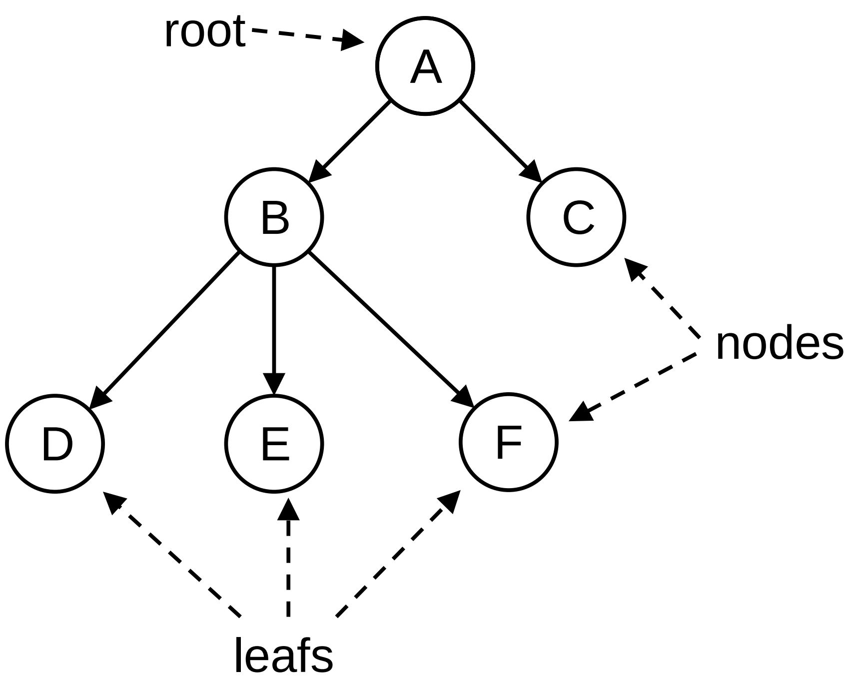
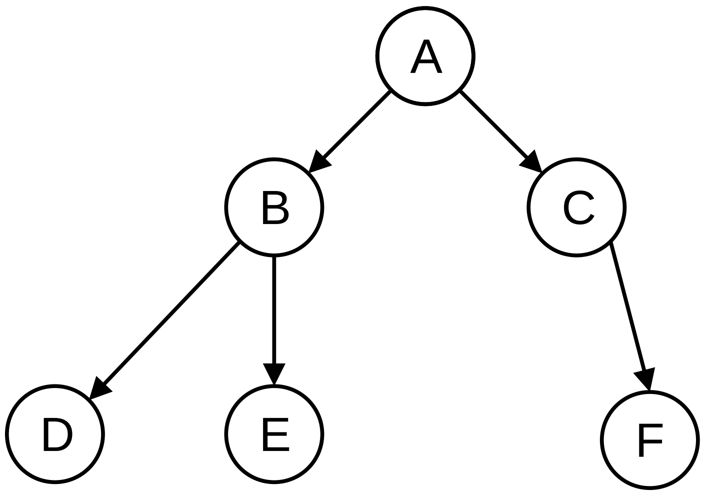
Trees and graphs

Simeon Monov

Trees



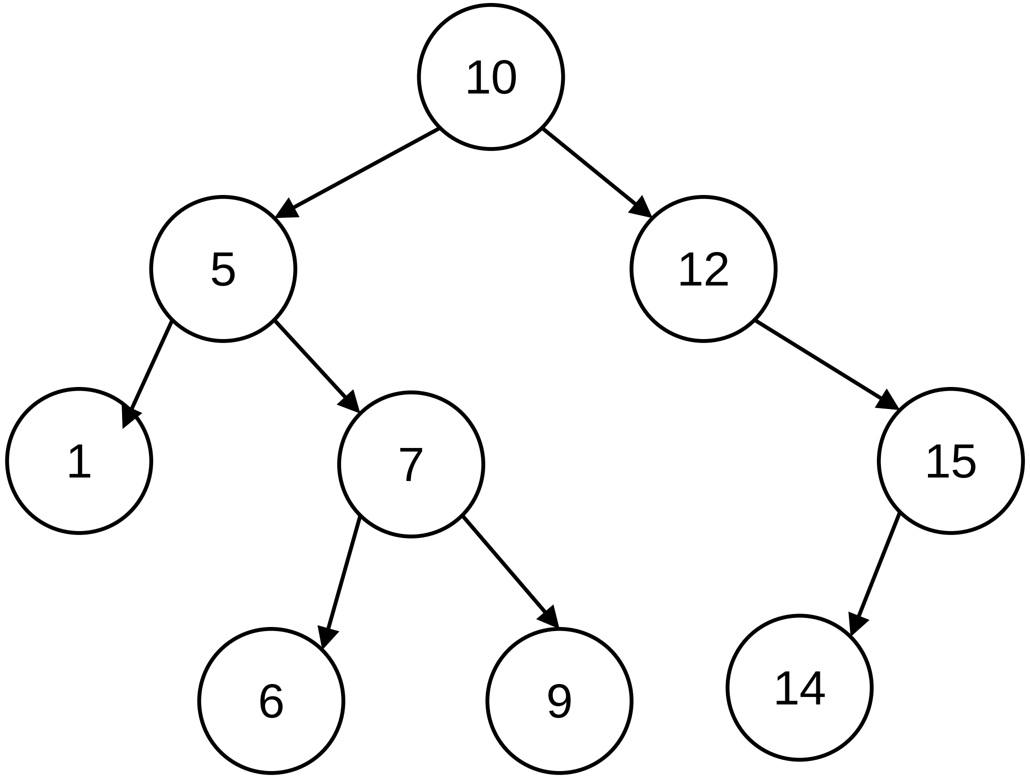
Tree



Binary tree

Binary search tree

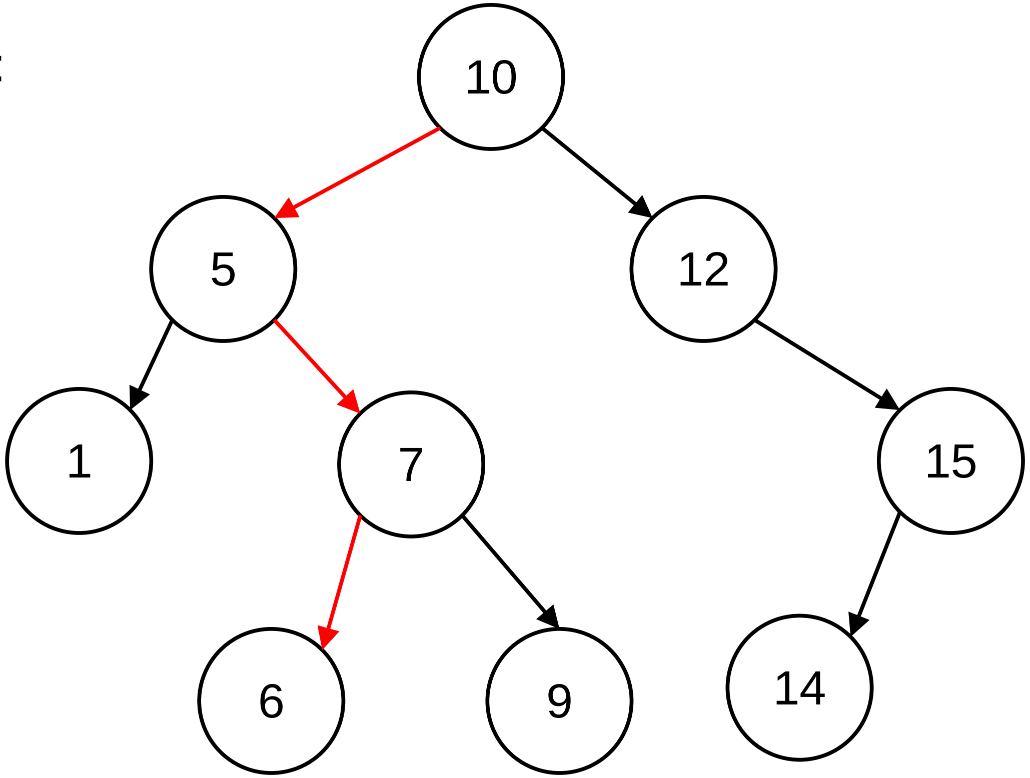
- Binary search tree contains nodes with unique keys
- Left subtree contains nodes with keys lesser than the node’s key
- Right subtree contains nodes with keys greater than the node’s key.
- Left and right subtree are binary search trees.



Binary search tree – search by key

search(t, key):
1) If key < key(t) search in the left subtree: search(left(t), key)
2) If key > key(t) search in the right subtree: search(right(t), key)
3) If key == key(t), we found the node

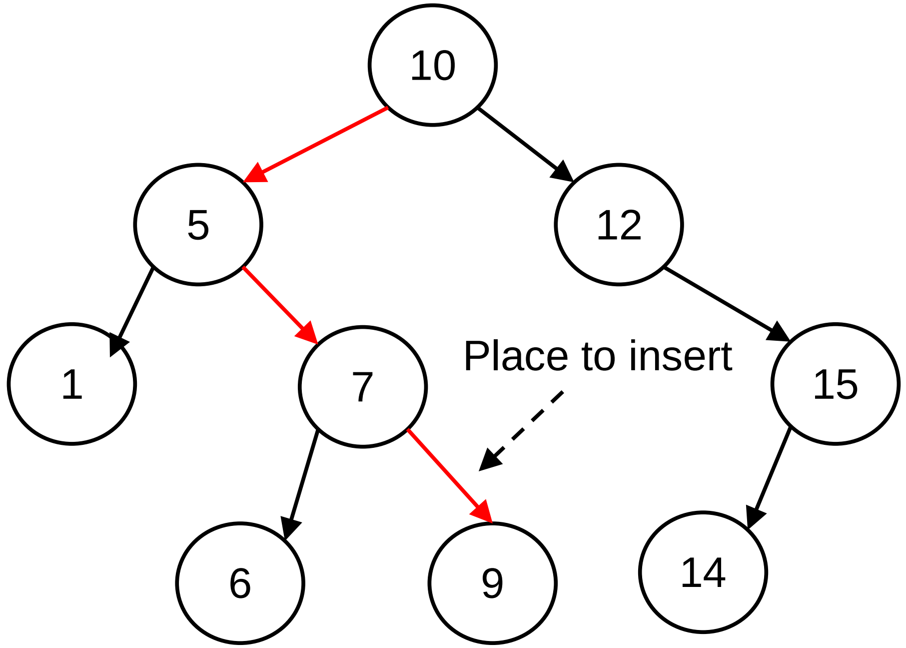
Example for key=6:



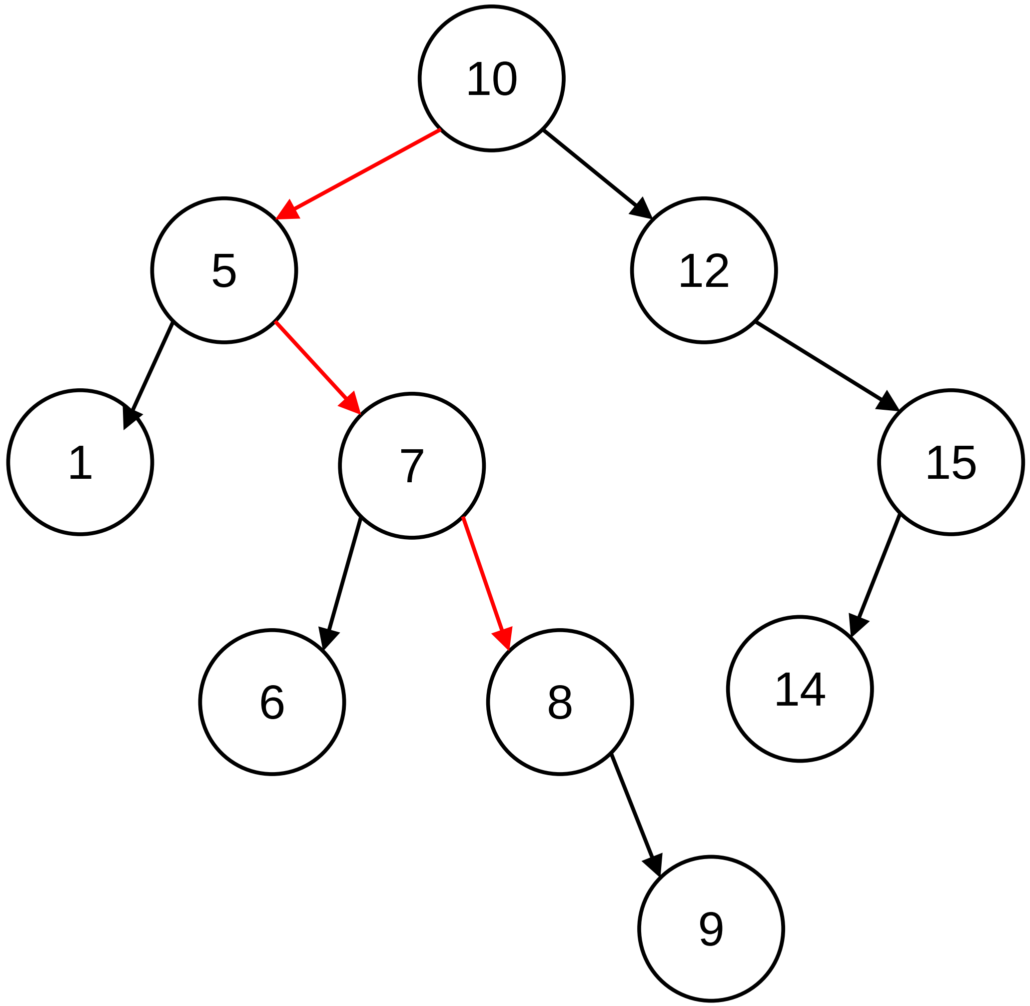
Binary search tree – insert node

insert(t, node):

1) If node == null, we found the place to insert node
2) If key(node) < key(t): insert(left(t), node)
3) If key(node) > key(t): insert(right(t), node)
4) If key(node) == key(t), node with such key already exists. We either return nothing or throw an error



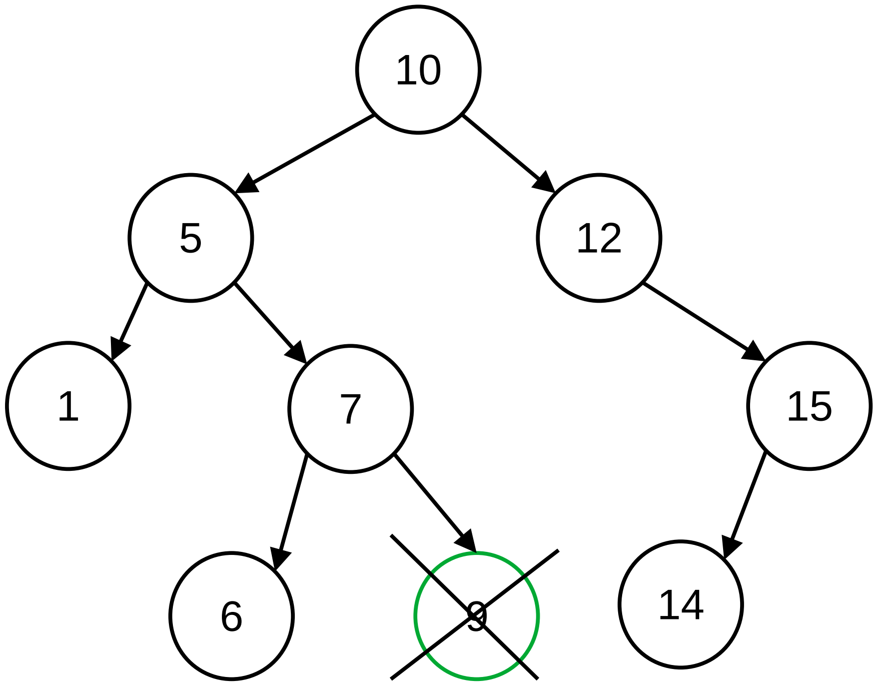
Example for new node(8)



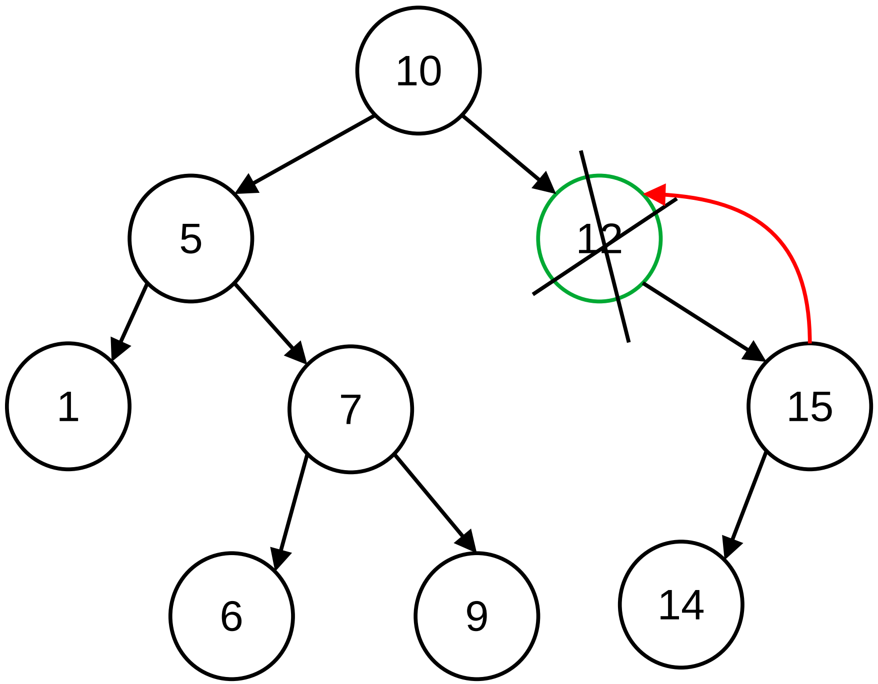
Binary search tree – delete node

First, we need to locate/find the node in the tree.

If the node is a leaf, e.g. key=9, delete the node



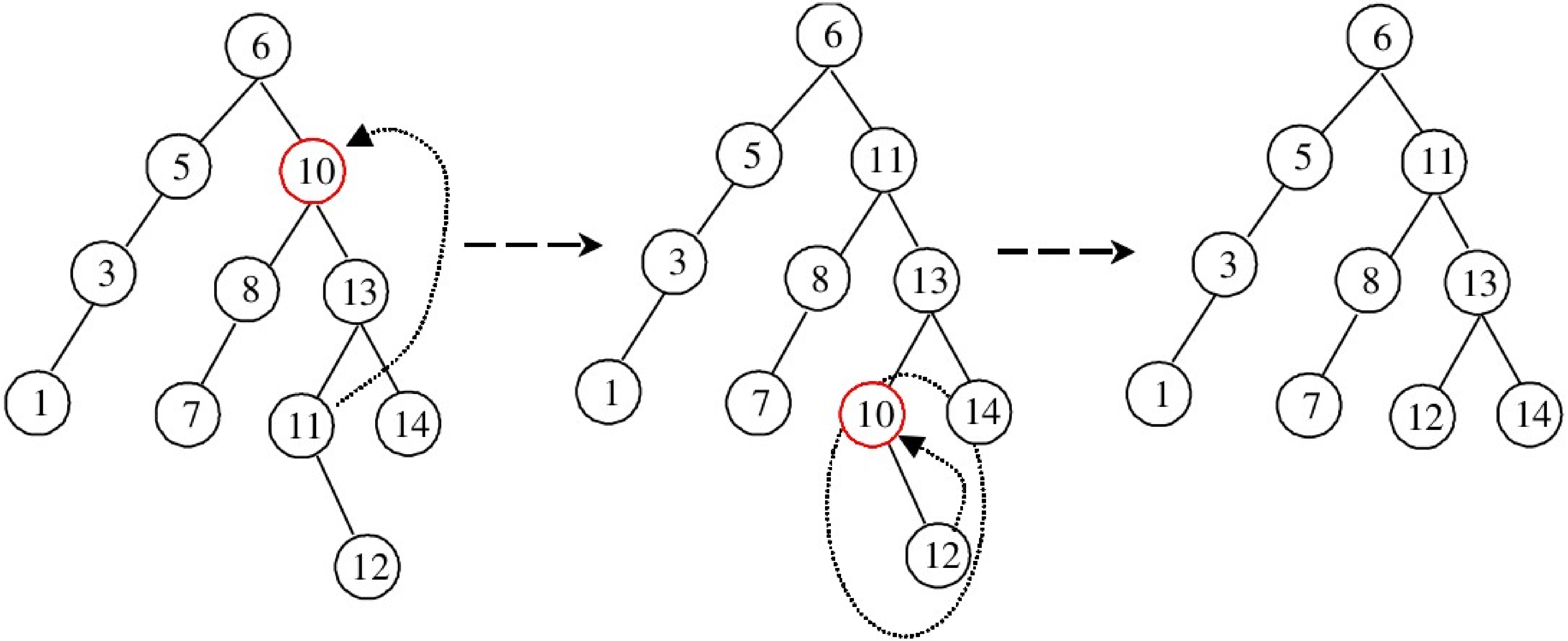
If the node has only one subtree (left or right), e.g. key=12, replace the node with the root of this subtree



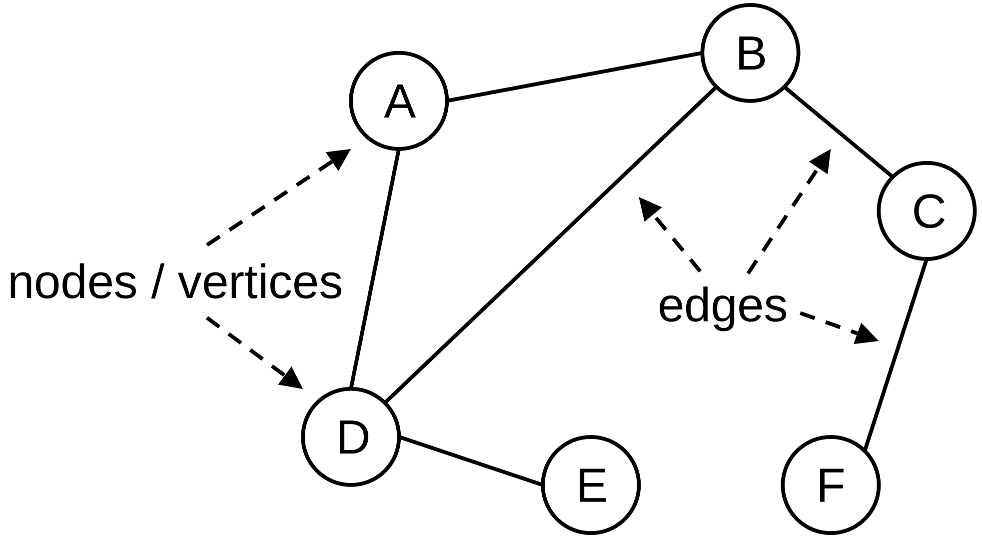
Binary search tree – delete node

If the node has both left and right subtree, e.g. key=10:

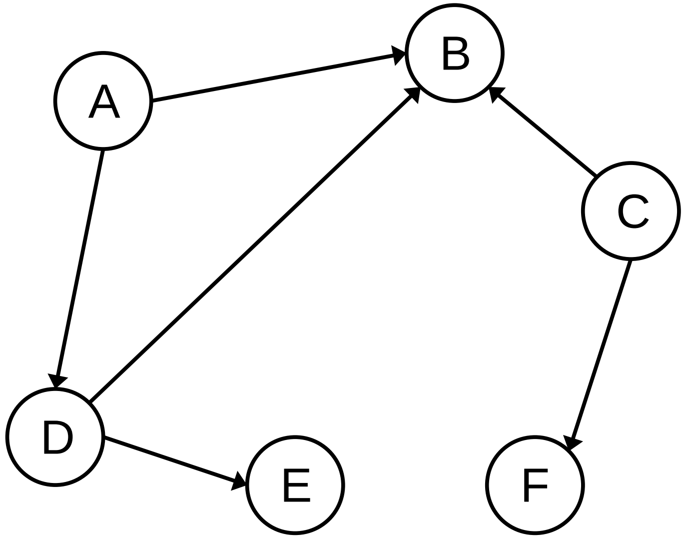
1) Find the smallest key in the right subtree (most left bottom node)
2) Swap the nodes
3) Remove the search node (after swap it will become possible to delete the node by the previous rules



Graphs

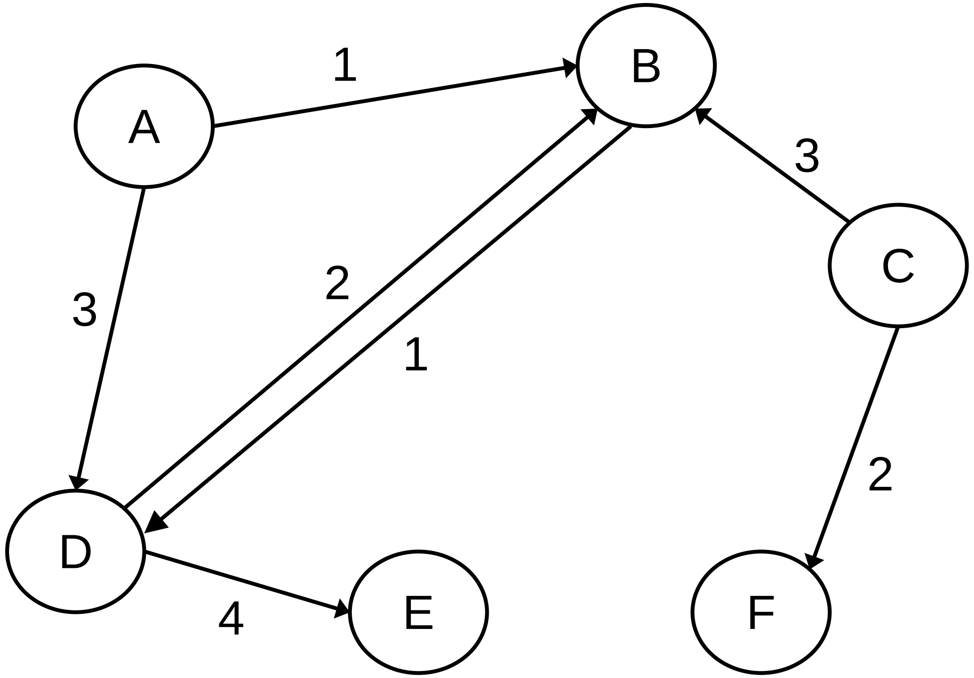


Undirected graph



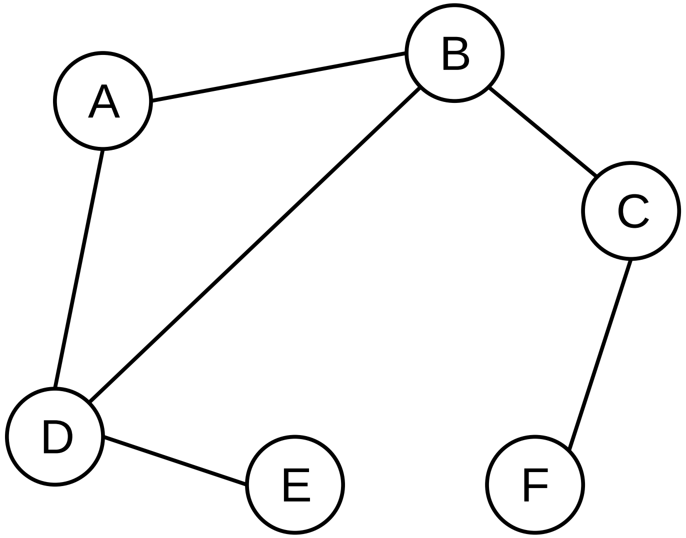
Directed graph

Graphs – weighted graph



Weighted graph

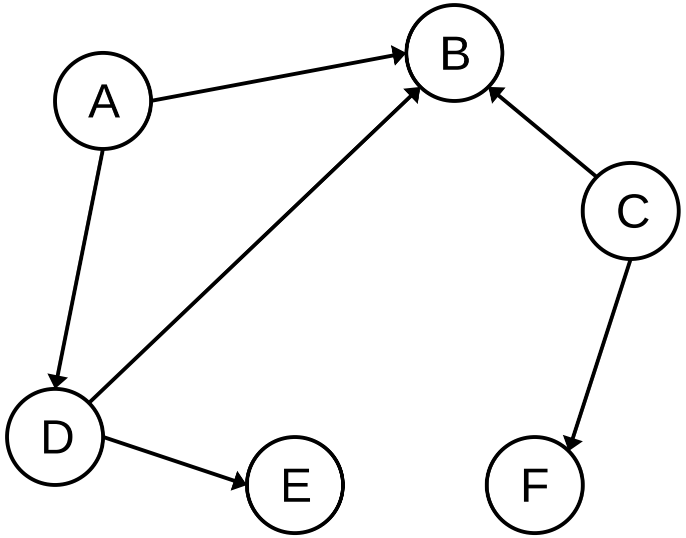
Adjacency matrix representation – undirected graph



Undirected graph

	A	B	C	D	E	F
A	0	1	0	1	0	0
B	1	0	1	1	0	0
C	0	1	0	0	0	1
D	1	1	0	0	1	0
E	0	0	0	1	0	0
F	0	0	1	0	0	0

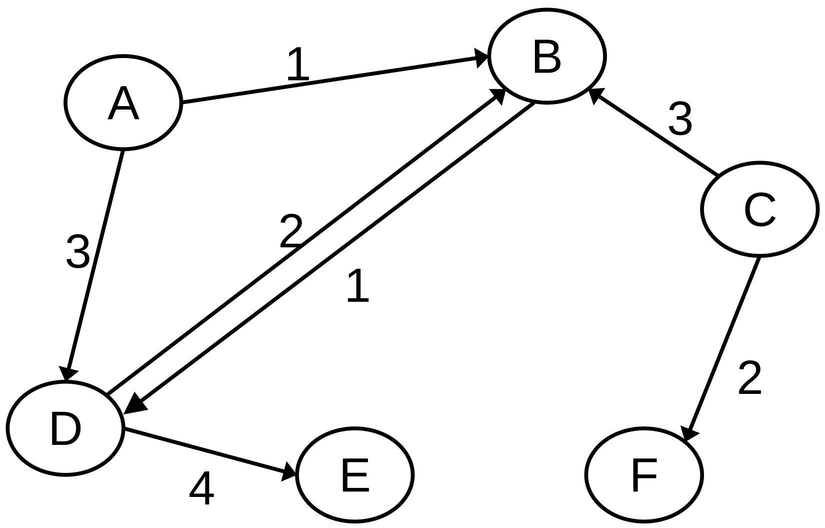
Adjacency matrix representation – directed graph



Directed graph

	A	B	C	D	E	F
A	0	1	0	1	0	0
B	0	0	0	0	0	0
C	0	1	0	0	0	1
D	0	1	0	0	1	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

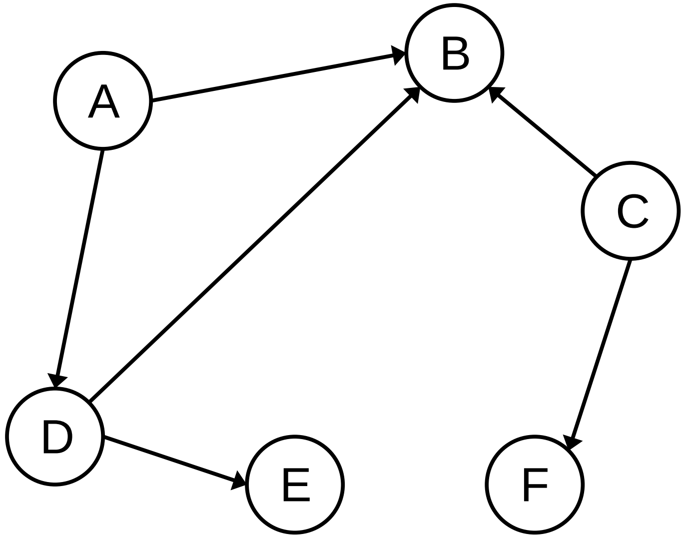
Adjacency matrix representation – weighted graph



Weighted graph

	A	B	C	D	E	F
A	0	1	0	3	0	0
B	0	0	0	1	0	0
C	0	3	0	0	0	2
D	0	2	0	0	4	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

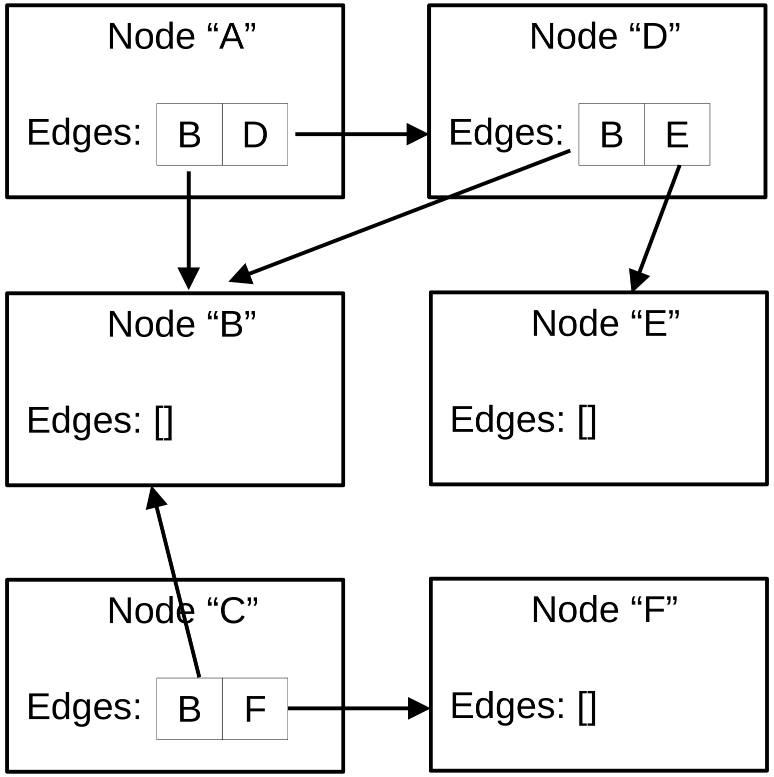
Adjacency list representation



Directed graph

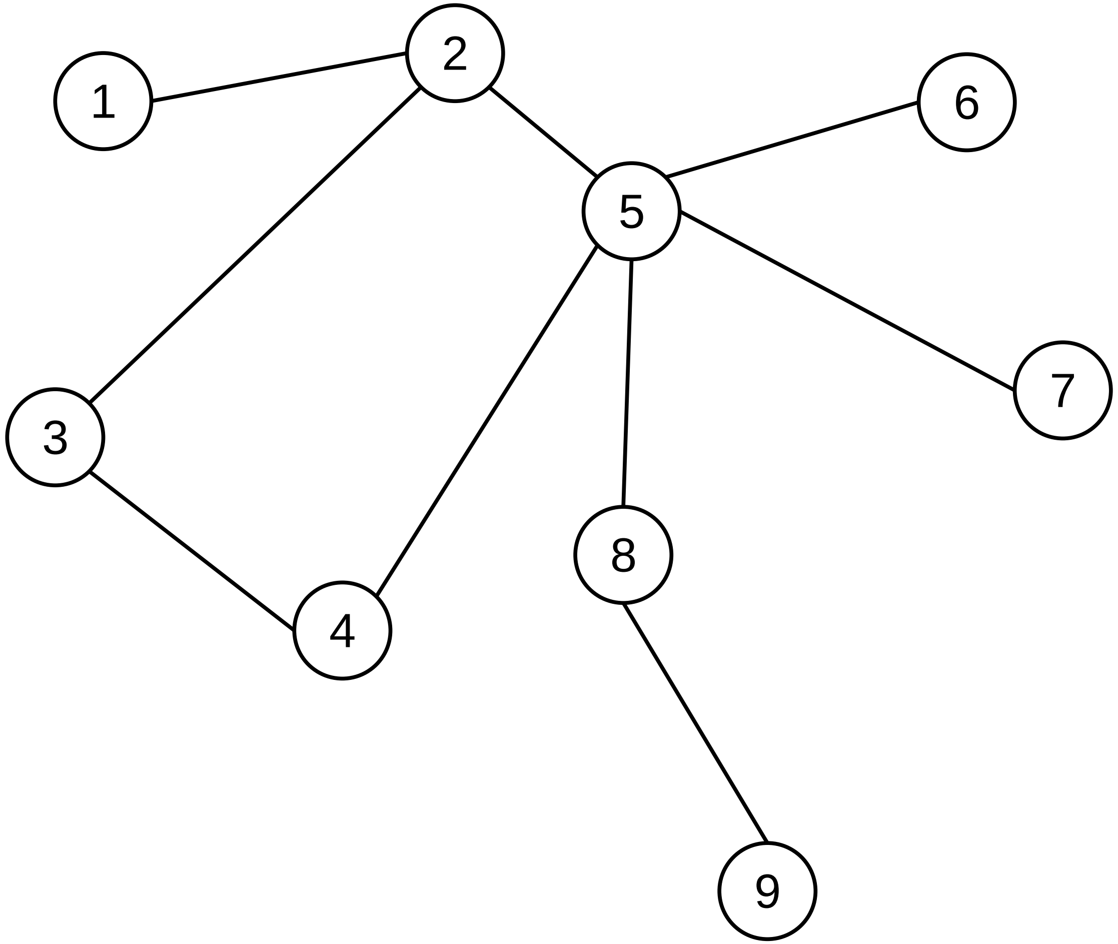
A - { B, D}
B - { }
C - { B, F }
D - { B, E }
E - { }
F - { }

Static lists



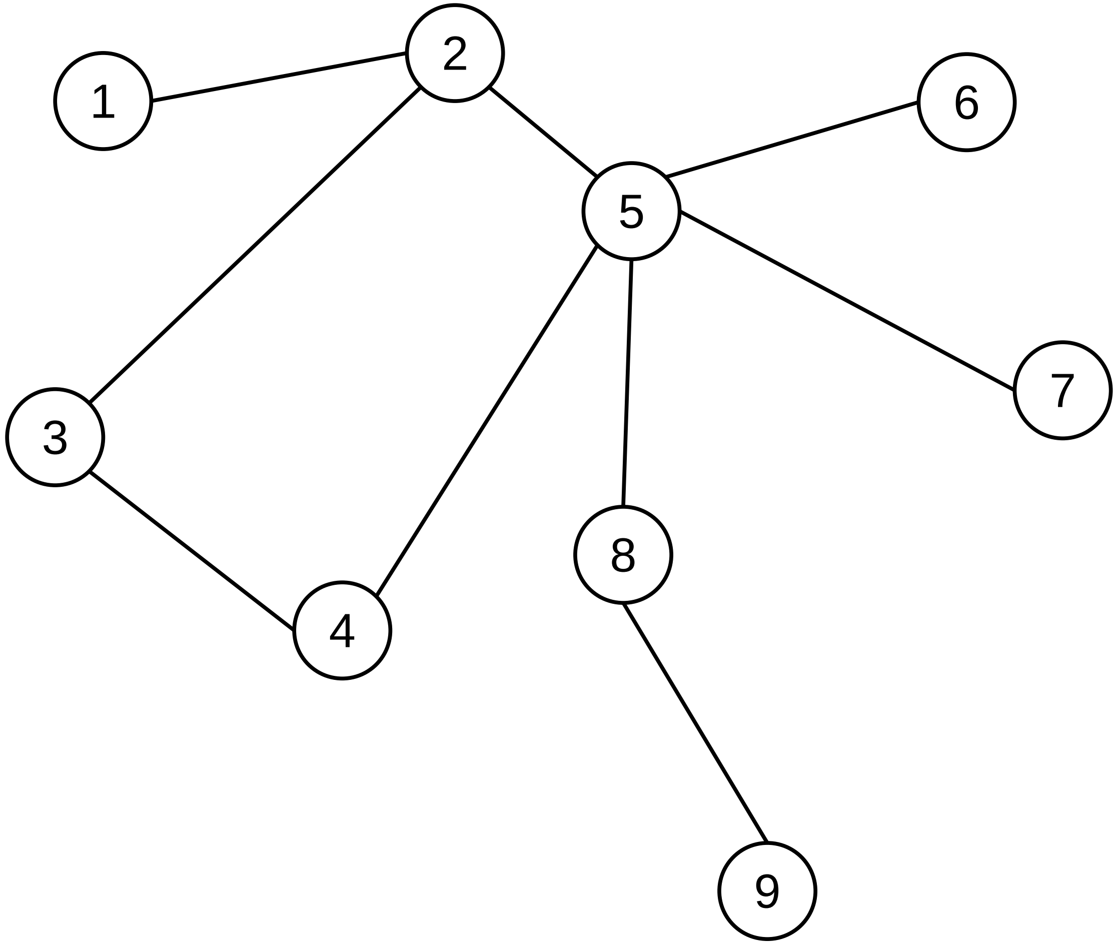
Dynamic implementation

Graph traversal – breadth first search (BFS)




```
BFS(startNode)
{
    Queue q = new Queue();
    q.Enqueue(startNode);
    <mark node startNode as used>
    while (q.Count()>0)
    {
        p = q.Dequeue();
        <extract all direct descendants of p>
        foreach node j in the descendats of p
        {
            If (<j is not used>)
            {
                q.Enqueue(j);
                <mark node j as used>
            }
        }
    }
}
```

Graph traversal – breadth first search (BFS) - example

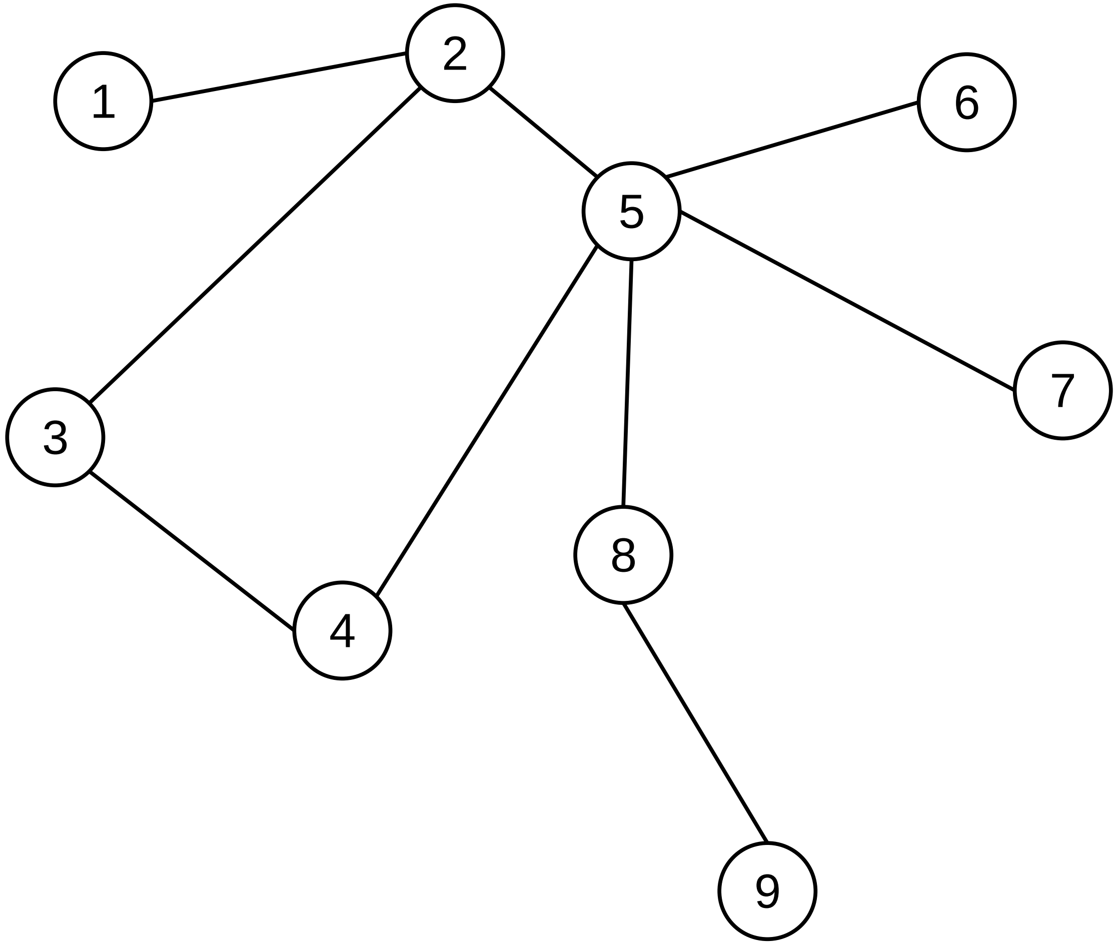


Starting from 1, the queue looks like this:

– [1]
1 – [2]
2 – [3, 5]
3 – [5, 4]
5 – [4, 6, 7, 8]
4 – [6, 7, 8]
6 – [7, 8]
7 – [8]
8 – [9]
9 – []

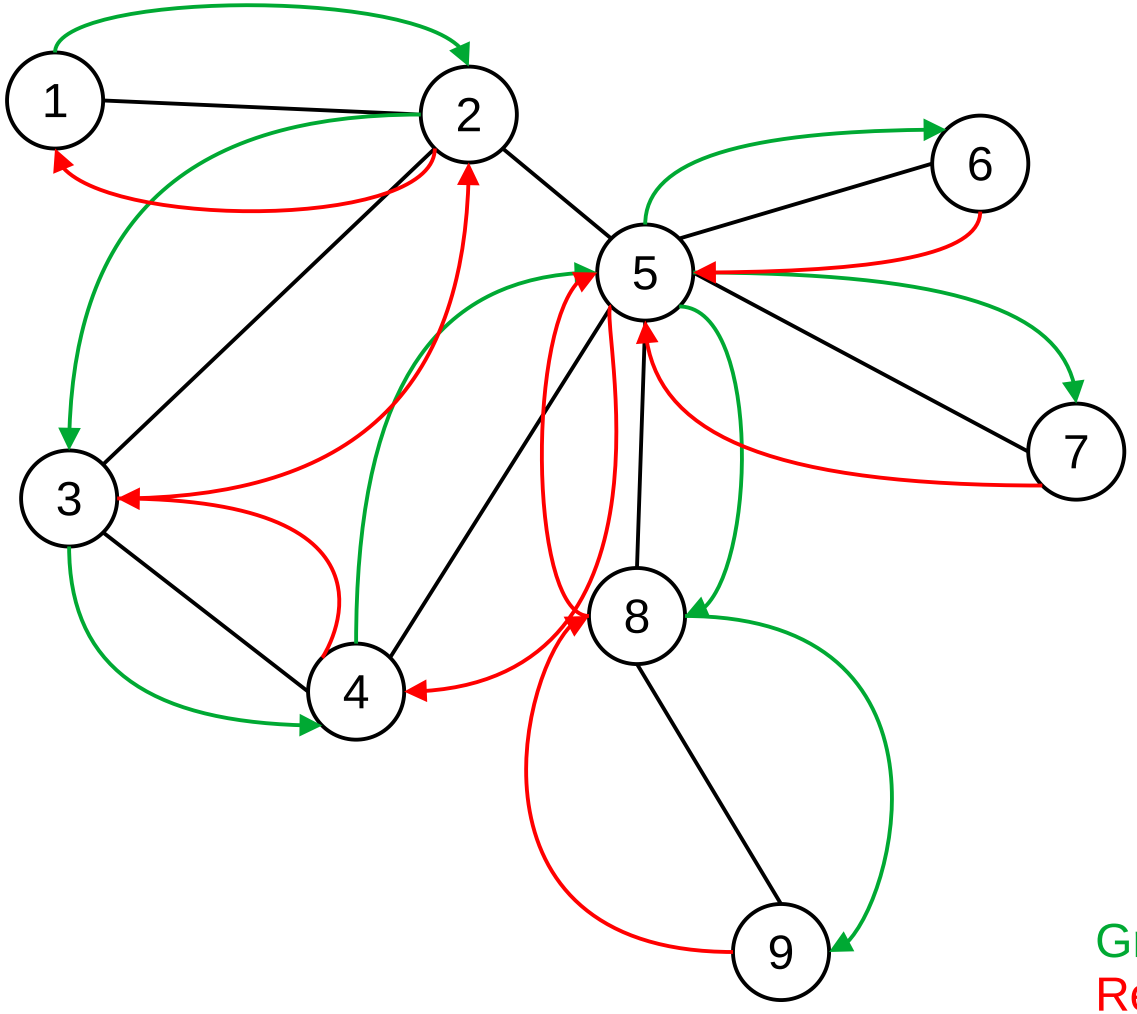
The resulting node traversal will be:
1, 2, 3, 5, 4, 6, 7, 8, 9

Graph traversal – depth first search (DFS)




```
DFS(node)
{
    <mark node as used>
    <extract all direct descendants of p>
    foreach node j in the descendats of p
    {
        if <j is not used>
        {
            DFS(j);
        }
    }
}
```

Graph traversal – breadth first search (BFS) - example



Starting from 1, the traversal will follow:

1 → 2 → 3 → 4 → 5 → 6 → 7 →8 → 9

Green arrows show going into the recursion
Red arrows show going back