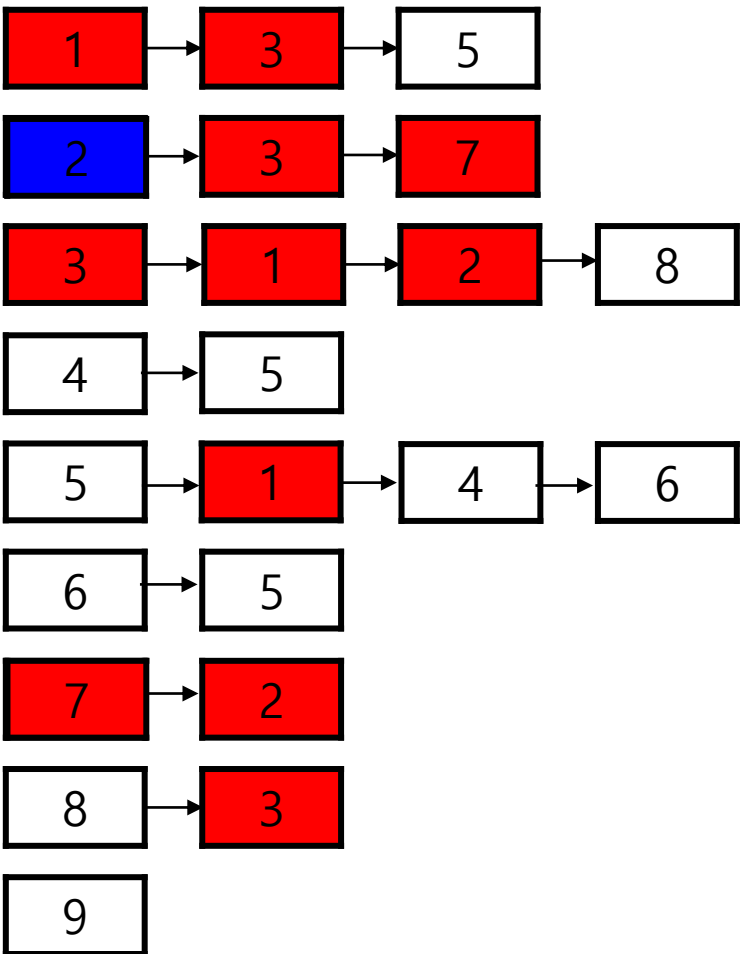
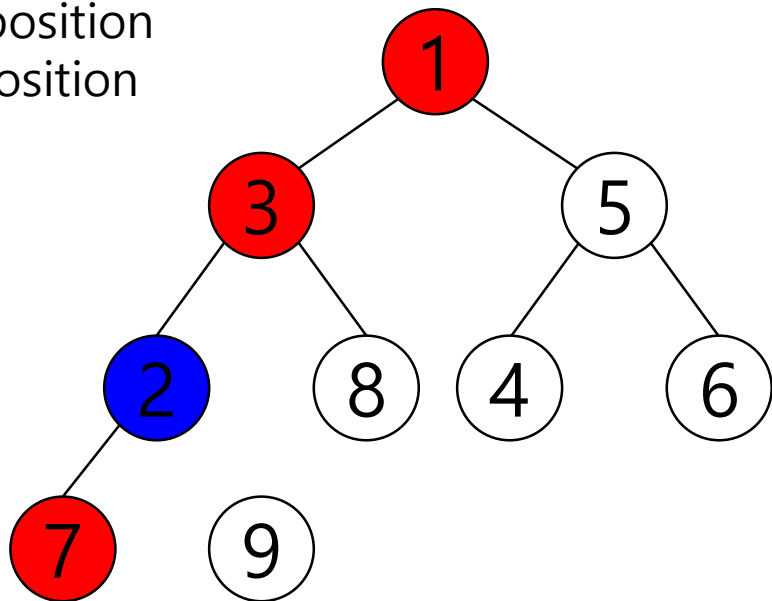




Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

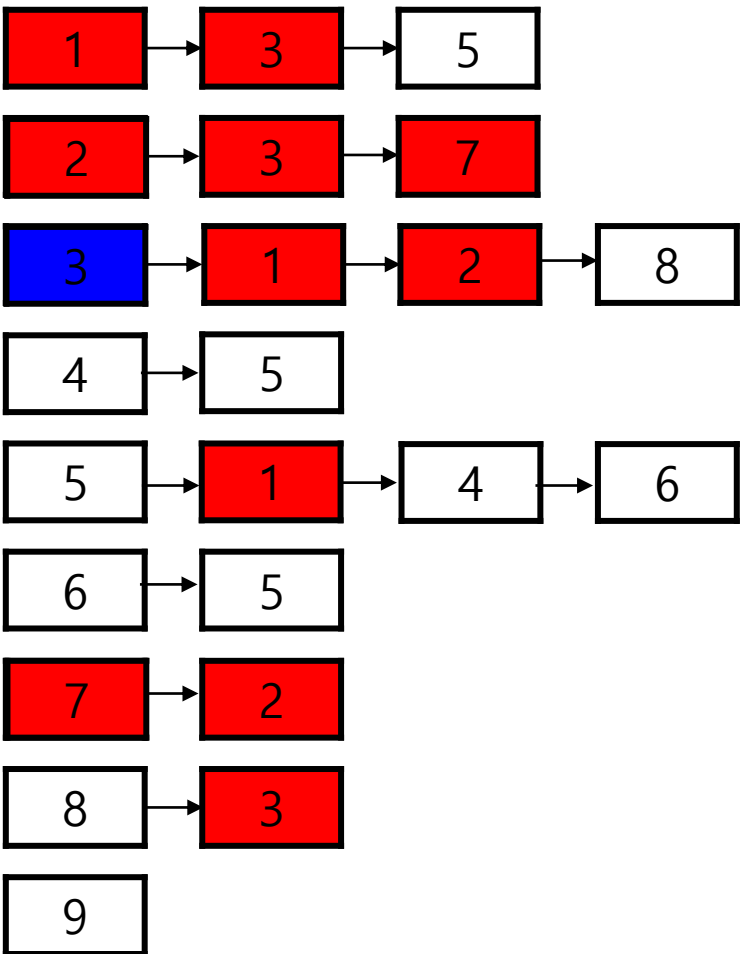
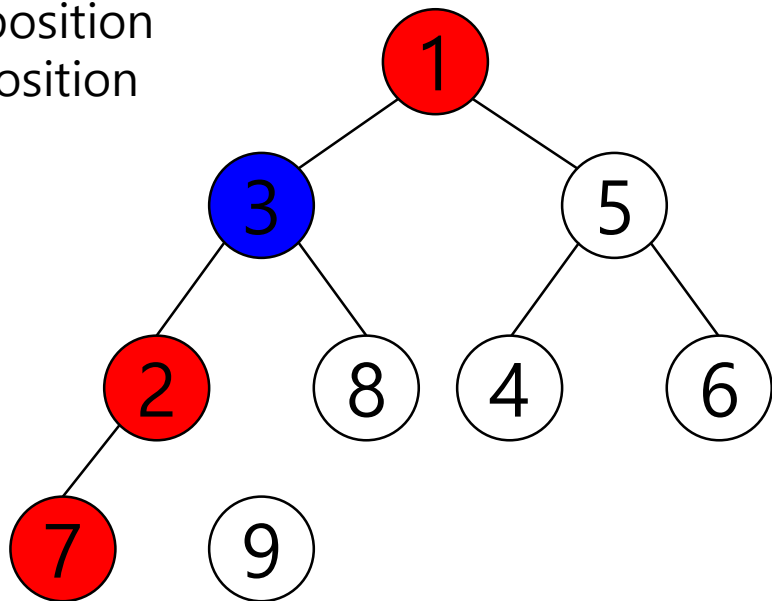




Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

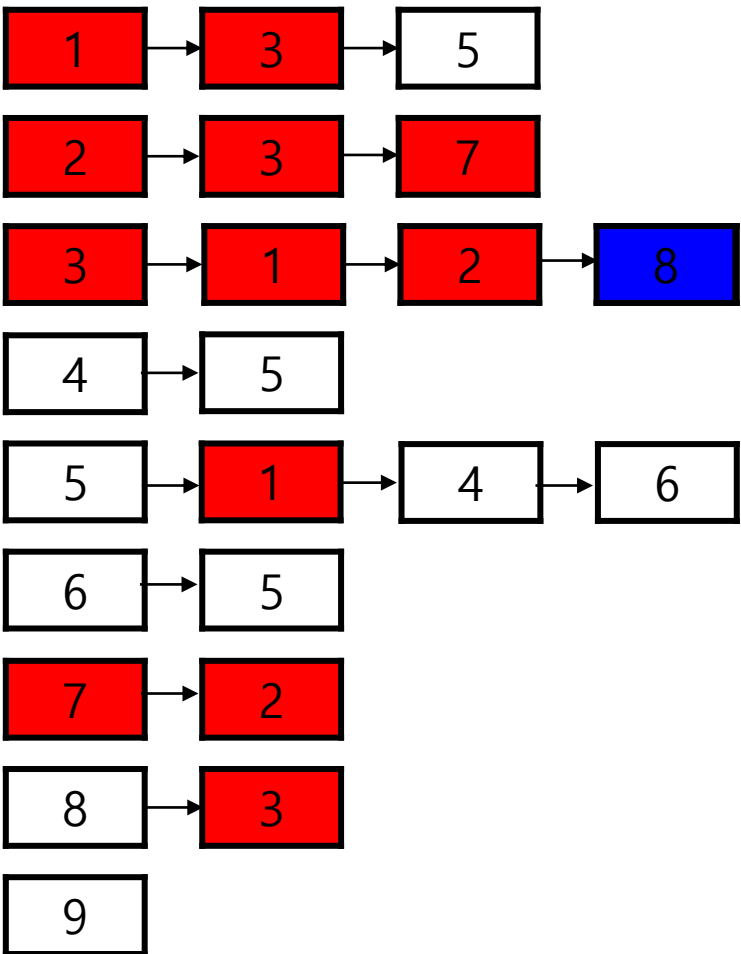
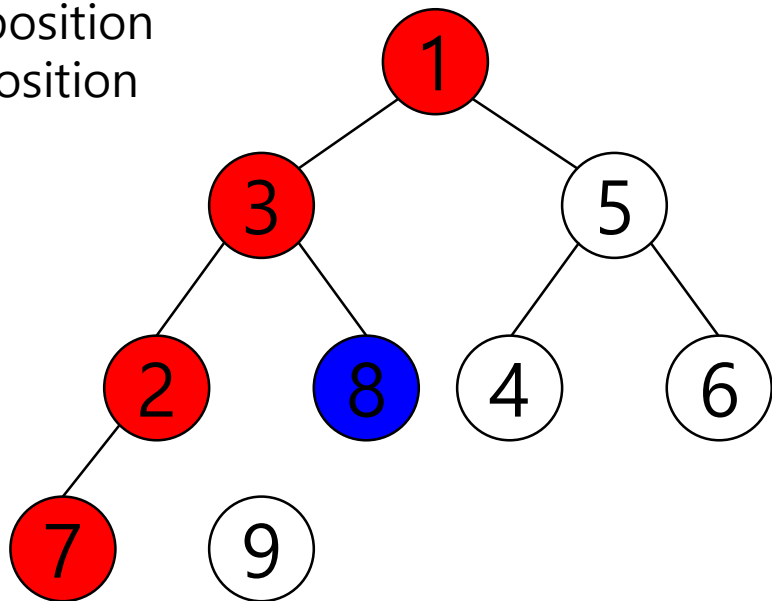




Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

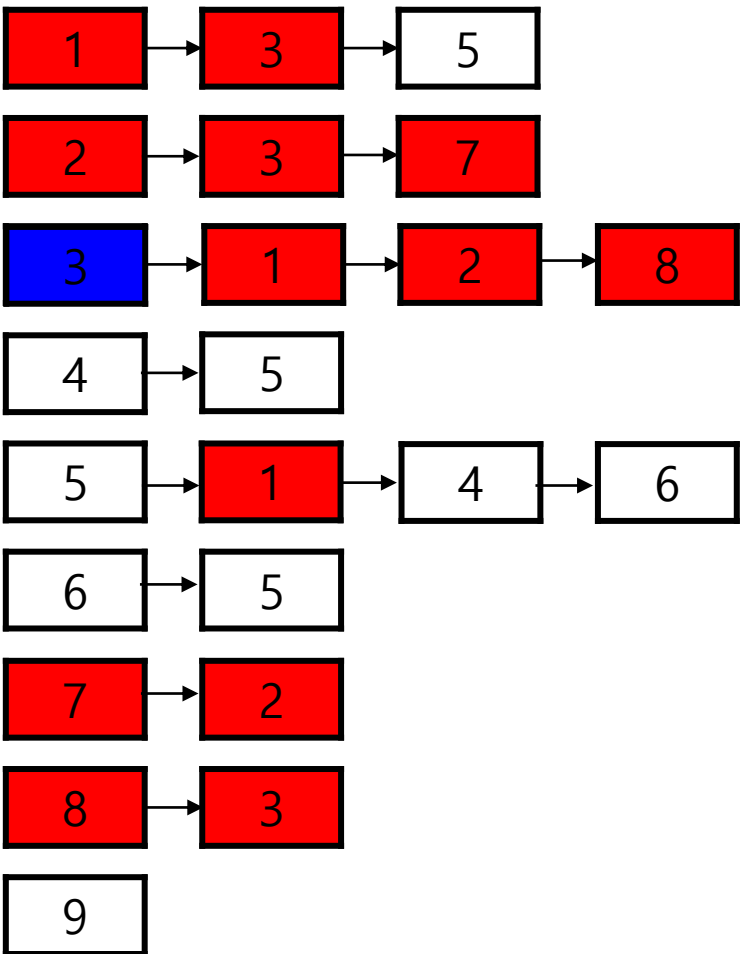
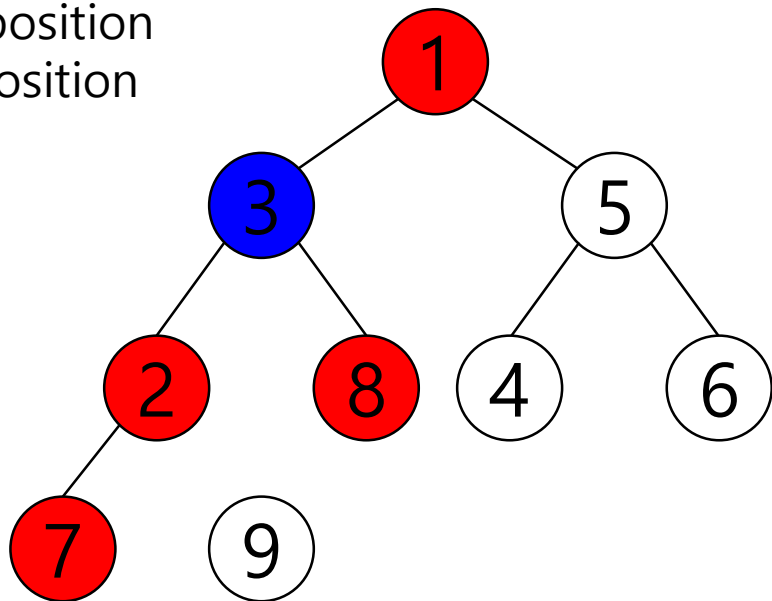




Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

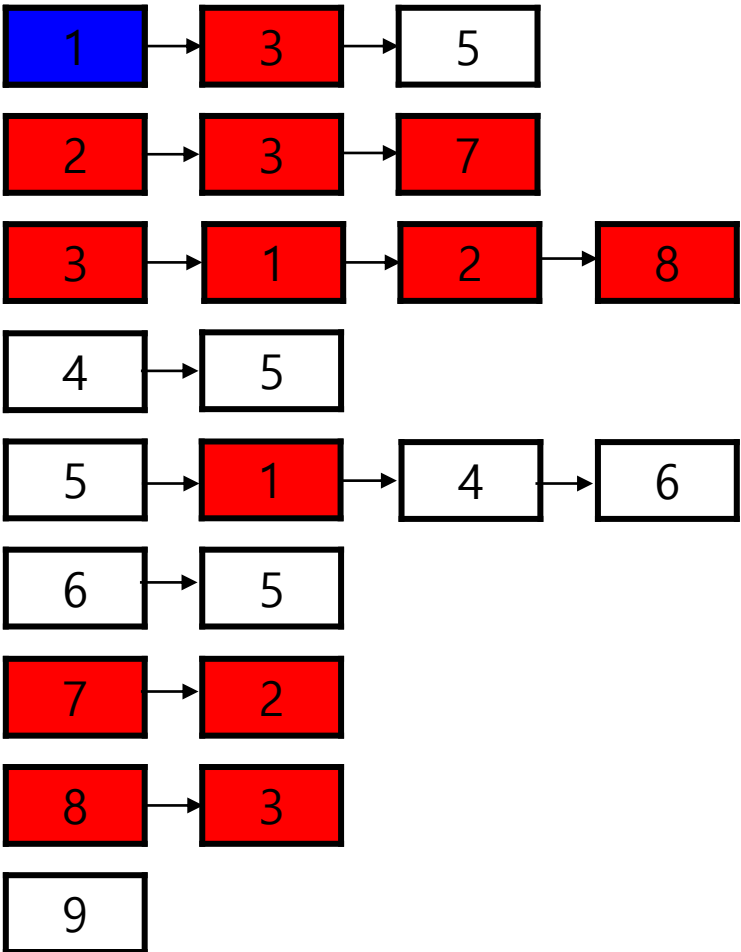
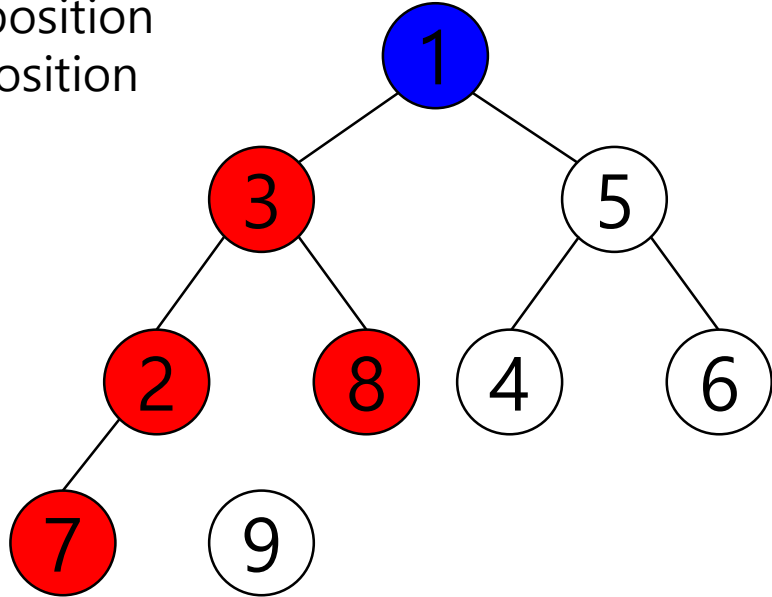




Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

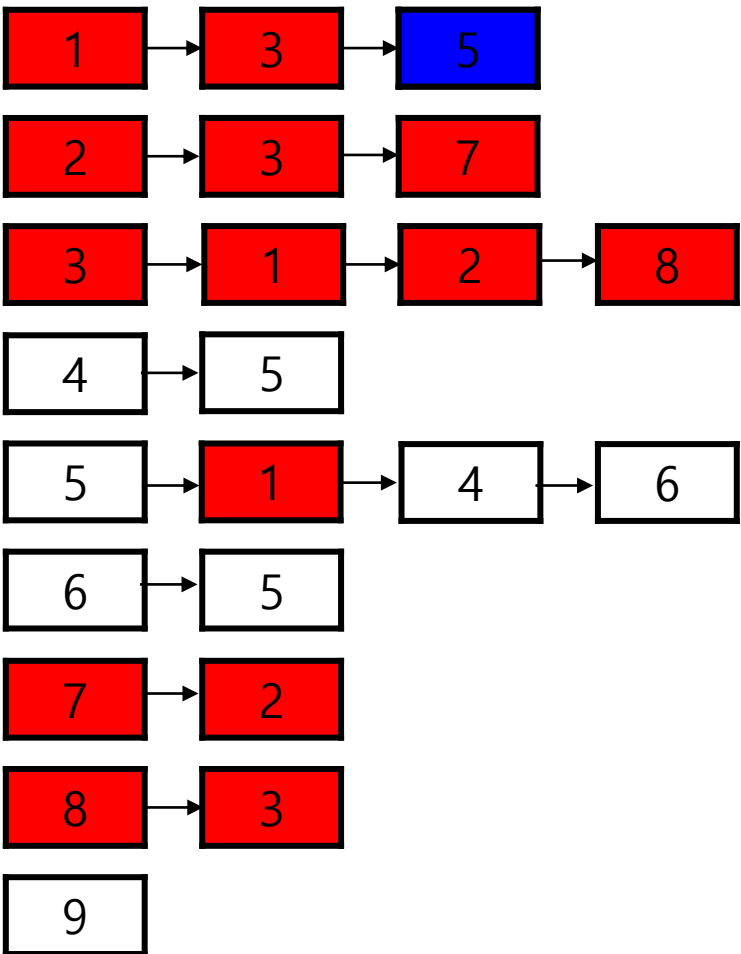
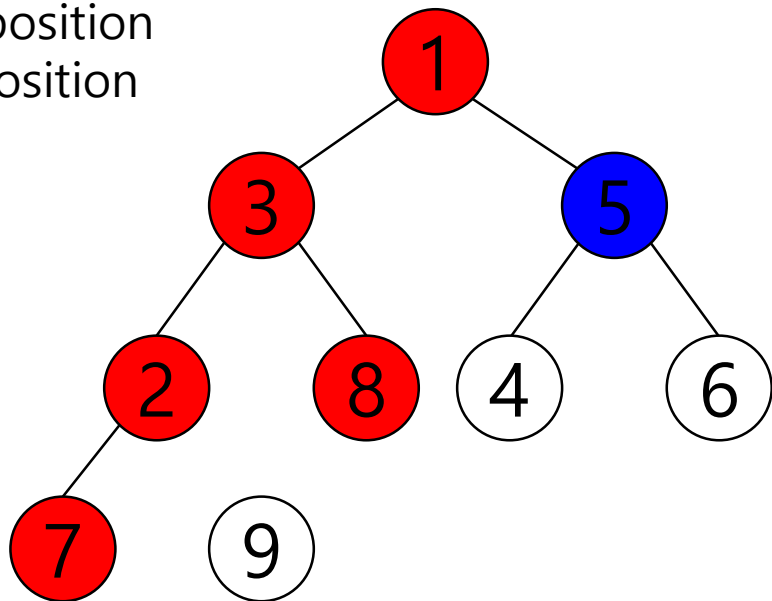




Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

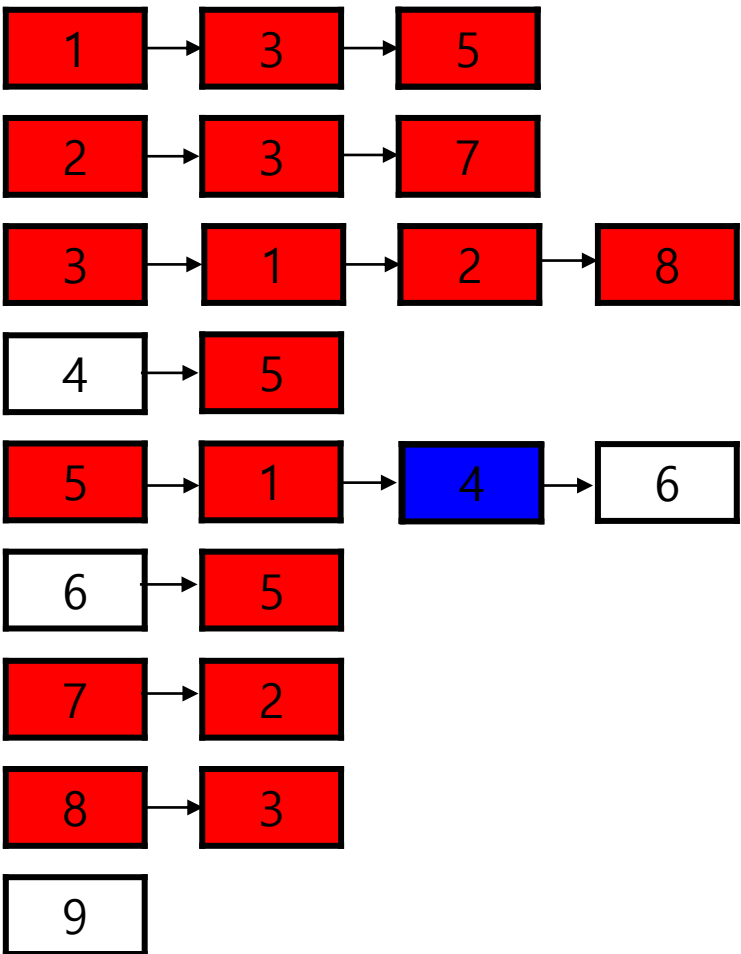
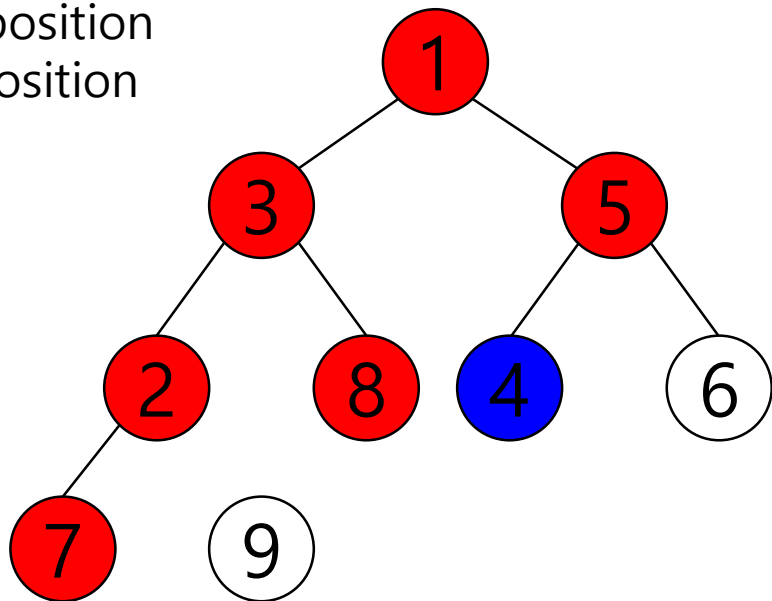




Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

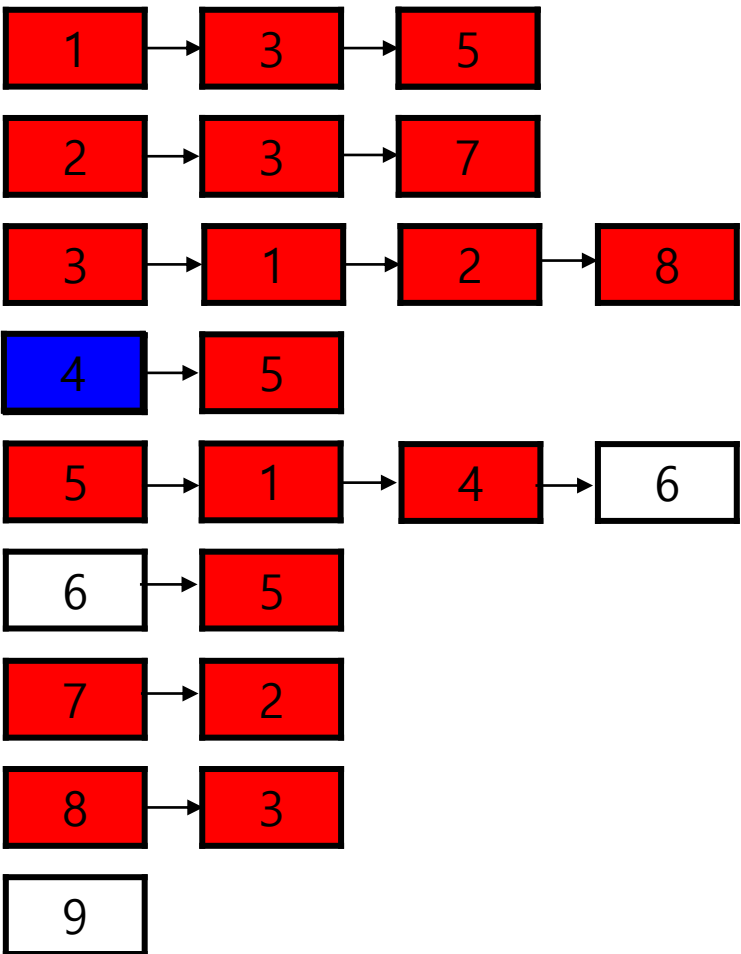
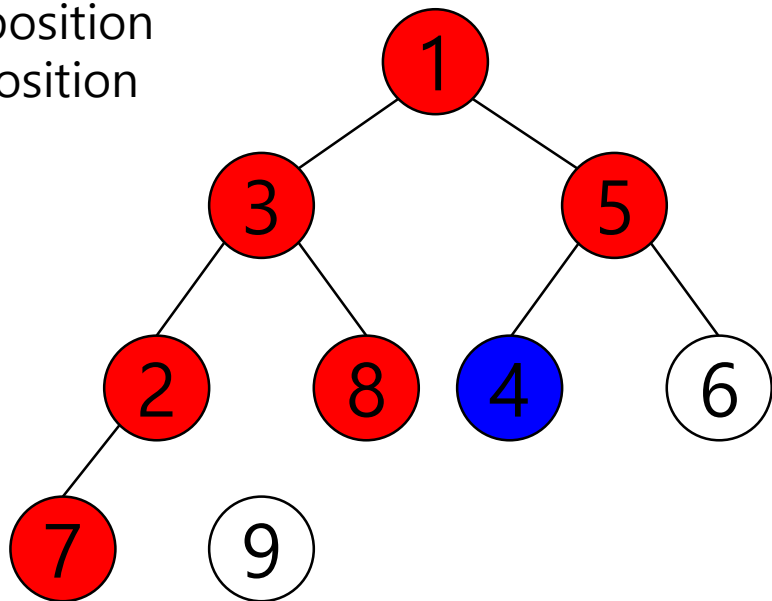




Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

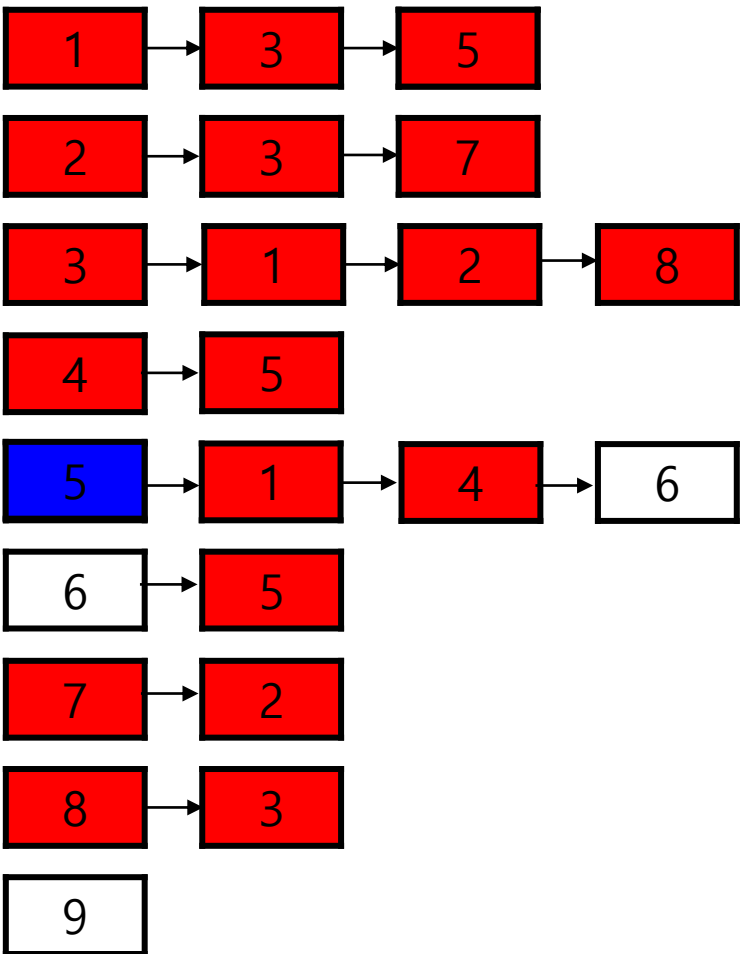
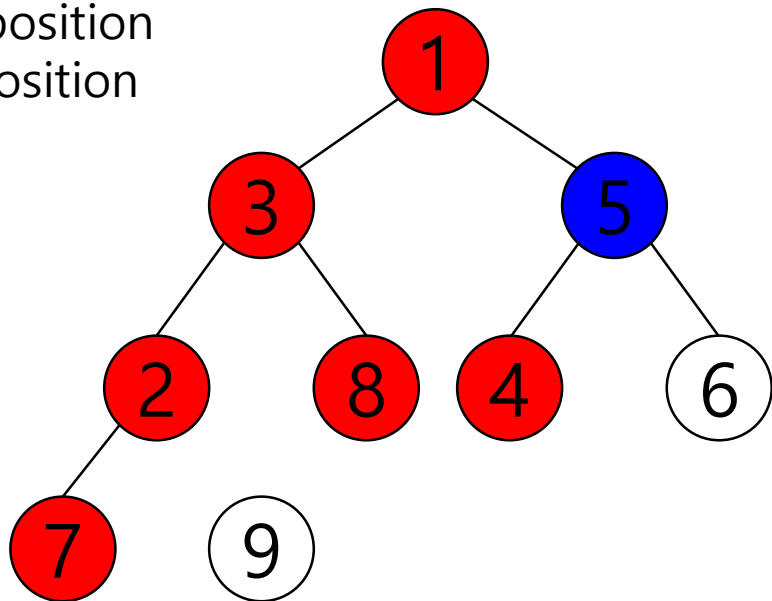




Depth First Search(DFS)

start location : 1
finish location : 6



 : current position
 : visited position

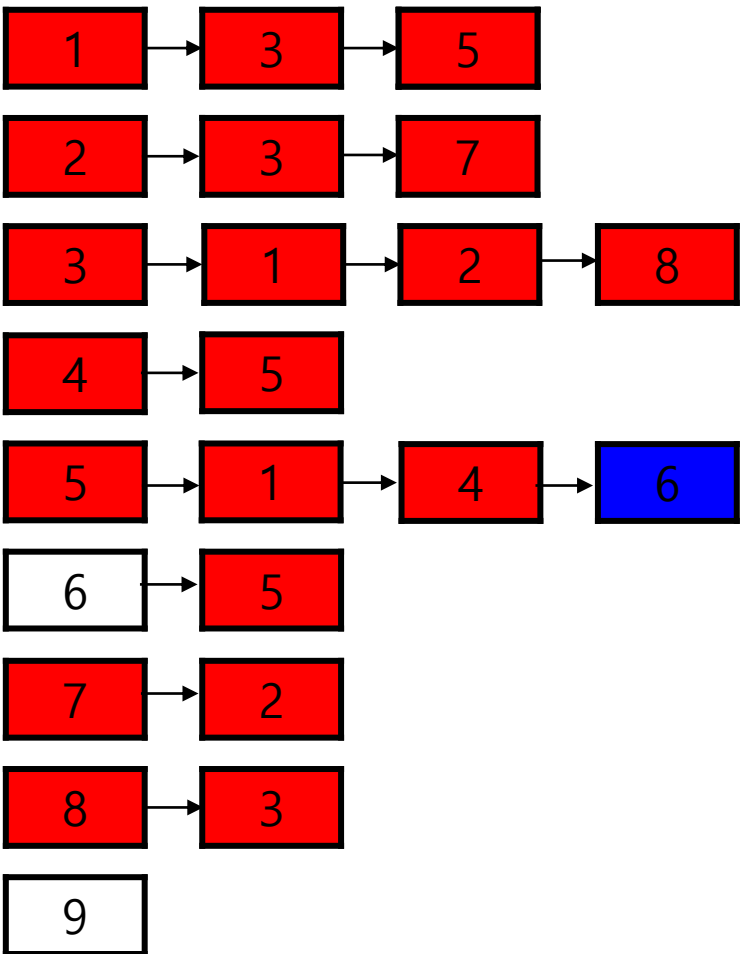
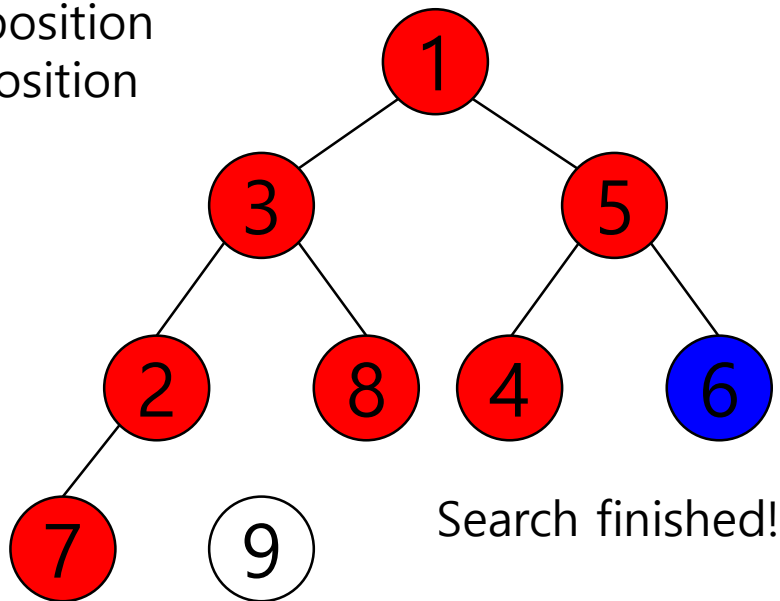




Depth First Search(DFS)

start location : 1
finish location : 6

 : current position
 : visited position





인접 리스트를 이용한 DFS 구현

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_V 101

typedef struct node* nptr;
typedef struct node {
    int vertex;
    nptr link;
}NODE;
```

```
int V, E;
nptr adj_list[MAX_V];
bool visited[MAX_V];
```

```
void dfs(int curr) {
    nptr w;
    visited[curr] = true;
    printf("%d ", curr);
    for (w = adj_list[curr]; w; w = w->link)
        if (!visited[w->vertex])
            dfs(w->vertex);
}
```




인접 리스트를 이용한 DFS 구현

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_V 101

typedef struct node* nptr;
typedef struct node {
    int vertex;
    nptr link;
}NODE;

int V, E;
nptr adj_list[MAX_V];
bool visited[MAX_V];
```

```
void dfs(int curr, int target) {
    nptr w;
    visited[curr] = true;
    if (curr == target) {
        _find = true;
        return;
    }
    for (w = adj_list[curr]; w; w = w->link)
        if (!visited[w->vertex])
            dfs(w->vertex, target);
}
```



vector을 쓴다면..?

```
#include <vector>
using namespace std;
#define MAX_V 101

vector<int> adj_list[MAX_V];
bool visited[MAX_V];

void dfs(int curr) {
    visited[curr] = true;
    printf("%d ", curr);
    for (int i = 0; i < adj_list[curr].size(); i++)
        if (!visited[adj_list[curr][i]])
            dfs(adj_list[curr][i]);
}
```



더 간결하게도 가능해요

```
void dfs(int curr) {  
    visited[curr] = true;  
    printf("%d ", curr);  
    for (int next : adj_list[curr])  
        if (!visited[next])  
            dfs(next);  
}
```





Breadth First Search(BFS)

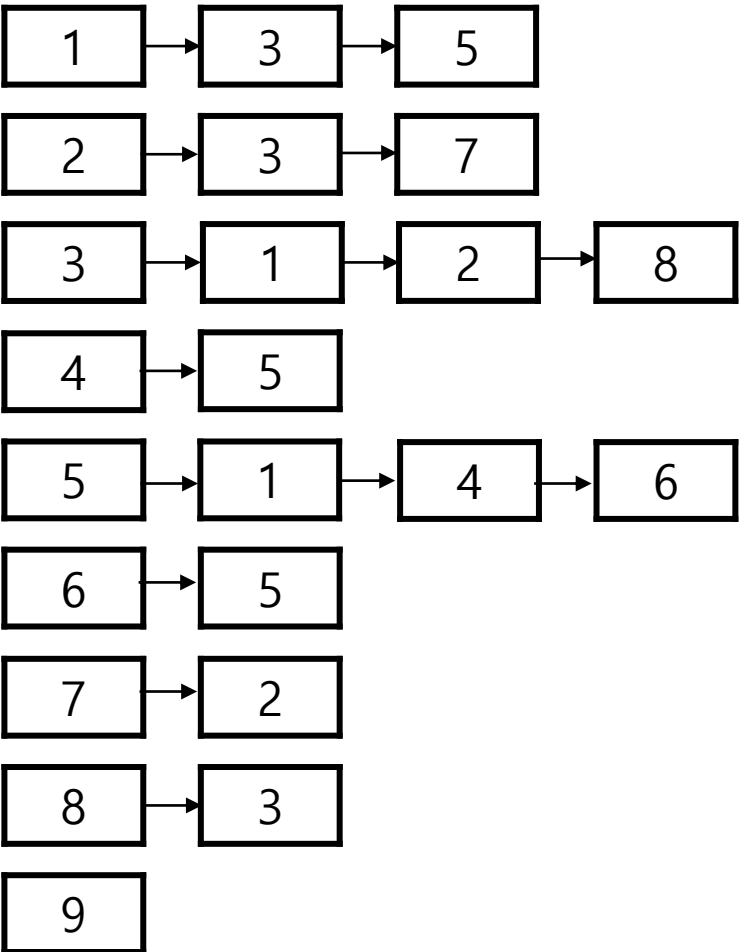
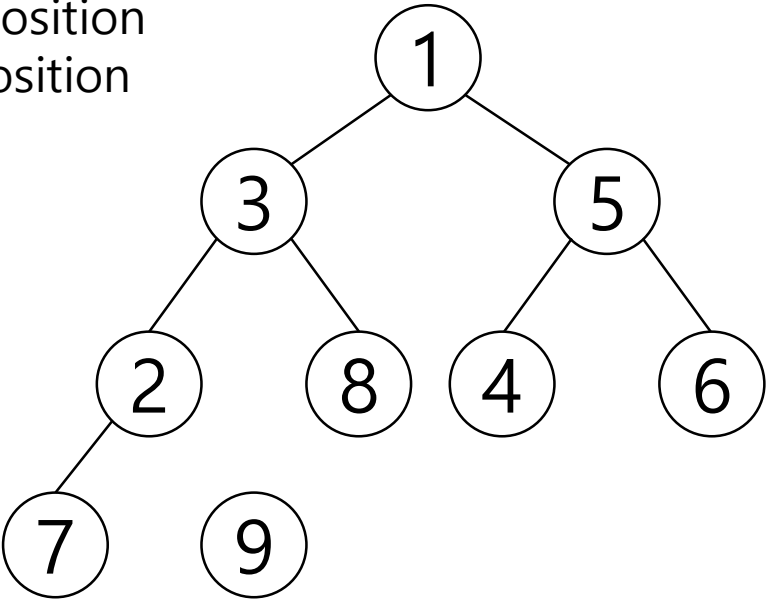
- 한 노드부터 시작하여 해당 노드에 인접한 노드들을 우선으로 탐색하는 방법
- 탐색 실패 시 이전 단계(Level)에서 검색했던 노드들에 대해 해당 노드에 인접한 노드들을 우선으로 탐색



Breadth First Search(BFS)

start location : 1
finish location : 6

 : current position
 : visited position





queue

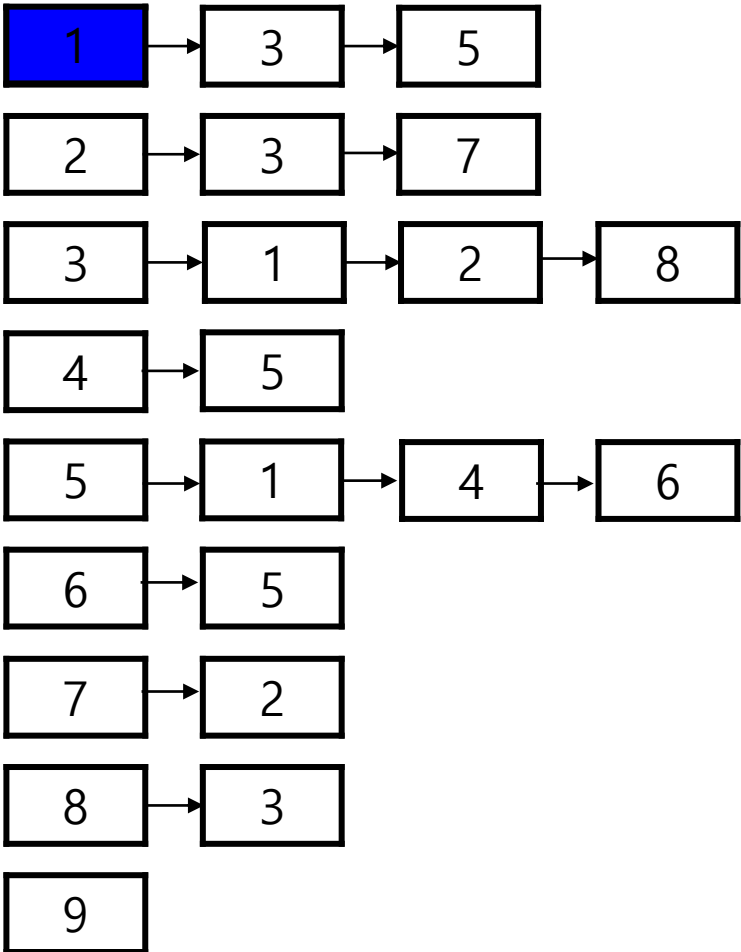
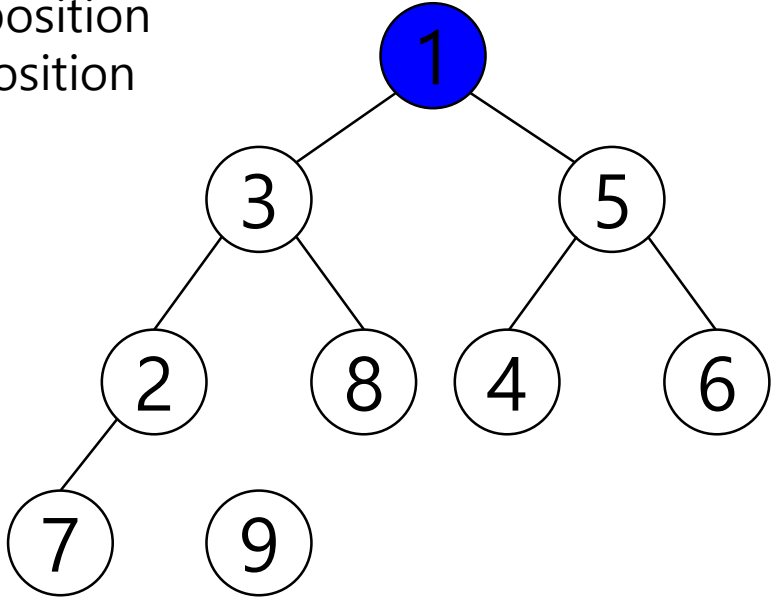




Breadth First Search(BFS)

start location : 1
finish location : 6

 : current position
 : visited position





queue

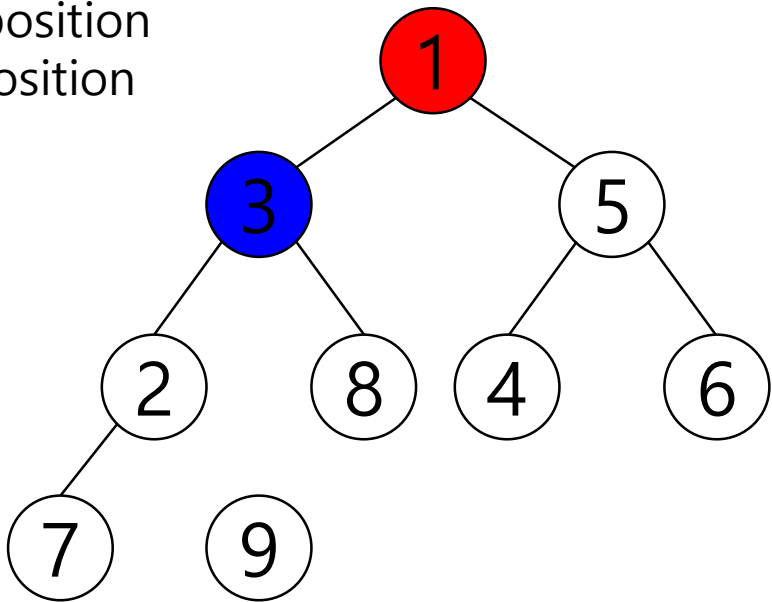




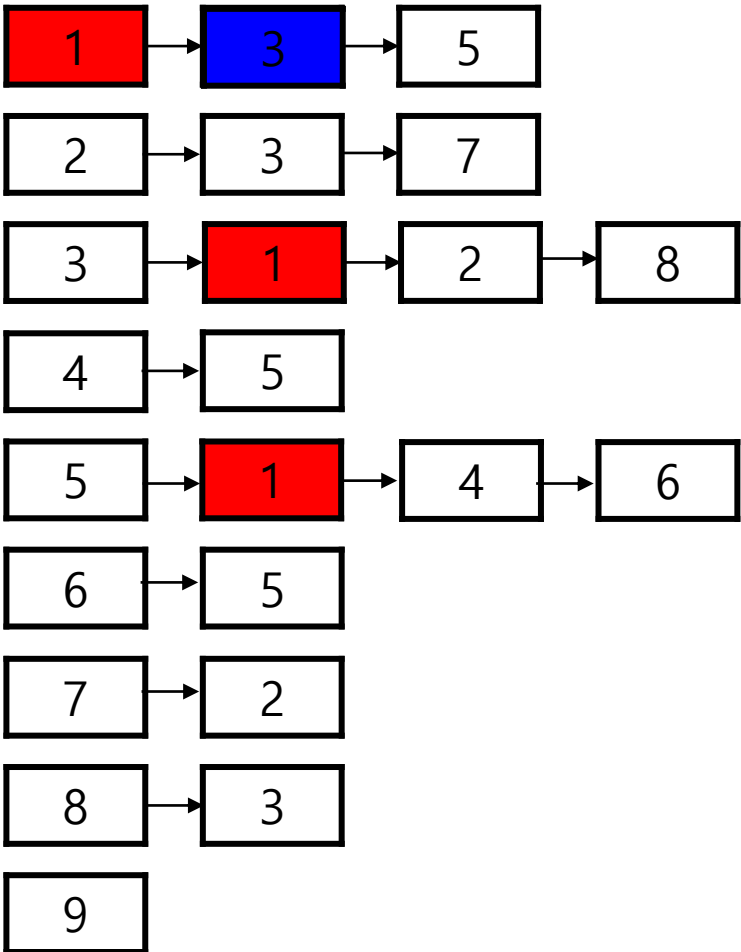
Breadth First Search(BFS)

start location : 1
finish location : 6

 : current position
 : visited position





queue

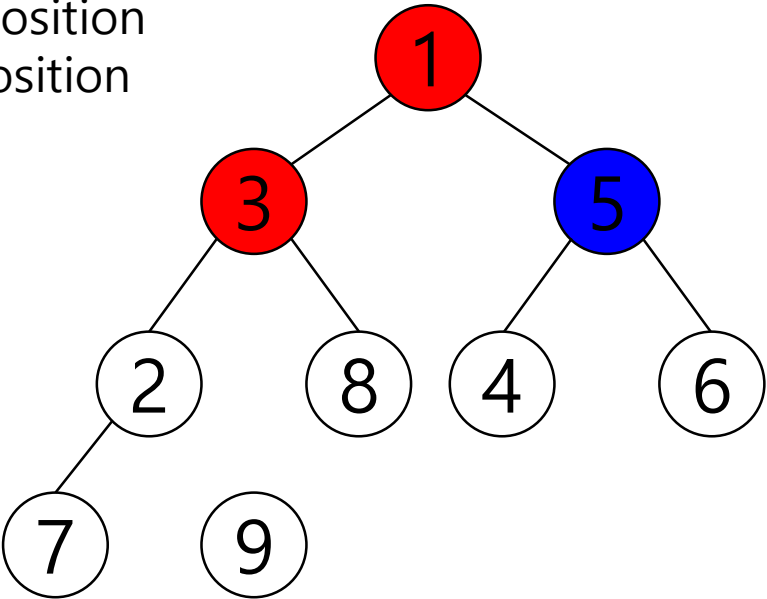




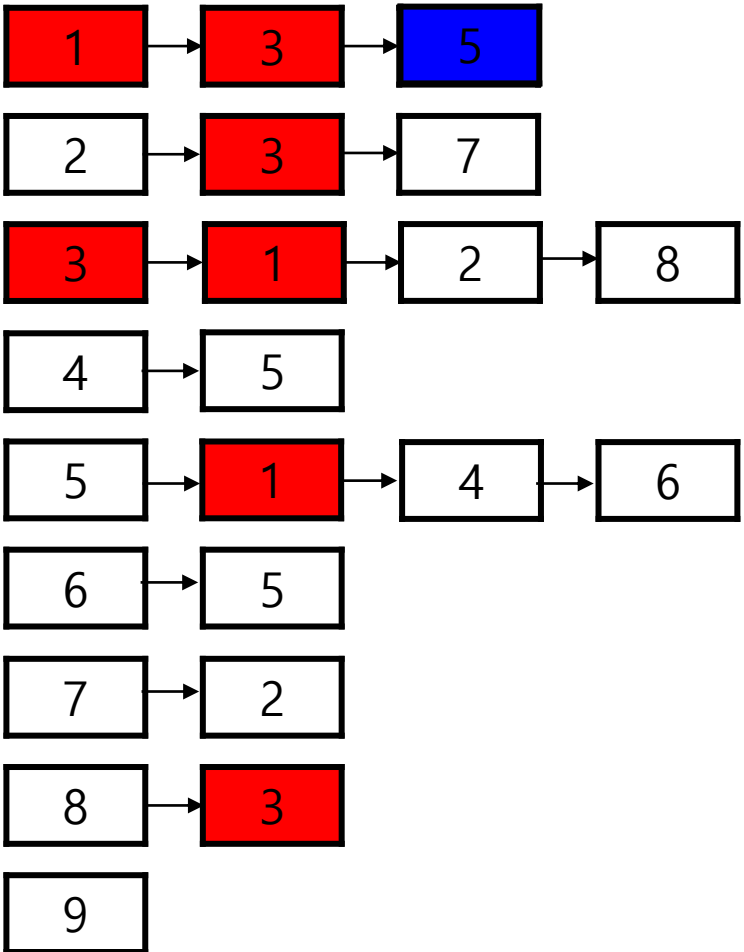
Breadth First Search(BFS)

start location : 1
finish location : 6

 : current position
 : visited position





queue

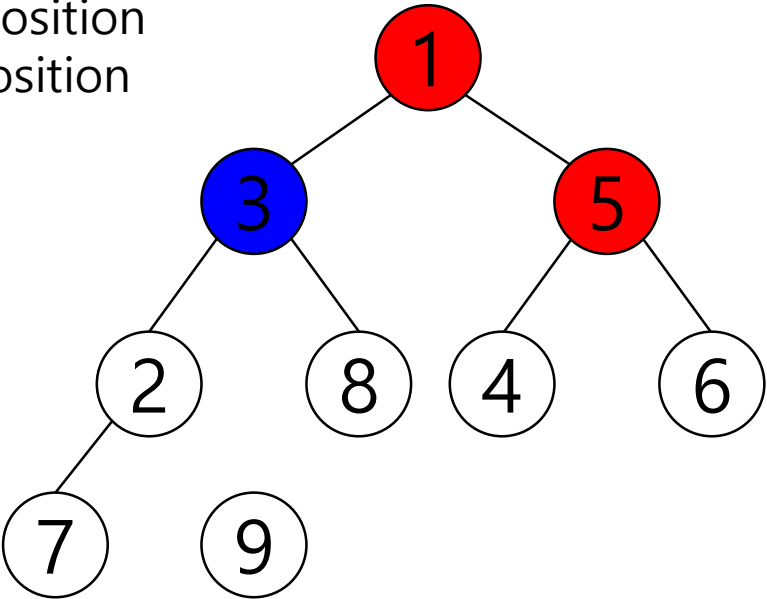




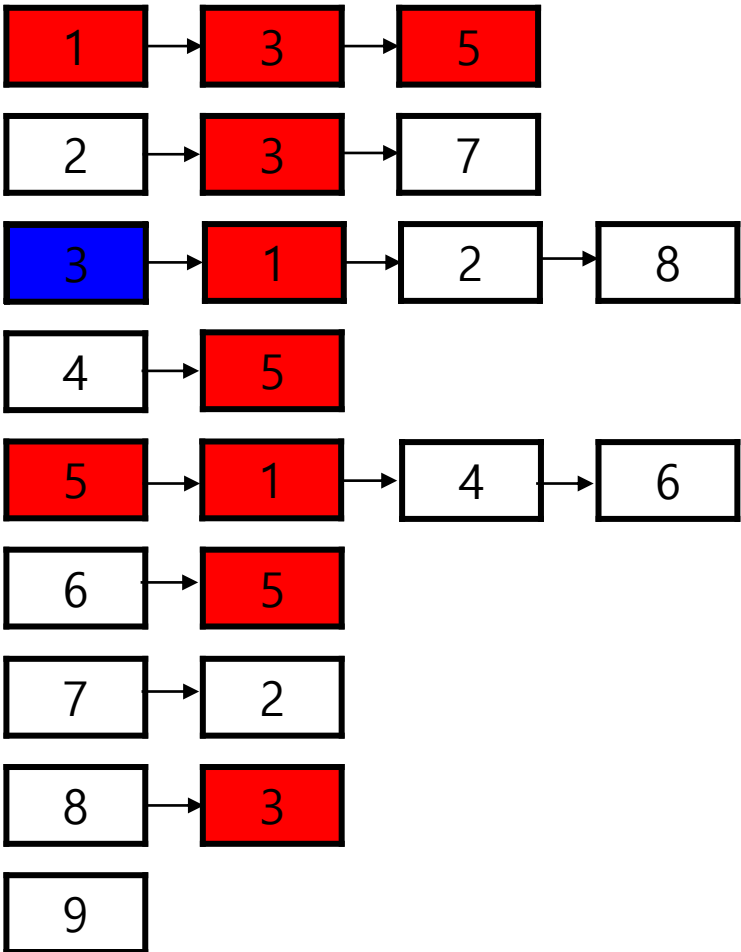
Breadth First Search(BFS)

start location : 1
finish location : 6

 : current position
 : visited position





queue

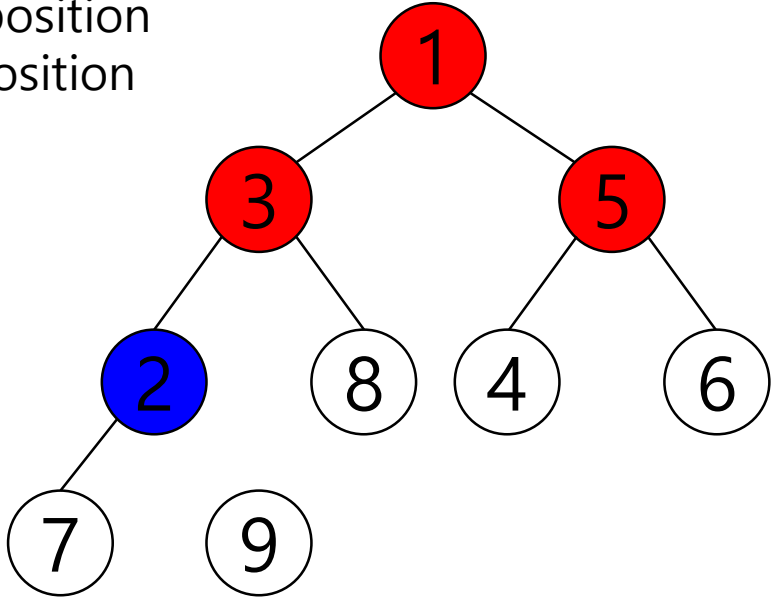




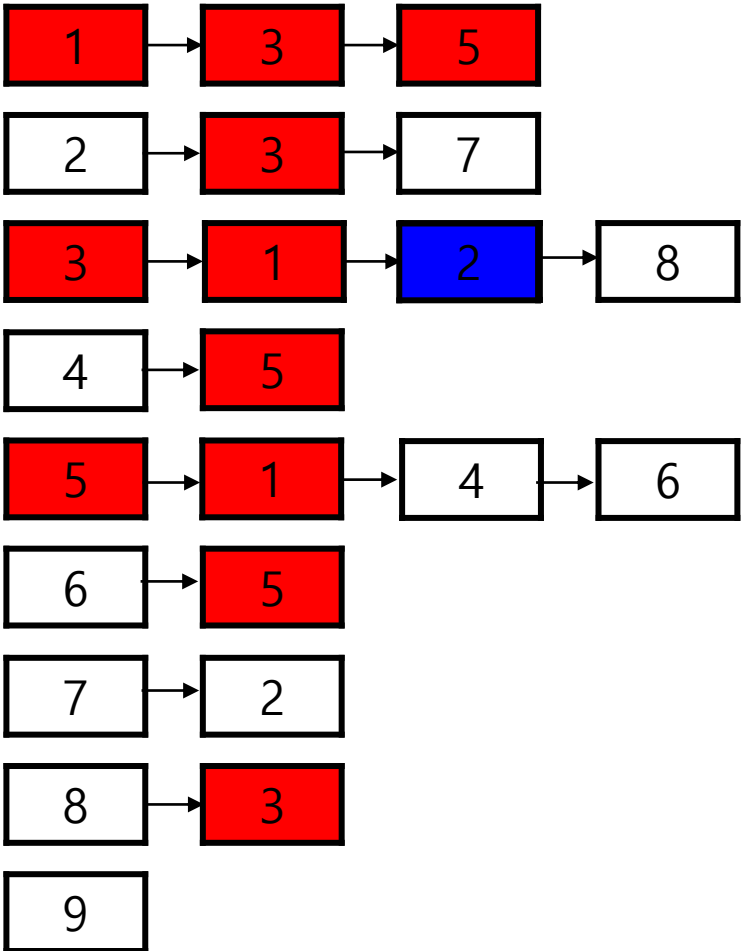
Breadth First Search(BFS)

start location : 1
finish location : 6

 : current position
 : visited position





queue

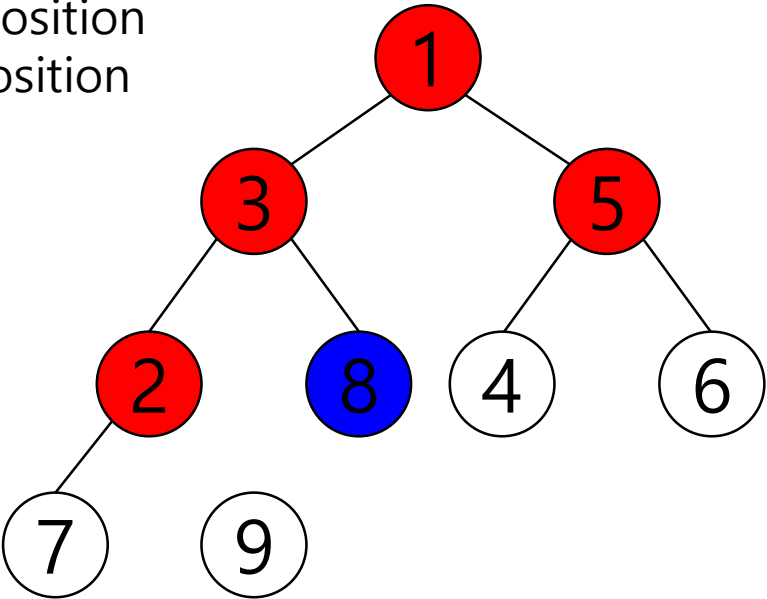




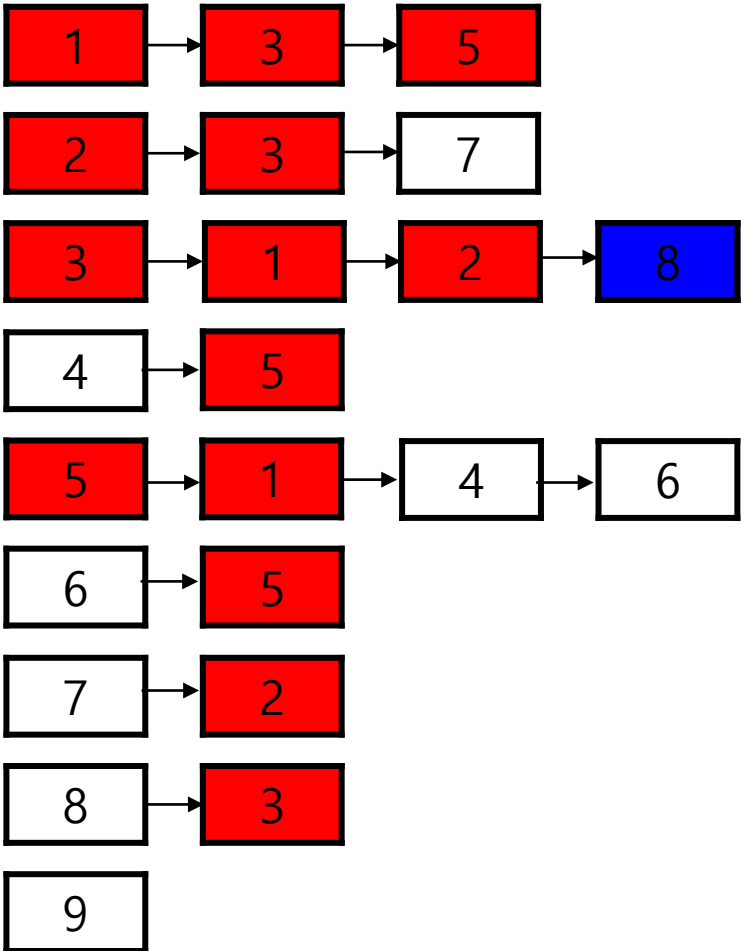
Breadth First Search(BFS)

start location : 1
finish location : 6

 : current position
 : visited position





queue

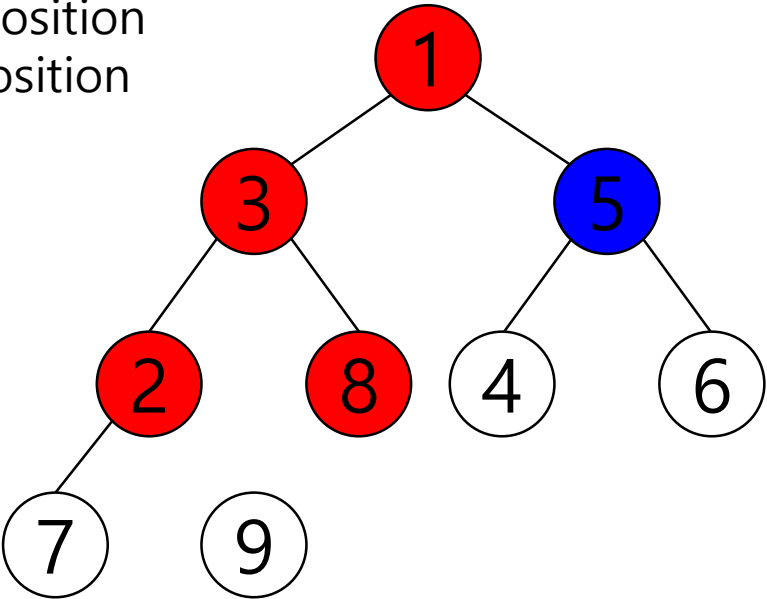




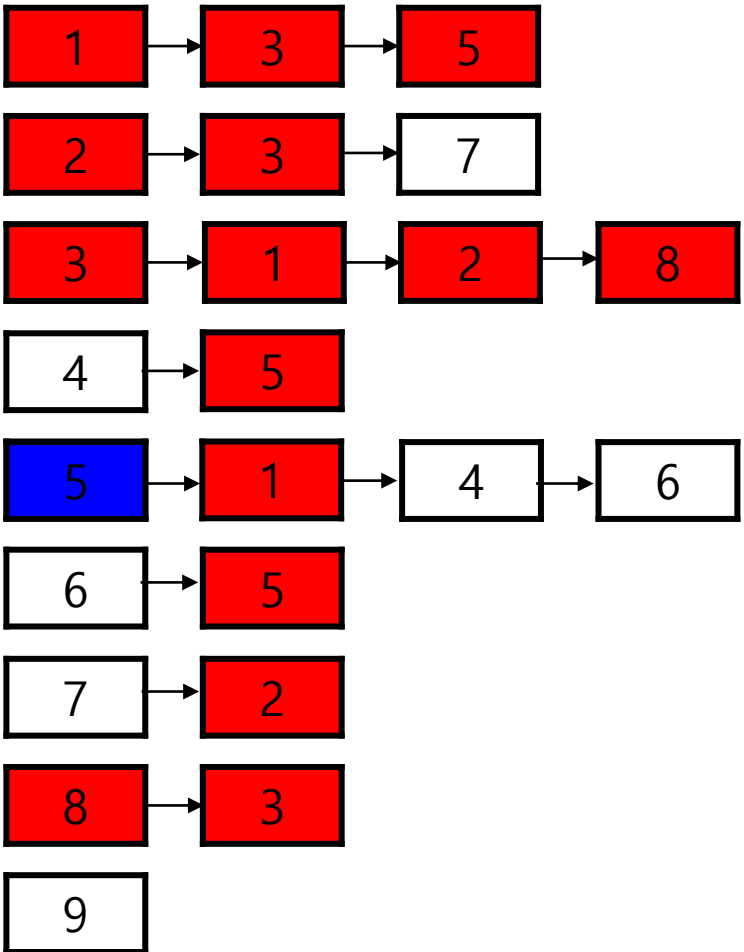
Breadth First Search(BFS)

start location : 1
finish location : 6

 : current position
 : visited position





queue

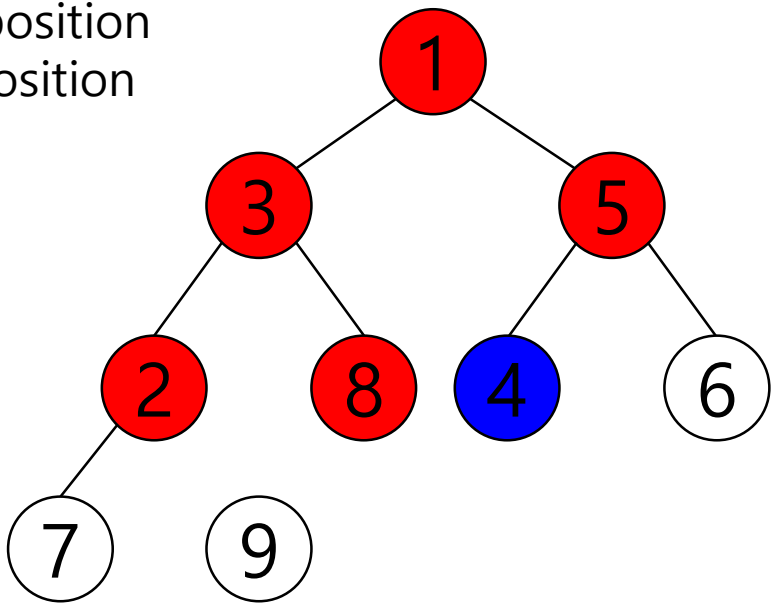




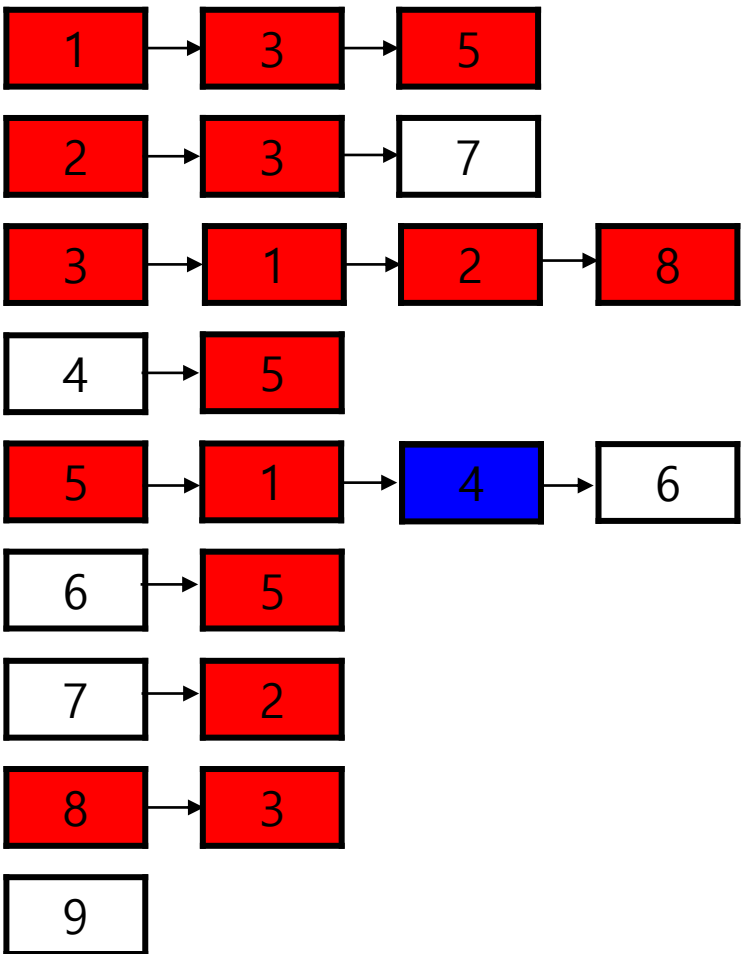
Breadth First Search(BFS)

start location : 1
finish location : 6

 : current position
 : visited position





queue

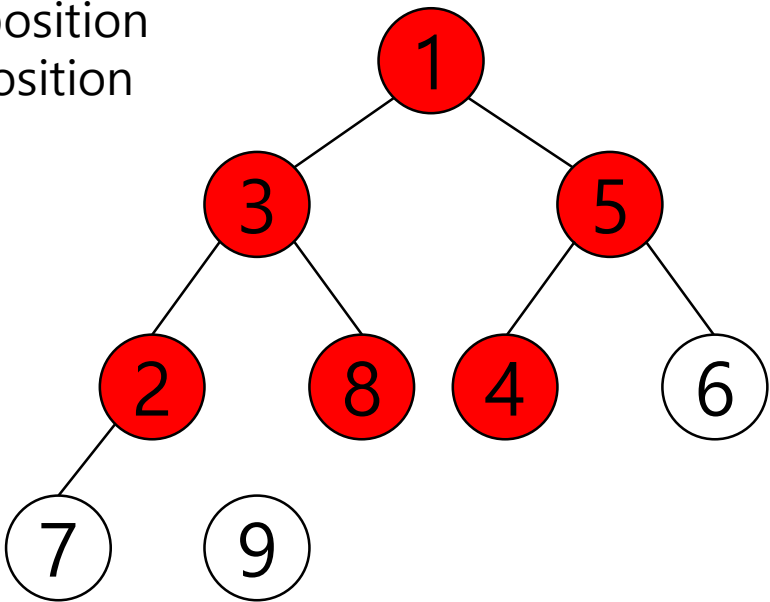




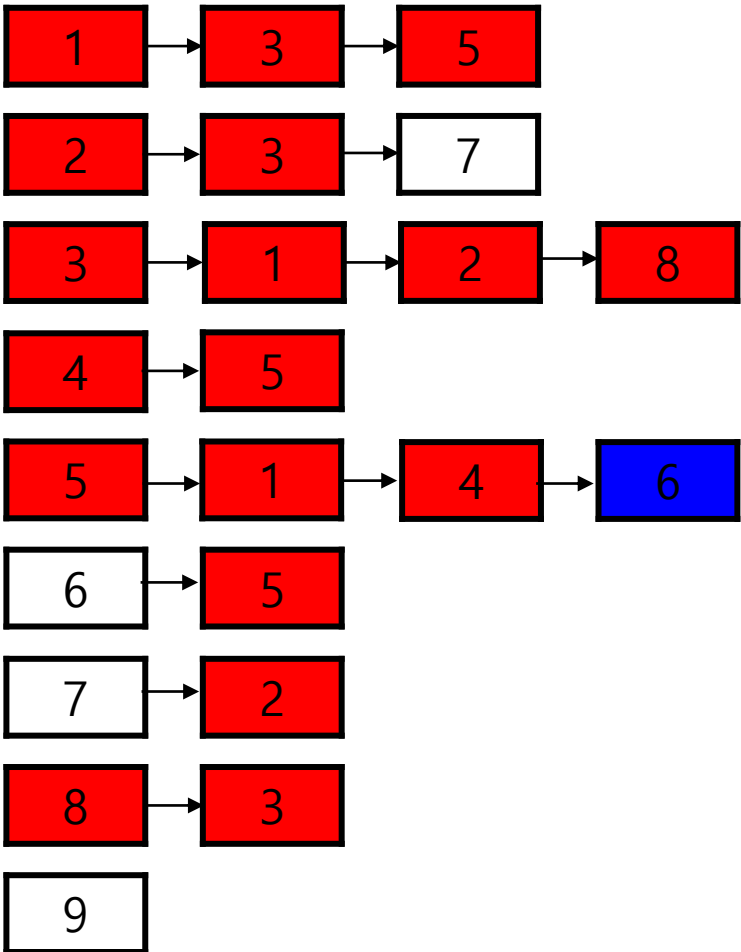
Breadth First Search(BFS)

start location : 1
finish location : 6

 : current position
 : visited position



queue





인접리스트와 큐를 이용한 BFS 구현

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_V 101

typedef struct node* nptr;
typedef struct node {
    int vertex;
    nptr link;
}NODE;

int Queue[MAX_V * 2 + 1], qfront, qback;
nptr adj_list[MAX_V];
bool visited[MAX_V];
```

```
void bfs(int start) {
    int curr;
    nptr w;
    qfront = 0, qback = -1;
    printf("%d ", start);
    visited[start] = true;
    Queue[++qback] = start;
    while (qfront <= qback) {
        curr = Queue[qfront];
        qfront++;
        for(w = adj_list[curr]; w; w = w->link)
            if (!visited[w->vertex]) {
                printf("%d ", w->vertex);
                Queue[++qback] = w->vertex;
                visited[w->vertex] = true;
            }
    }
}
```



vector도 쓰고, queue도 써봅시다.

```
#include <vector>
#include <queue>
using namespace std;
#define MAX_V 101

vector<int> adj_list[MAX_V];
bool visited[MAX_V];
queue<int> q;

void bfs(int start) {
    int curr;
    visited[start] = true;
    printf("%d ", start);
    q.push(start);
    while (!q.empty()) {
        for (int i = 0; i < q.size(); i++) {
            curr = q.front(); q.pop();
            for (int j = 0; j < adj_list[curr].size(); j++) {
                int next = adj_list[curr][j];
                if (!visited[next]) {
                    printf("%d ", next);
                    q.push(next);
                    visited[next] = true;
                }
            }
        }
    }
}
```




이거도 길쥬?

```
void bfs(int start) {  
    int curr;  
    visited[start] = true;  
    printf("%d ", start);  
    q.push(start);  
    while (!q.empty()) {  
        for (int i = 0; i < q.size(); i++) {  
            curr = q.front(); q.pop();  
            for (int next : adj_list[curr]) {  
                if (!visited[next]) {  
                    printf("%d ", next);  
                    q.push(next);  
                    visited[next] = true;  
                }  
            }  
        }  
    }  
}
```



시간복잡도

- 탐색 과정에서 최악의 경우 방문하게될 노드의 개수 : $|V|$ 개
- 해당 노드에 방문하기 위해 거쳐간 간선의 개수 : $|E|$ 개

$$O(|V| + |E|)$$



시간복잡도

- 인접 행렬을 사용했다면?
- 연결 여부 상관없이 썩다 검색을 하므로
 $O(|V|^2)$



Problem set

- 1260 DFS와 BFS
- 1012 유기농 배추
- 2583 영역 구하기
- 11724 연결 요소의 개수
- 13023 ABCDE
- 2178 미로 탐색
- 2644 촌수계산
- 5427 불
- 2206 벽 부수고 이동하기



#1012 유기농 배추

- 각 정점에 해당하는 것은 좌표이고,
- 간선이라고 할만한 것은 상, 하, 좌, 우로 이동하는 것이므로
인접리스트를 굳이 만들어서 표현할 필요 없이 바로 탐색을 돌려도 되겠다.



#11724 연결요소의 개수

- 방향 없는 그래프임을 유의
- 영역 구하기와는 다르게, 몇 개의 구역으로 나누어져 있는지 묻는 문제



#5427 불

- 내가 움직이면 동시에 불도 확장된다.
- 불은 인접한 4칸에 모두 확장된다.
- 불은 뜨겁다.

```
void bfs() {
    q.push({ SX, SY });
    while(!q.empty()) {
        int Size = q.size();
        int poolSize = wat.size();
        for(int i = 0; i < poolSize; i++) {
            [REDACTED]
            for(int j = 0; j < 4; j++) {
                [REDACTED]
            }
        }

        for(int i = 0; i < Size; i++) {
            currX = q.front().x; currY = q.front().y;
            [REDACTED]
            for(int j = 0; j < 4; j++) {
                int nextX = currX + addx[j], nextY = currY + addy[j];
                [REDACTED]
            }
        }
        ans++;
    }
}
```



#2206 벽 부술고 이동하기

- 벽은 한번만 부술 수 있다.
- 즉, (x,y) 를 도달하는데 벽을 부술고 도달했는지,
- 벽을 안부술고 도달했는지를 체크할 수 있다.

```
bool vis[MXV][MXV][2];
```