Separate Compilation and Namespaces

Week 13

2017 Fall

Problem 1

Multiplied palindrome number

Multiplied palindrome number

- A number is called as a palindrome when you can read it equally from the front or from the behind.
- The largest palindrome number that can be made by multiplying two-digit numbers is 9009 (= 91×99).
- What is the largest palindrome number that can be made by multiplying N-digit numbers?

Multiplied palindrome number

- Input
  - N ($2 \leq N \leq 4$)
- Output
  - The largest palindrome number that can be made by multiplying two N-digit numbers

- Example of input and output

> 2

9009

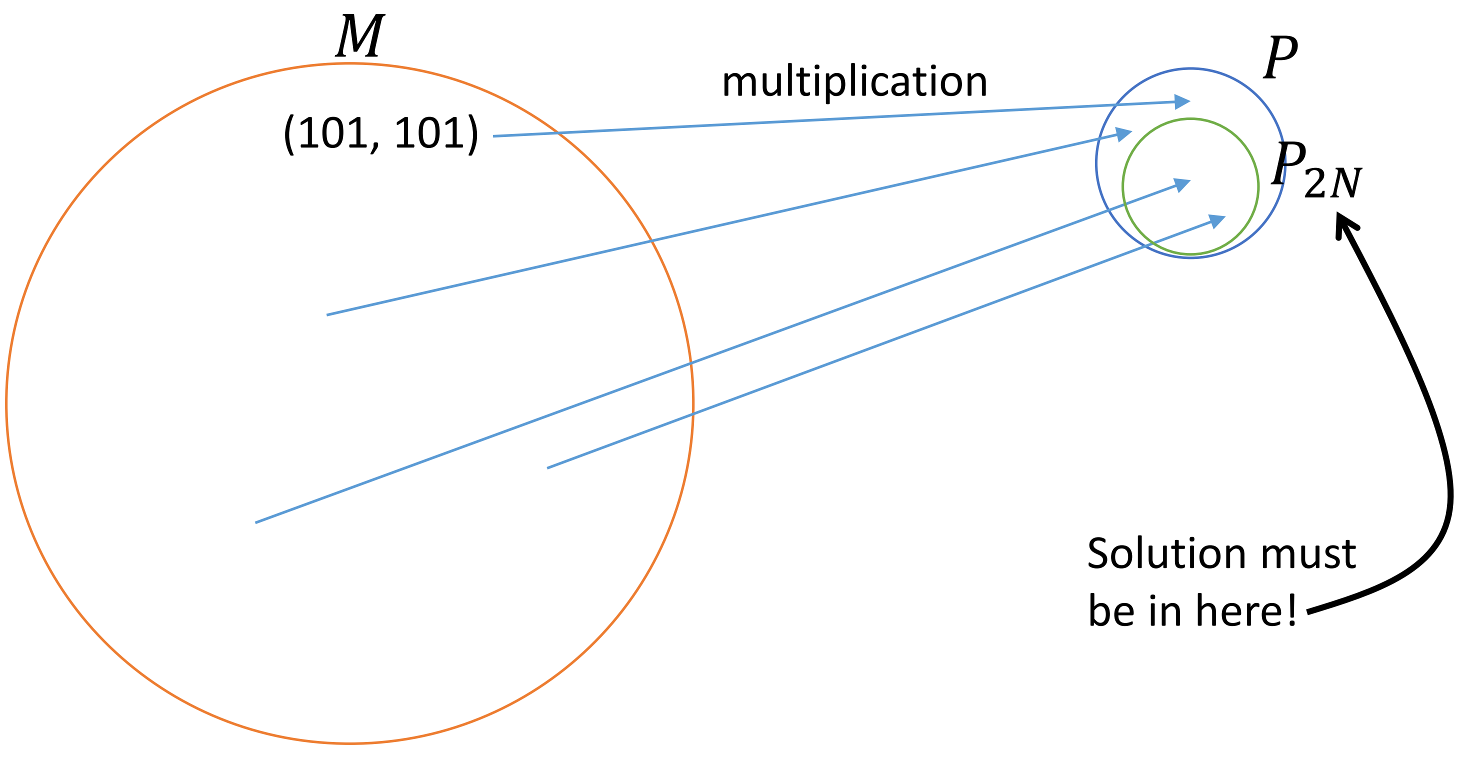
Brutal method

- For 2
  - 10*10, 10*11, 10*12, 11*11, ⋯, 99*98, 99*99
- For N
  - $(10^N - 10^{N-1}) * (10^N - 10^{N-1})$ number of operations

- Issue
  - Too much time to search the solution
  - Easy to fall into local optima

Some heuristics

$$M = \{10^{N-1} - 1 \ \ldots 10^{N}\} \times \{10^{N-1} - 1 \ \ldots 10^{N}\}$$
$$P = \{x | x \text{ is a palindrome}\}$$
$$P_{2N} = \{x | x \text{ is a 2N} - \text{digit palindrome}\}$$



Some heuristics

- (from the previous slide)
  Make 2N-digit palindrome first,
  then divide it to check if it is a solution
  - Avoid to fall into local optima
  - Reduce search space

and…

- Larger-to-smaller search
  - Prune unnecessary operations

Implementation

```cpp
int main(){
    int half = 1;
    int D;

    cin >> D;

    for(int i=0;i<D;i++){
        half *= 10;
    }
    half--;

    for(;half>0;){
        int palin = make_palindrome(half, D);
        if(is_prod_of_N_digits(palin, D)){
            cout << palin;
            break;
        }
        half--;
    }
}
```

Making 10^N-1
Ex) 999

If half = 976,
palin = 976679

If palin is product
of 2 N-digit numbers,
print it.

Implementation

```
int make_palindrome(int front, int digit){
    int rear = 0;

    for(int cfront = front;
        cfront >0;){
        rear *= 10;
        rear += cfront%10;
        cfront /= 10;
    }
    for(int i=0; i<digit; i++)
        front *= 10;

    return front+rear;
}
```

cfront = 976, rear = 0
cfront = 97, rear = 6
cfront = 9, rear = 67
cfront = 0, rear = 679

rear = 679

front = 976000

return 976679

Implementation

```
bool is_prod_of_N_digits(int n, int digit){
    int a = 1, b = 1;
    int q,r;
    for(int i=0; i<digit-1; i++)
        a*= 10;
    b = a*10;
    for(int i=a; i<b; i++){
        q = n/i;
        r = n%i;
        if(r==0 && a <= q && q< b){
            return true;
        }
    }
    return false;
}
```

$a = 10^{N-1}$

$b = 10^{N}$

i = N-digit number.

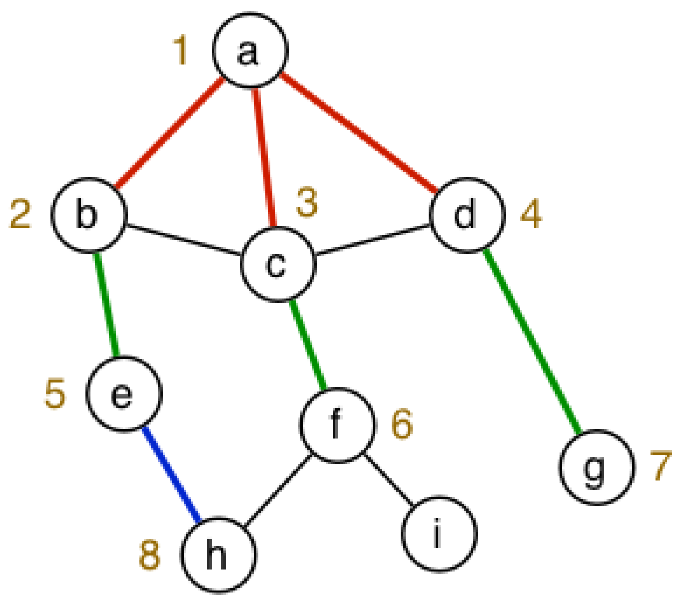
If n can be divided by i, and the quotient is N-digit, return true.

Problem 2

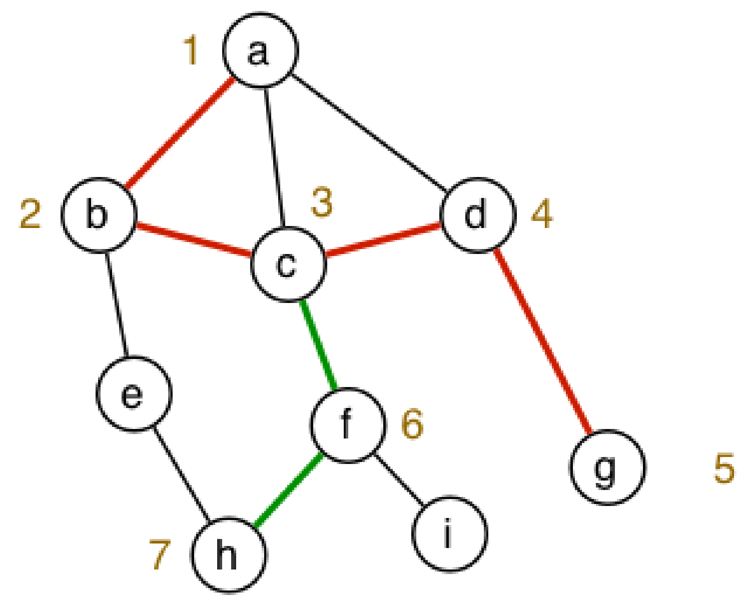
DFS v BFS

DFS vs BFS

- Search strategies for non-linear data structures
  - Depth-first-search(DFS)
  - Breadth-first-search(BFS)

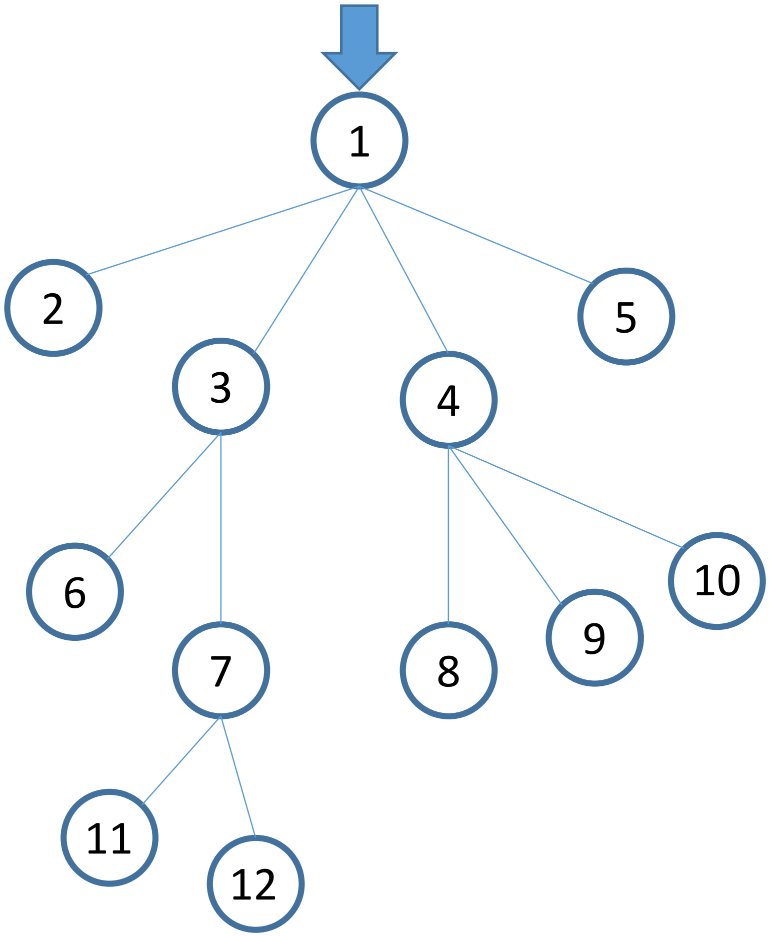


Breadth-first Search



Depth-first Search

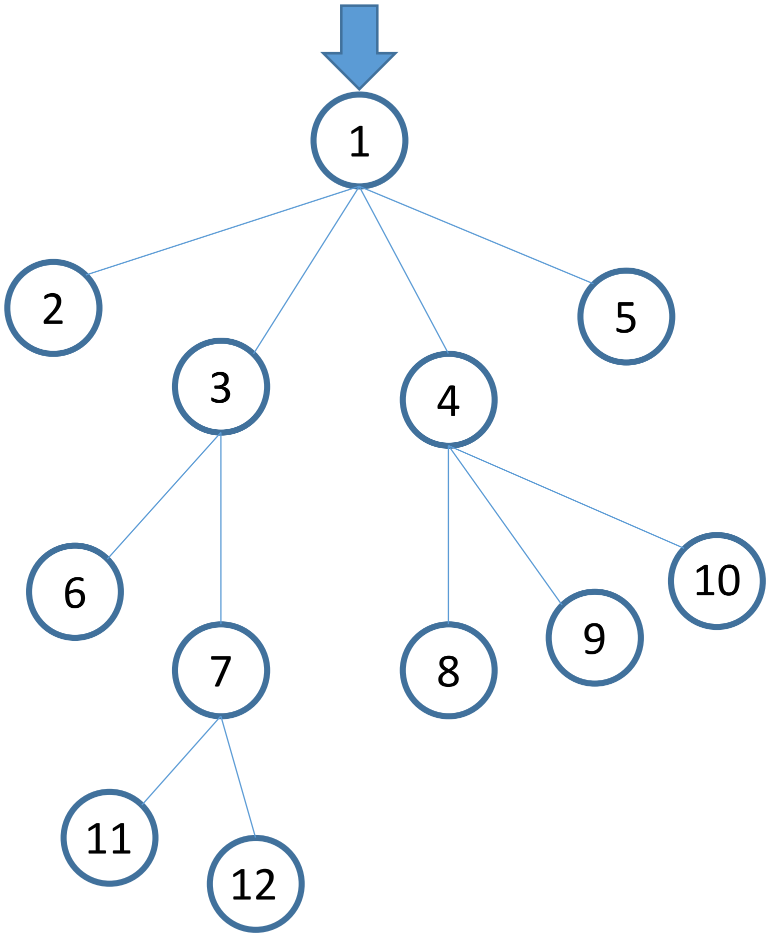
DFS with Stack



Stack

Root node can be in or not be in the stack at the beginning.
It is up to the implementation.

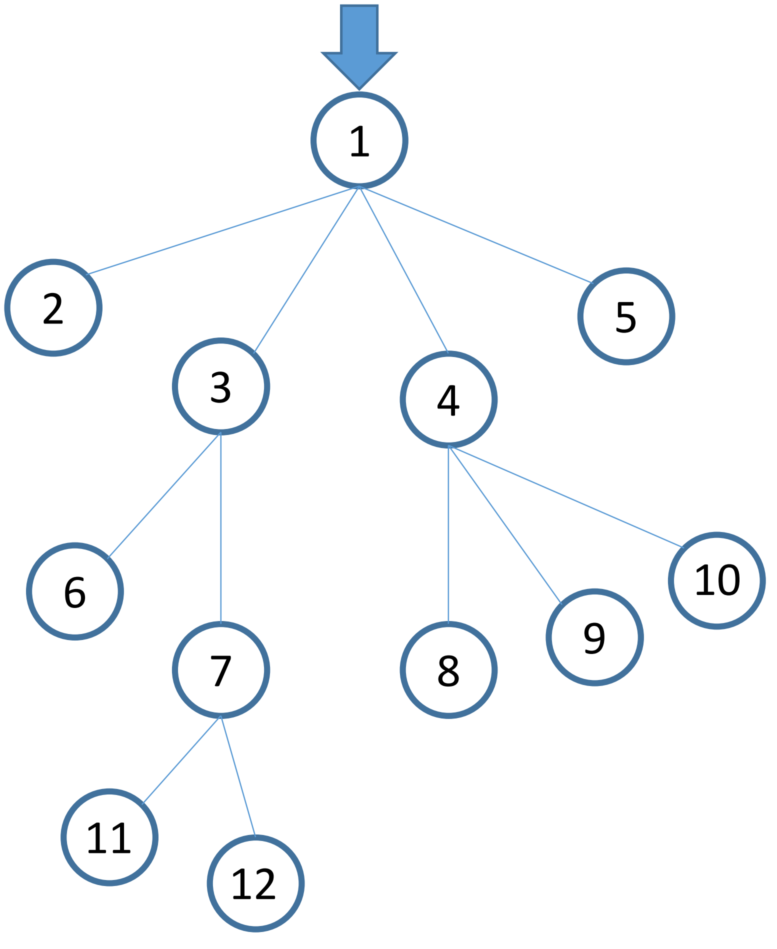
DFS with Stack



Stack

Push children of the now-seeing node to the stack, from the back to the front.

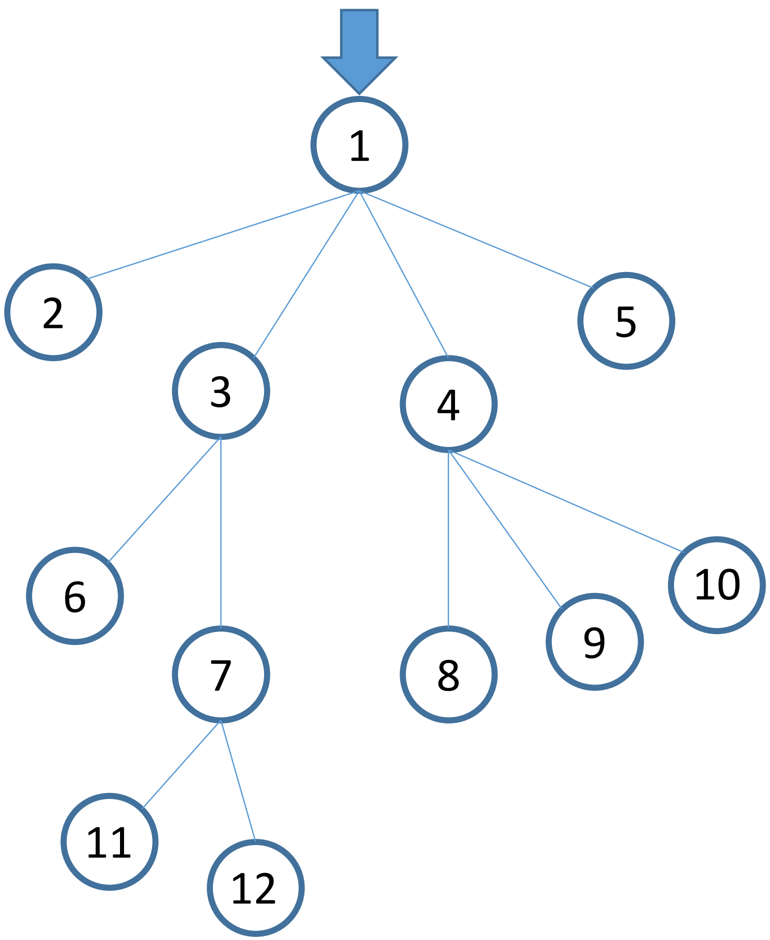
DFS with Stack

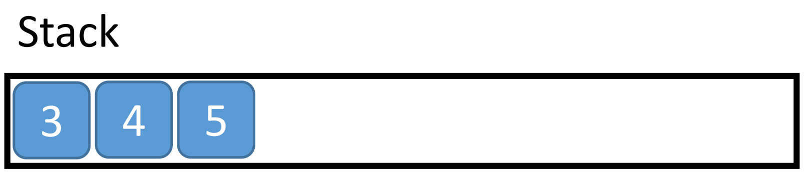


Stack

Push children of the now-seeing node to the stack, from the back to the front.

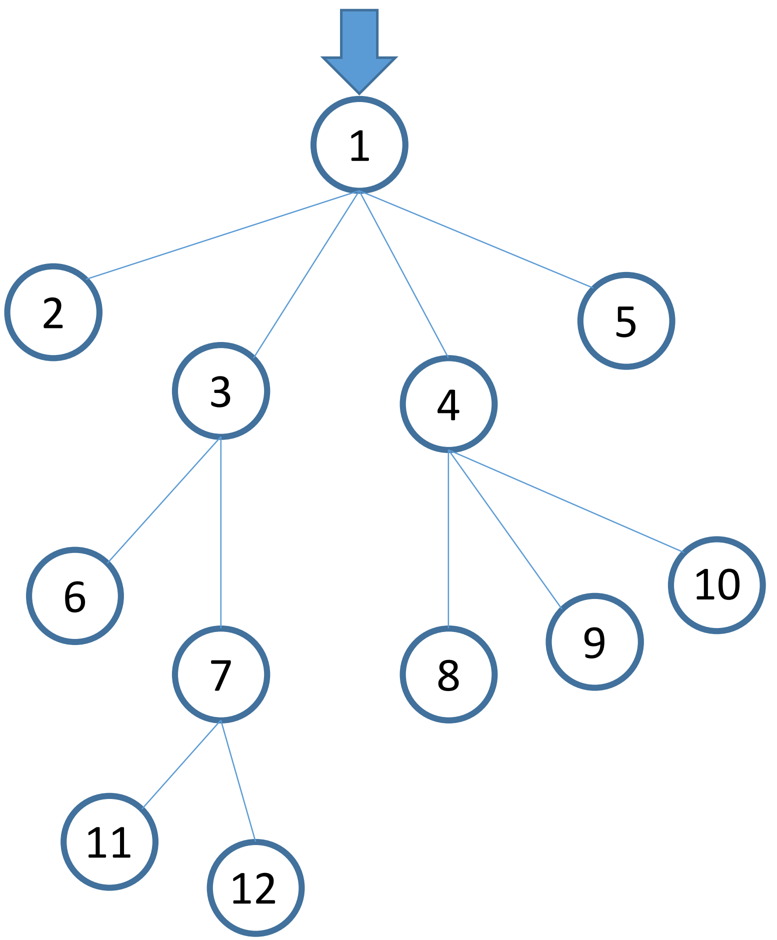
DFS with Stack





Push children of the now-seeing node to the stack, from the back to the front.

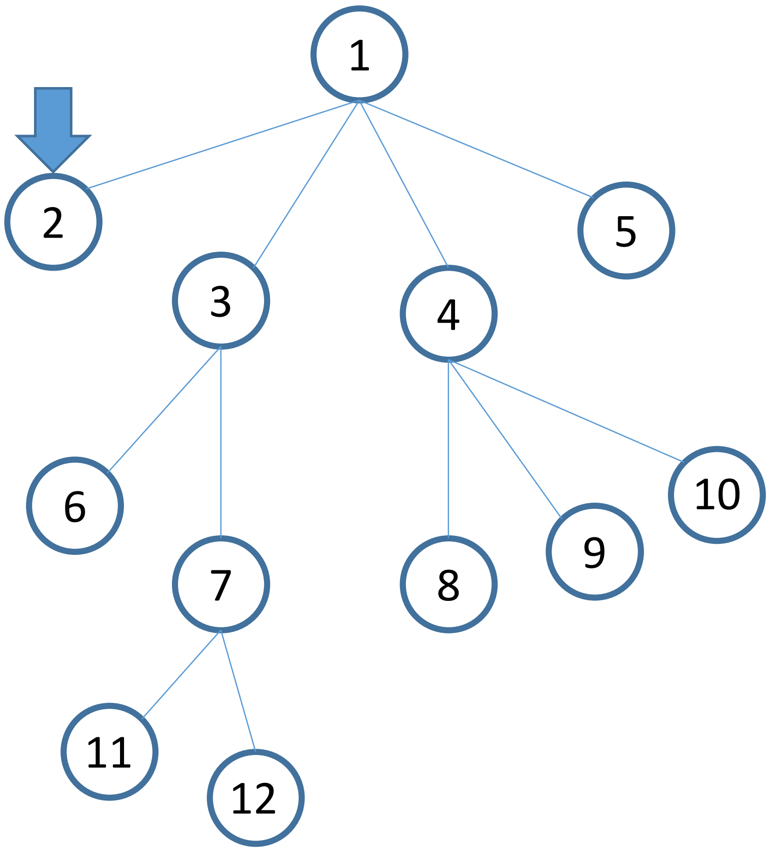
DFS with Stack

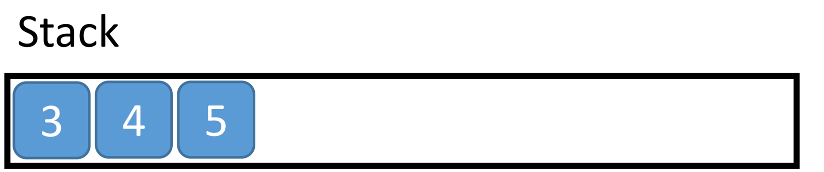


Stack

Push children of the now-seeing node to the stack, from the back to the front.

DFS with Stack

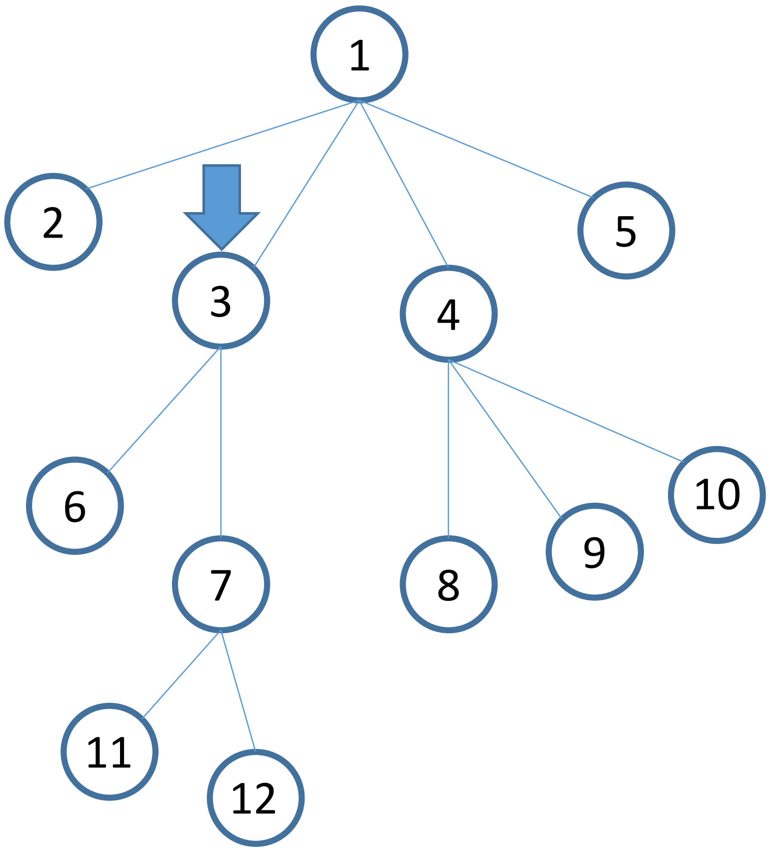




Pop one from the stack, and see it.
It has no children.

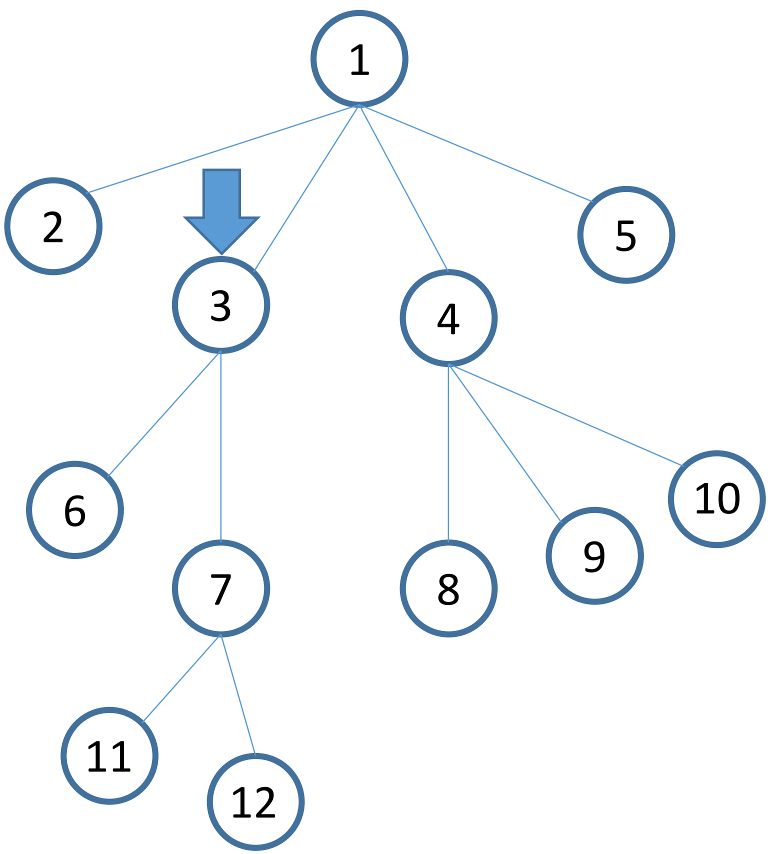
DFS with Stack

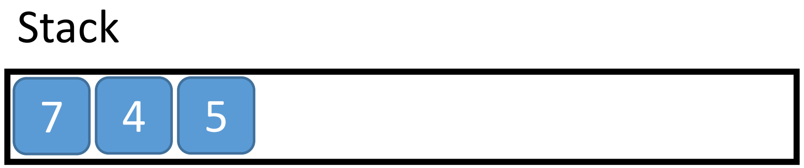
Stack



Pop one from the stack, and see it.
It has its children.

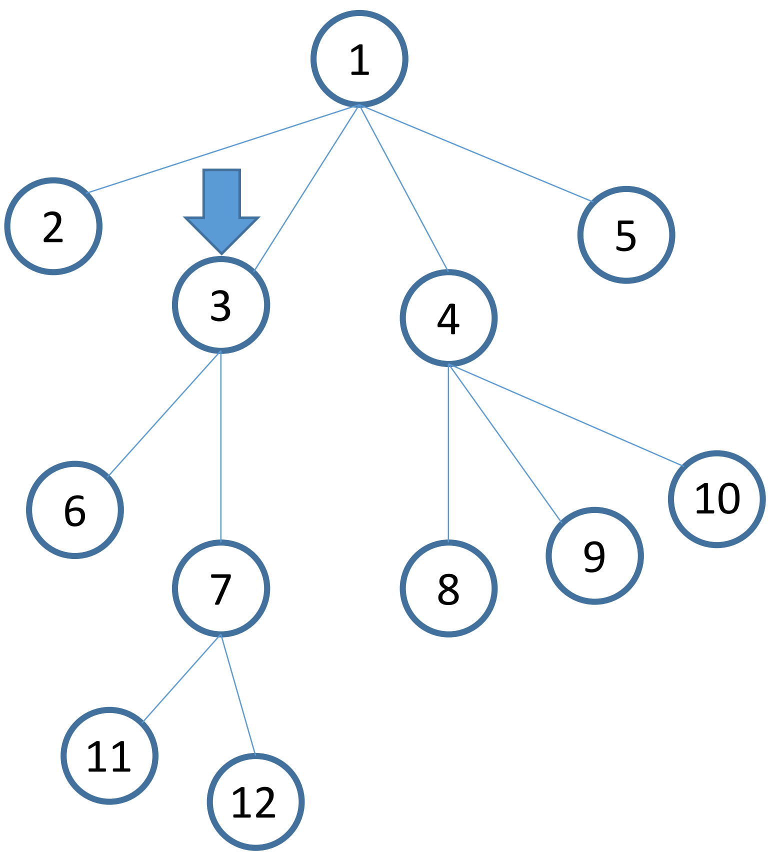
DFS with Stack





Push children of the now-seeing node to the stack, from the back to the front.

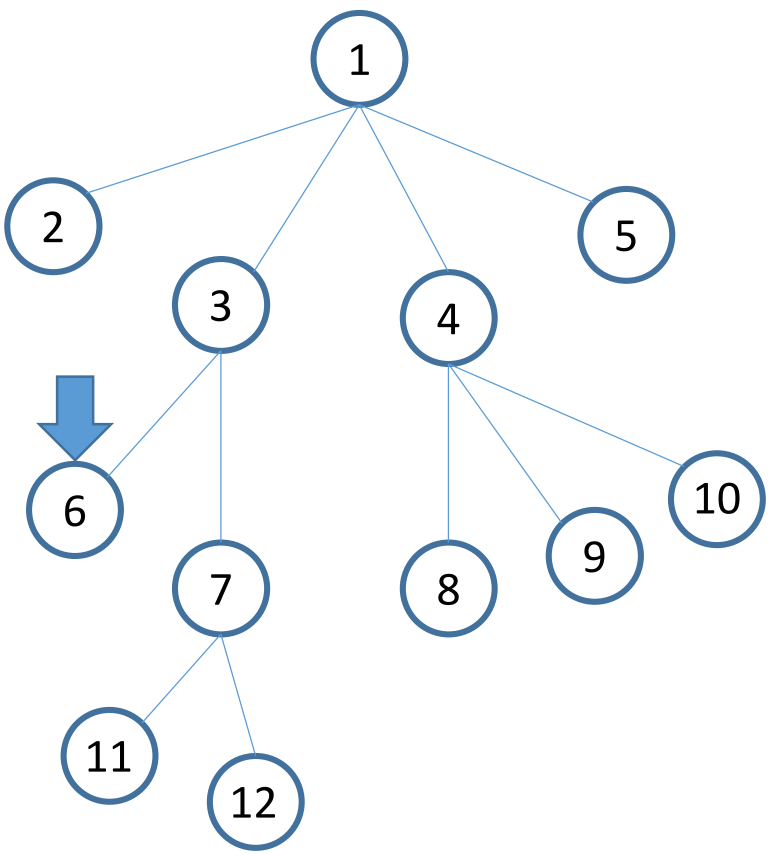
DFS with Stack



Stack

Push children of the now-seeing node to the stack, from the back to the front.

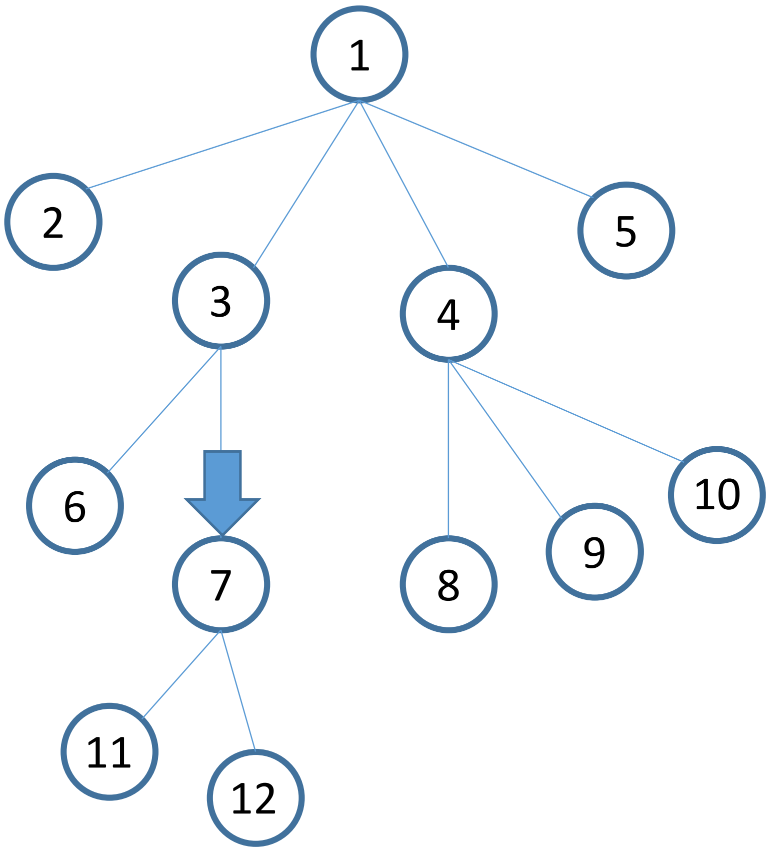
DFS with Stack



Stack

Pop one from the stack, and see it.
It has no children.

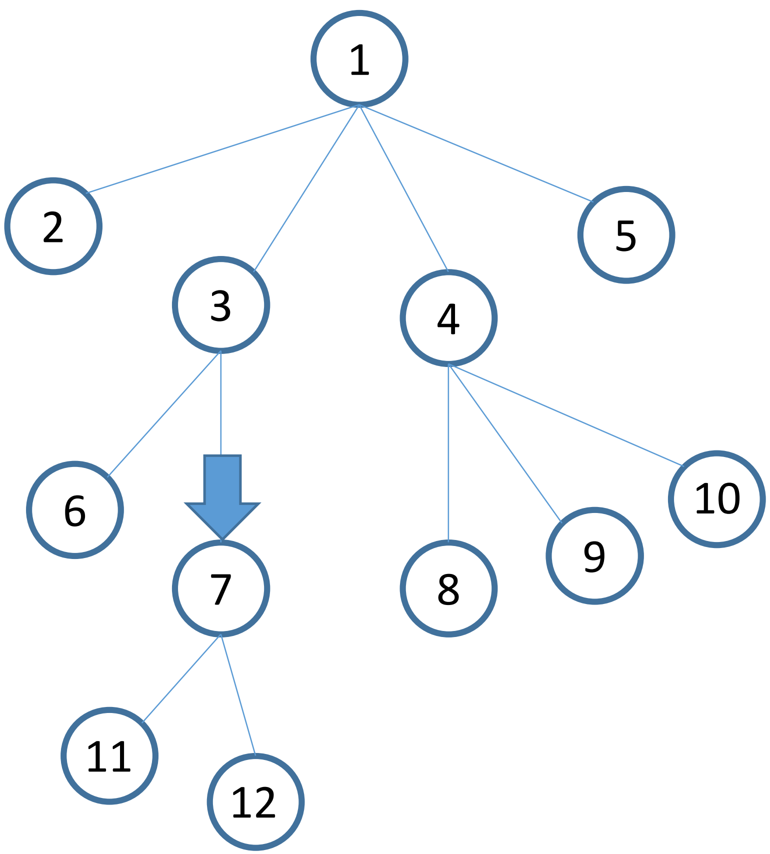
DFS with Stack

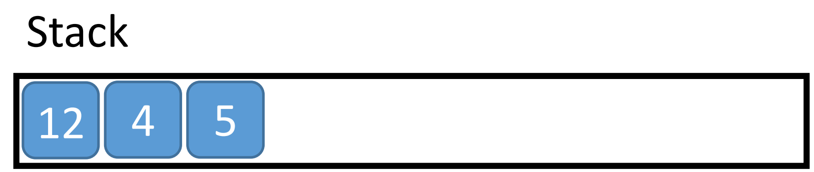


Stack

Pop one from the stack, and see it.
It has its children.

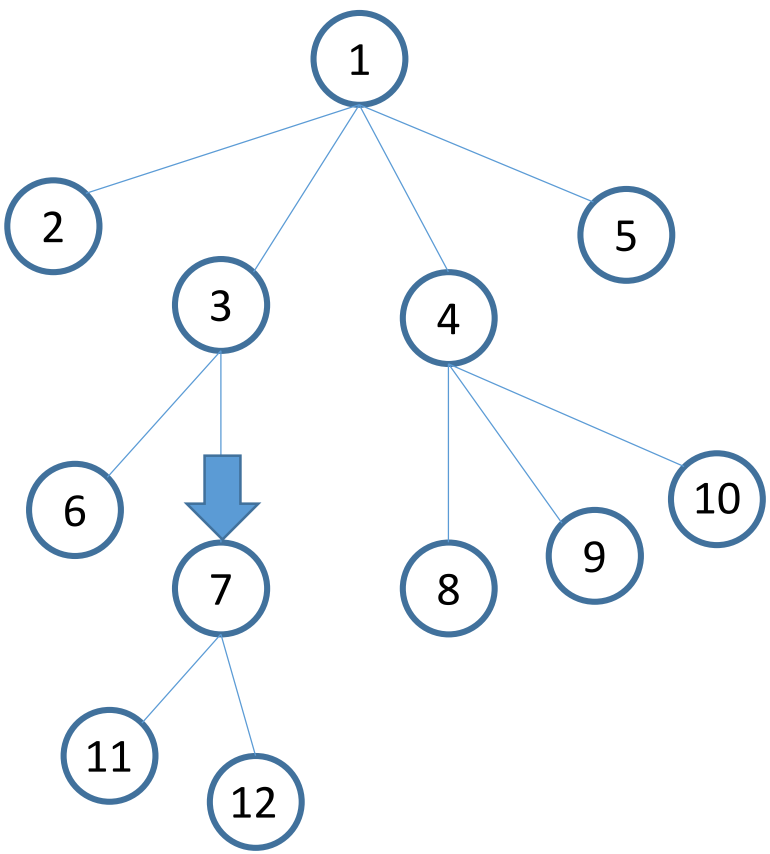
DFS with Stack





Push children of the now-seeing node to the stack, from the back to the front.

DFS with Stack

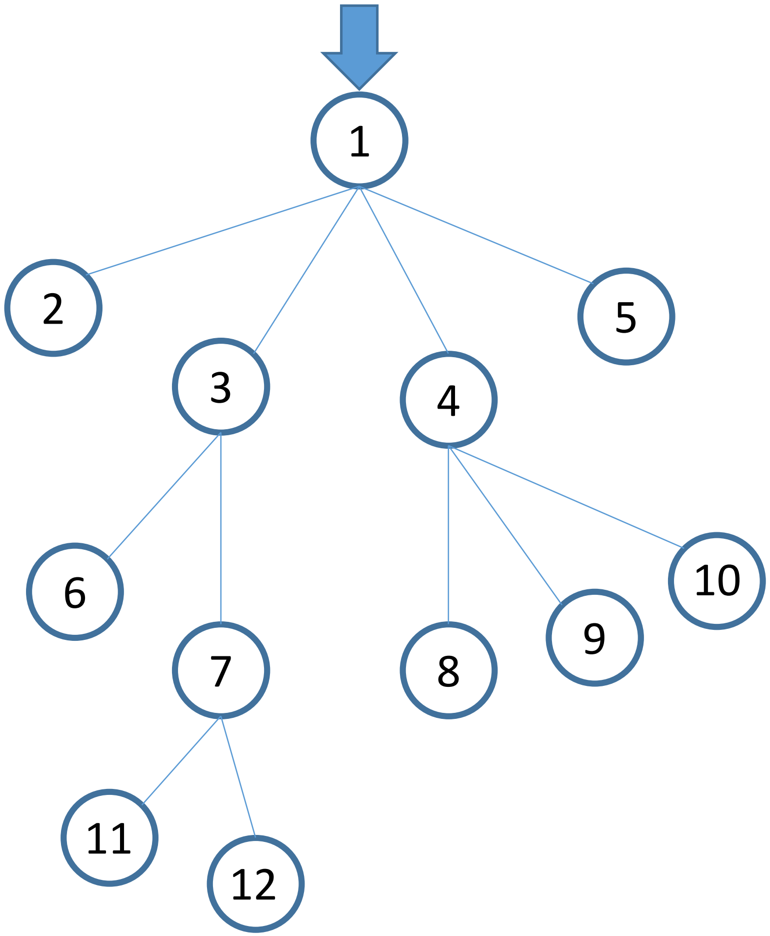


Stack

Push children of the now-seeing node to the stack, from the back to the front.

DFS is performed like this.

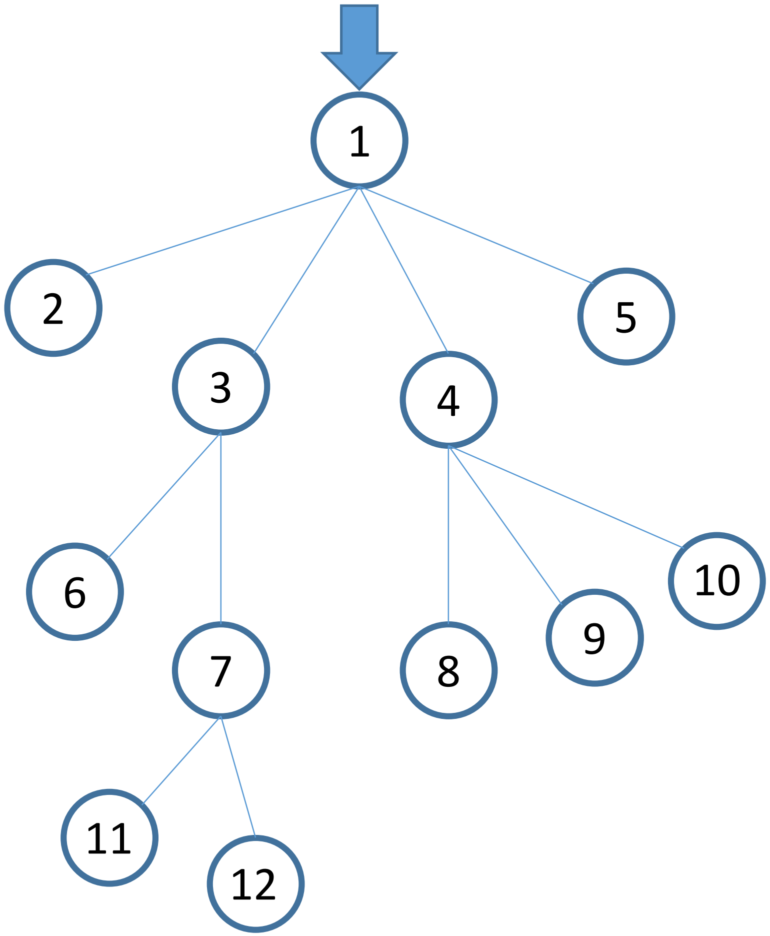
BFS with Queue



Queue

Root node can be in or not be in the queue at the beginning.
It is up to the implementation.

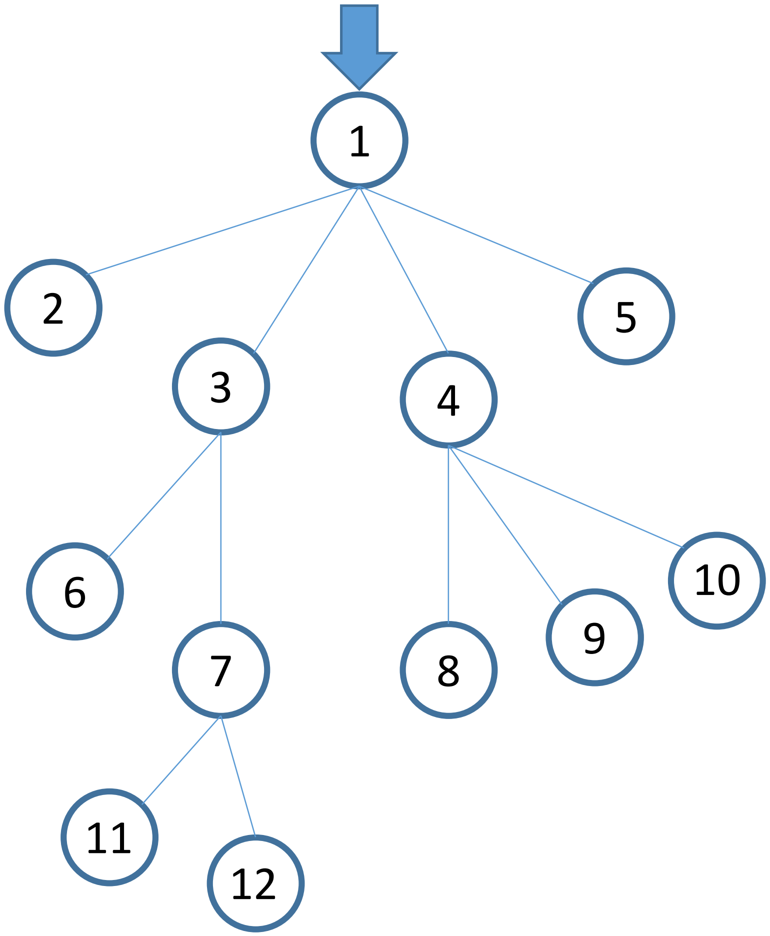
BFS with Queue



Queue

Enqueue children of the now-seeing node to the queue, in order.

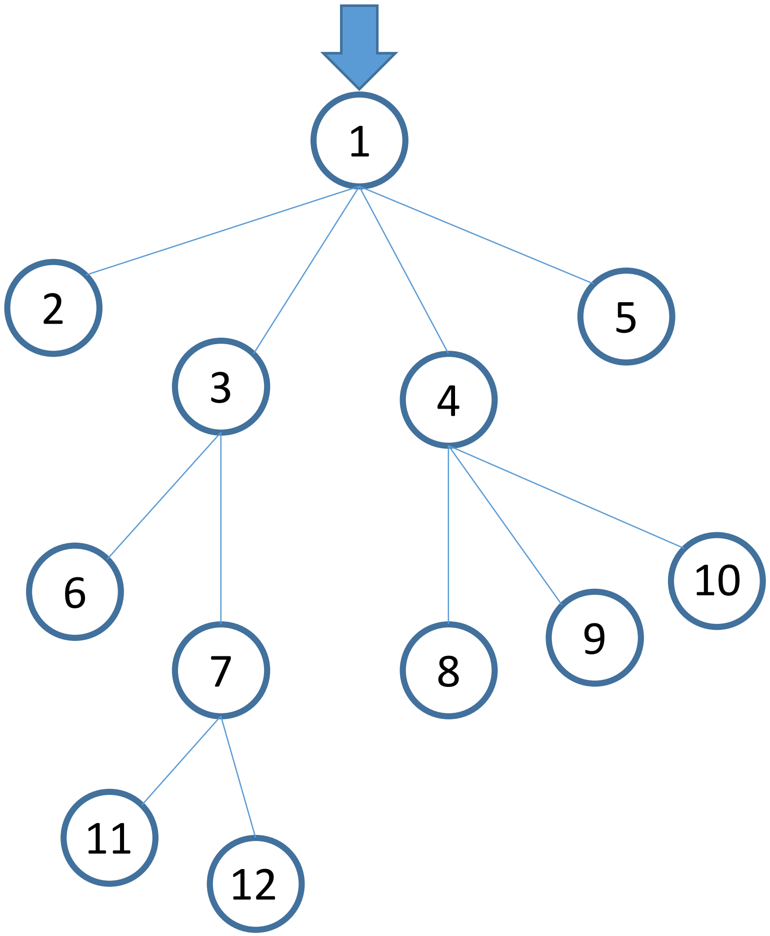
BFS with Queue



Queue

Enqueue children of the now-seeing node to the queue, in order.

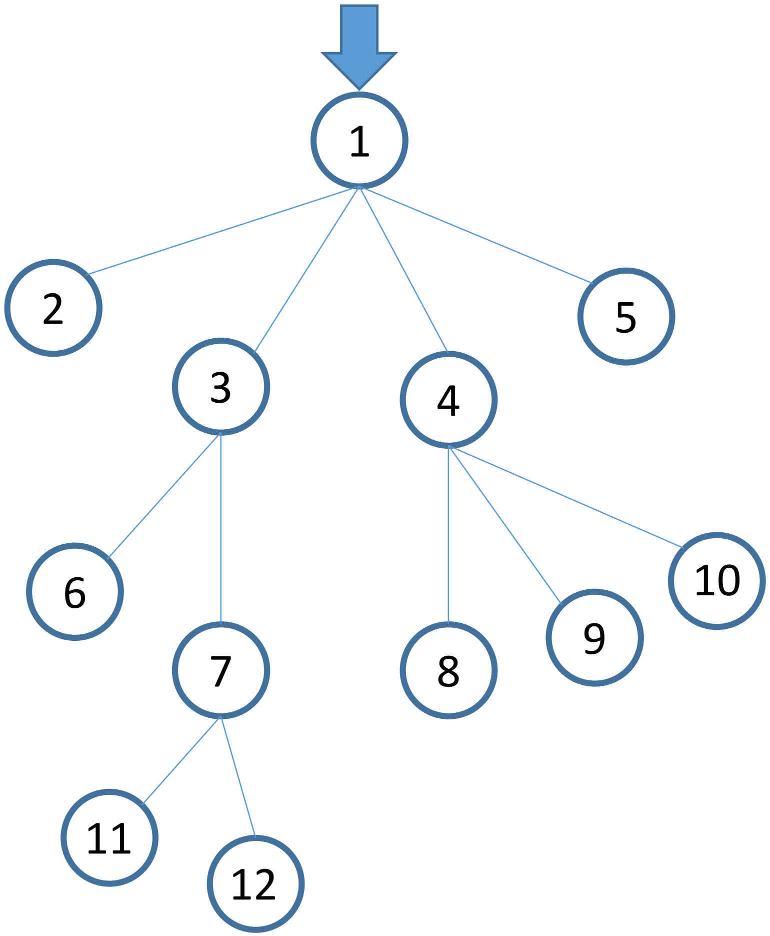
BFS with Queue



Queue

Enqueue children of the now-seeing node to the queue, in order.

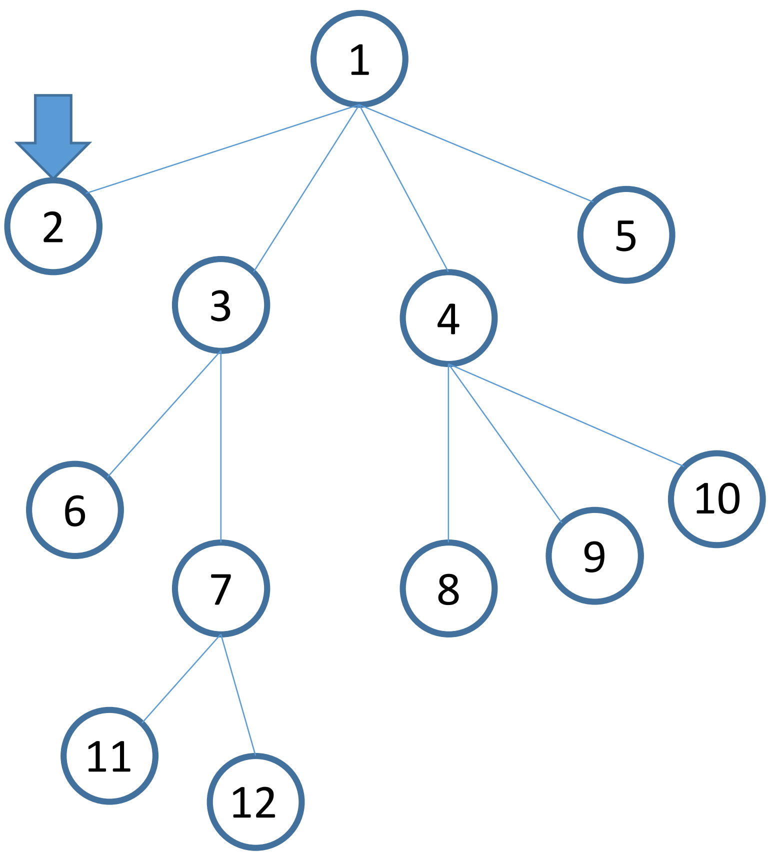
BFS with Queue



Queue

Enqueue children of the now-seeing node to the queue, in order.

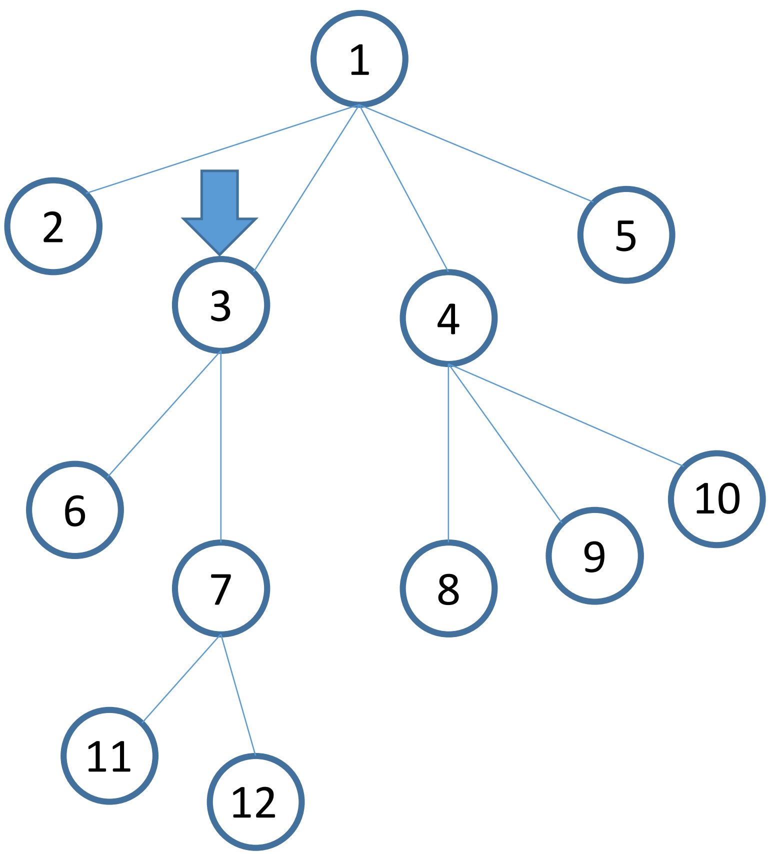
BFS with Queue



Queue

Dequeue one from the stack, and see it.
It has no children.

BFS with Queue

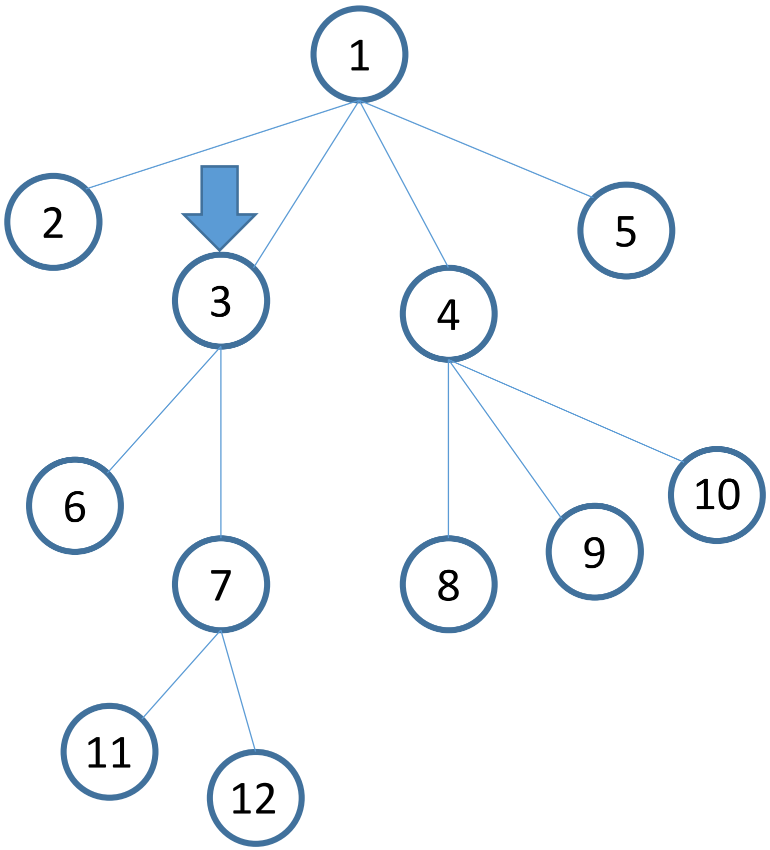


Queue

Dequeue one from the stack, and see it.
It has its children.

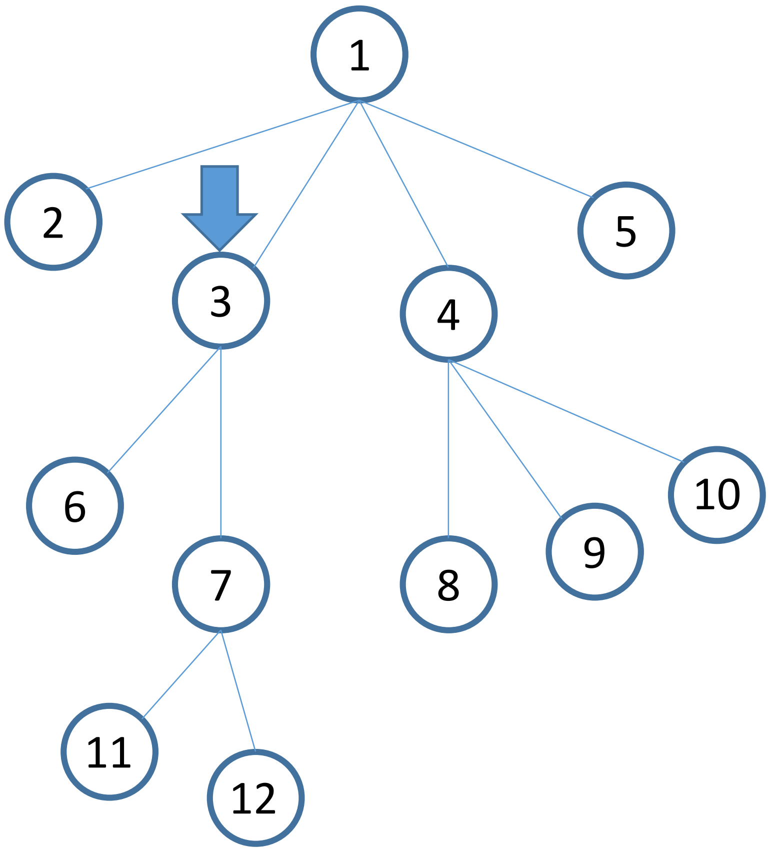
BFS with Queue

Queue



Enqueue children of the now-seeing node to the queue, in order.

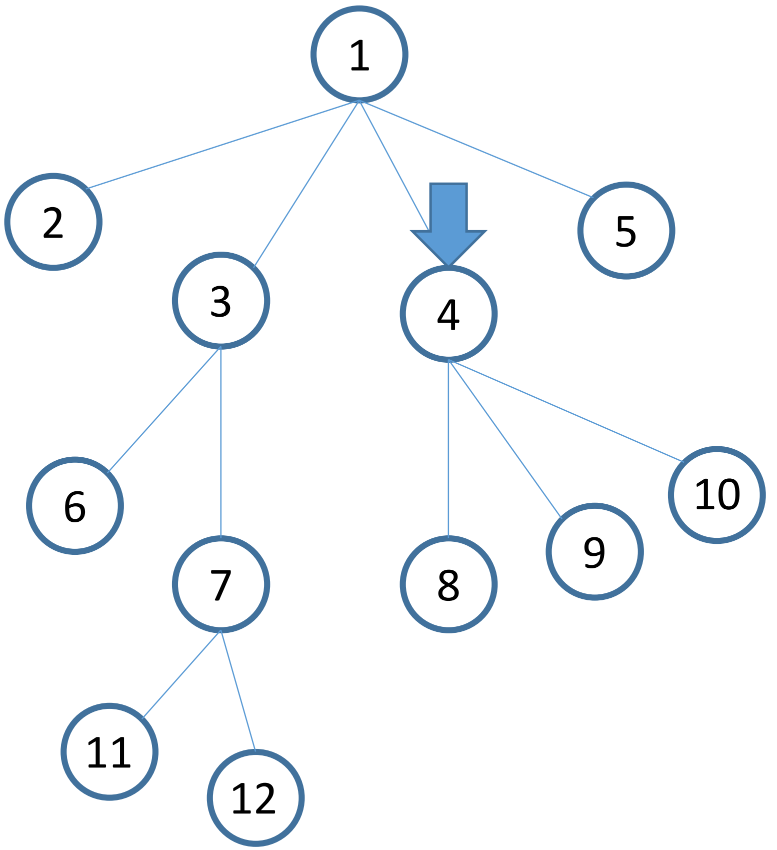
BFS with Queue



Queue

Enqueue children of the now-seeing node to the queue, in order.

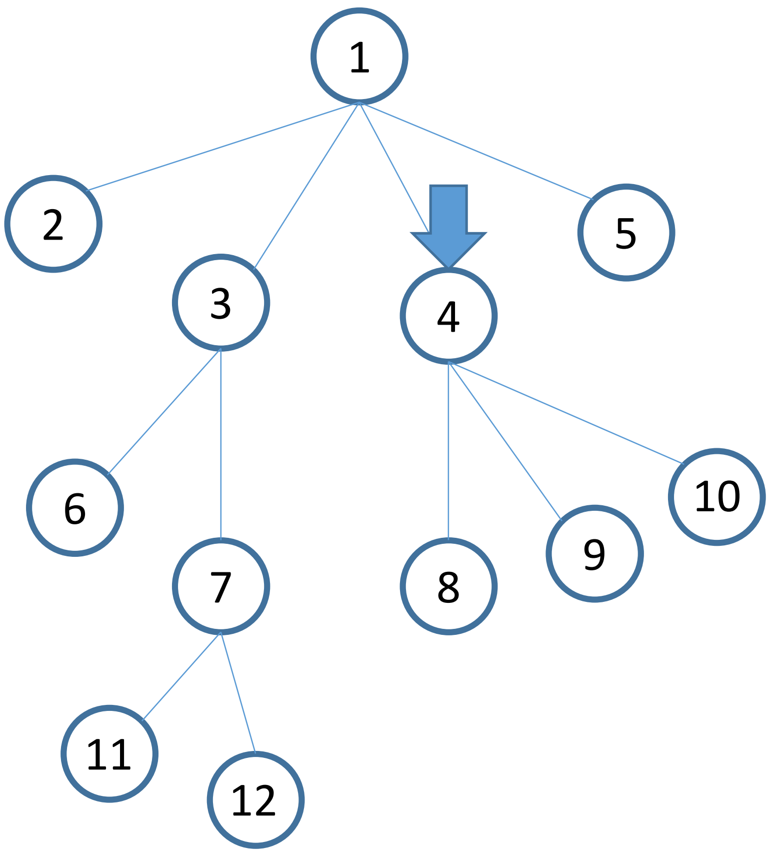
BFS with Queue



Queue

Dequeue one from the stack, and see it.
It has its children.

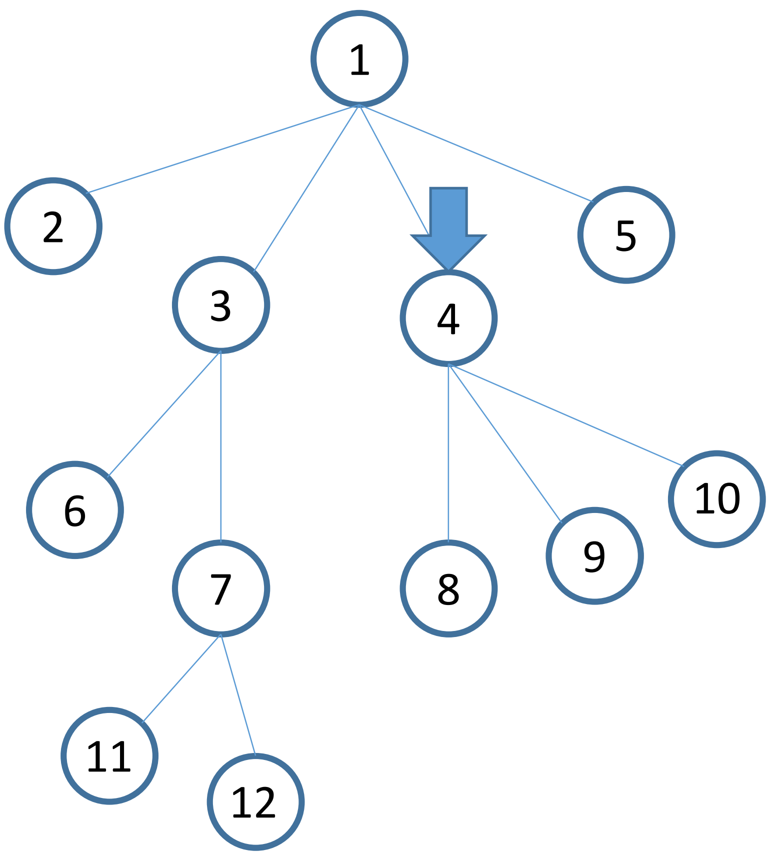
BFS with Queue



Queue

Enqueue children of the now-seeing node to the queue, in order.

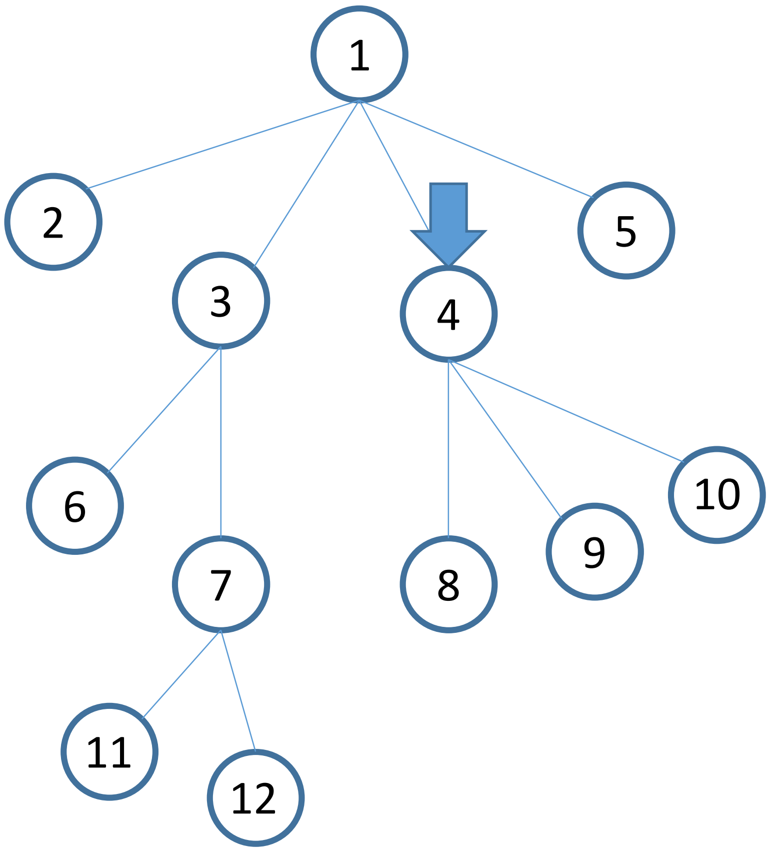
BFS with Queue



Queue

Enqueue children of the now-seeing node to the queue, in order.

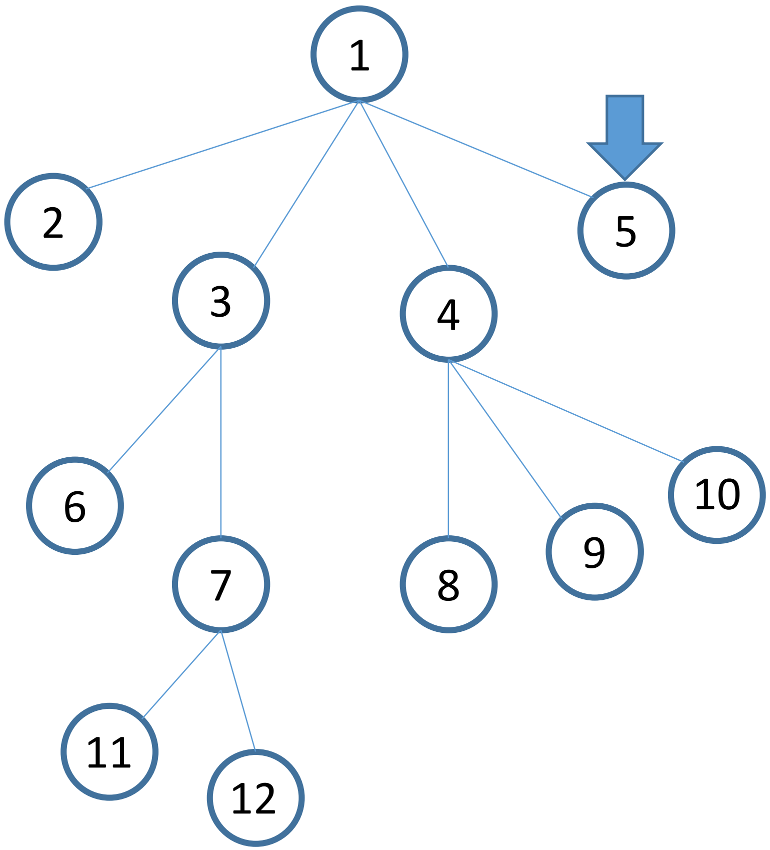
BFS with Queue



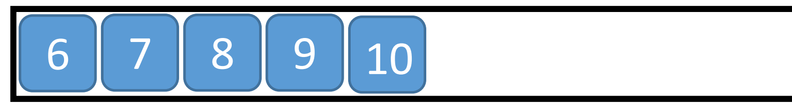
Queue

Enqueue children of the now-seeing node to the queue, in order.

BFS with Queue



Queue

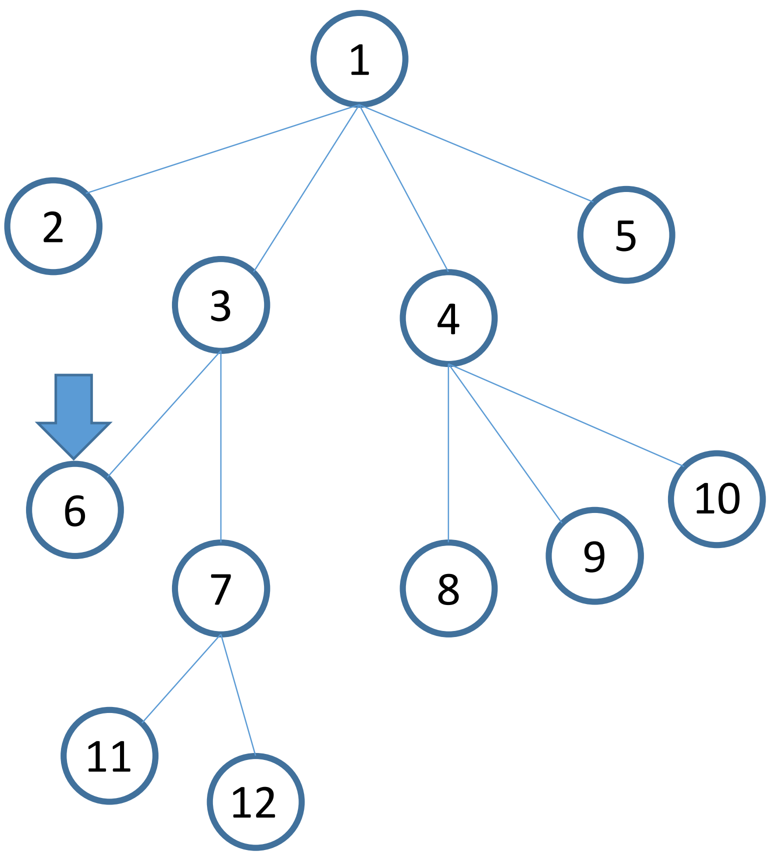


Dequeue one from the stack, and see it.
It has no children.

BFS with Queue

Queue

Dequeue one from the stack, and see it.
It has no children.



BFS is performed like this.

TreeNode

- In the code, the ADTs of a tree data structure has been implemented.
- The class name is TreeNode.
  - TreeNode represents a node of tree data structure.
  - Each node have a non-zero integer value.
  - Each node can have five children at most.
    - Others are ignored to be added.
  - Children are pointed with the pointers in an array.
- Some externally implemented functions are helpful to treat the tree structure.

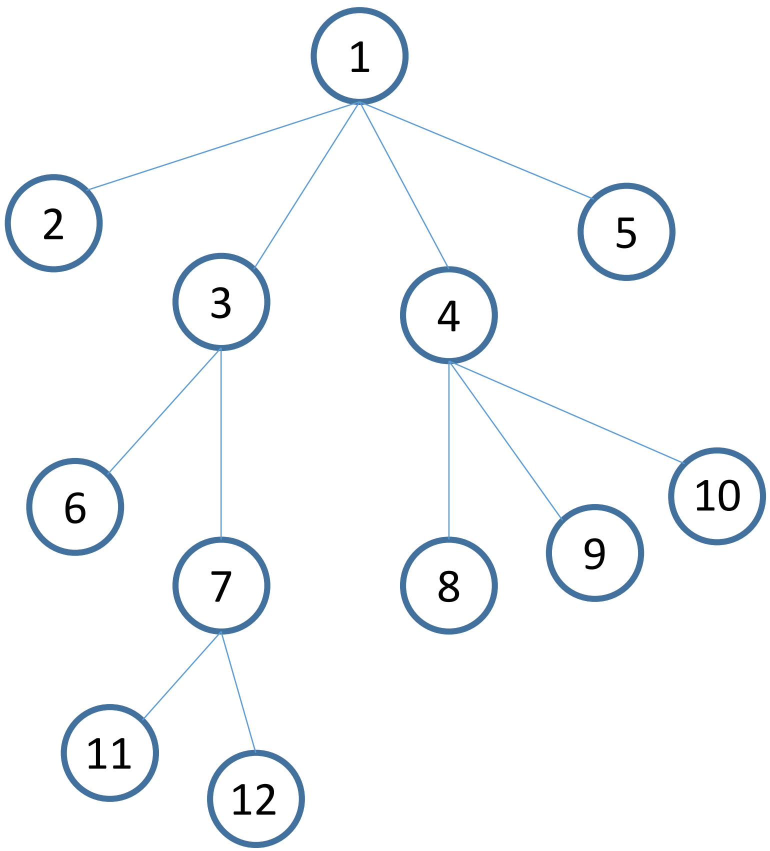
TreeNode

```
class TreeNode{
private:
    int value;
    int no_children = 0;
    TreeNode *children[5] = {0,};
public:
    TreeNode(int v): value(v);
    TreeNode(int v, int cv[], int size);
    TreeNode(int v, TreeNode *c[], int size);
    void add_child(TreeNode *c);
    void add_children(TreeNode *c[], int size);
    void add_children(list<TreeNode*> *c);
    int get_value();
    void set_value(int v);
    TreeNode *get_child(int index);
    TreeNode **get_children();
    int get_no_children();
};
```

External functions

- `TreeNode *make_node_from_string(string str);`
  - To make a tree node from a well formed string.
- `void make_children_from_string`
  `(string str, list<TreeNode*> *children);`
  - To make a list of tree nodes from a well formed string.
- `void print_tree(TreeNode *root);`
  - To print tree as the string.
- `void print_path(list<TreeNode*> path);`
  - To print the path searched by the '*search*' *functions*.

String format



- Each node is expressed as
  - {node value}
    - If the node has no children
  - {node value}[{its children}]
    - Otherwise, children are enclosed in a pair of square brackets
- Children of a node is expressed as
  - {node} {node} …
  - Separated with spaces.
- Example
  - 1[2 3[6 7[11 12]] 4[8 9 10] 5]

Implementing DFS & BFS

- There are the namespaces 'DFS' and 'BFS'.
- In those, functions for searching exist.
  - 'DFS' one uses the list as stack.
  - 'BFS' one uses the list as queue.
- Implemented function(can be modified)
  - `void add_to_list(`
    `list<TreeNode*> *l, TreeNode* children[], int size);`
  - `TreeNode *get_next(std::list<TreeNode*> *l);`
- **Function to be implemented (Your goal)**
  - **`TreeNode *search(`**
    `TreeNode* root, int target, list<TreeNode*> *path);`
    - To search the node whose value is same as 'target',
      and put the search path to reach that node in the 'path'.

Goal of the program

- Input
  - First line: The search target as an integer value.
  - Second line: String representing a root node of tree.
- Output
  - Paths by the two search methods.
    - Depth-first-search with Stack
    - Breadth-first-search with Queue
  - Two lines: One line for DFS, the other line for BFS.
- Please don't be concerned about input & output.
- Just focus on implementing 'search' functions.

Example

- Input
  - -22
    37[5[3 16 -4 1 7] 8[13 2[19 9 -22 4] 6[17]] 14[34 23[20 15] 11] -12[-5 -7 10]]
- Output
  - 37 5 3 16 -4 1 7 8 13 2 19 9 -22
    37 5 8 14 -12 3 16 -4 1 7 13 2 6 34 23 11 -5 -7 10 19 9 -22

Solution

With well-defined get_next() and add_to_list() functions, just one search algorithm can be applied to the both.

```
TreeNode *search(TreeNode* root, int target, list<TreeNode*> *path){
    list<TreeNode*> l;

    l.push_back(root);
    for(TreeNode* now = get_next(&l);
        now;
        now = get_next(&l)){
        path->push_back(now);
        if(now->get_value() == target){
            return now;
        }
        else{
            add_to_list(&l, now->get_children(), now->get_no_children());
        }
    }
    return NULL;
}
```

It is just repeating…

get one, then is it the target?
-> no – add its children
-> yes – return it