Algorithms and Data Structures

Module 1

Lecture 4

Graph traversals: depth-first search, breadth-first search and their applications. Part 1

Adigeev Mikhail Georgievich

mgadigeev@sfedu.ru

Graph traversals

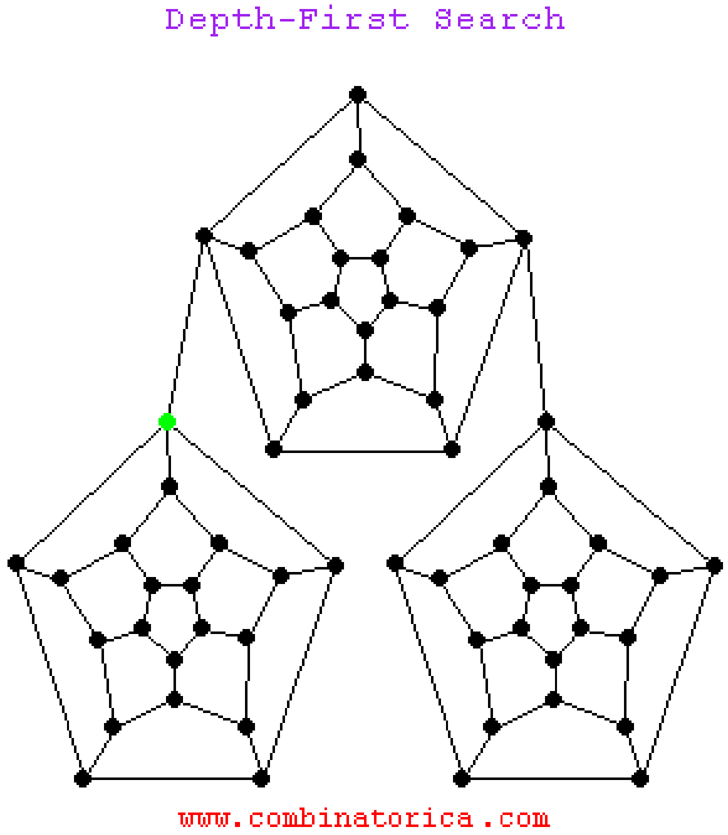
Graph G=(V,E).

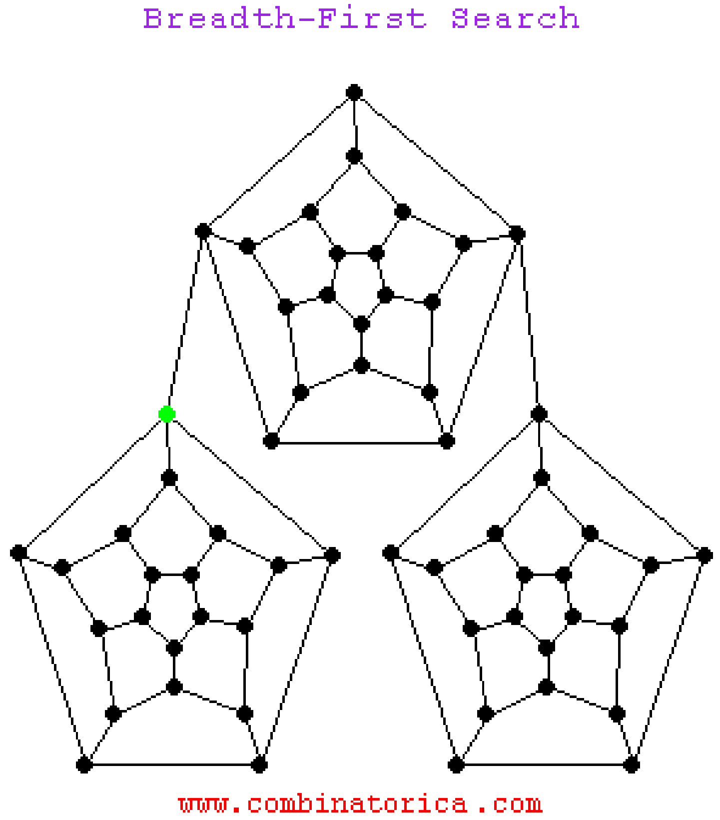
A graph *traversal*: start at a certain vertex and visit other vertices of G in a specific order.

Traversals let us explore the graph and discover its structure.

- Depth-first traversal (DFS)
- Breadth-first traversal (BFS)

Graph traversals





https://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html

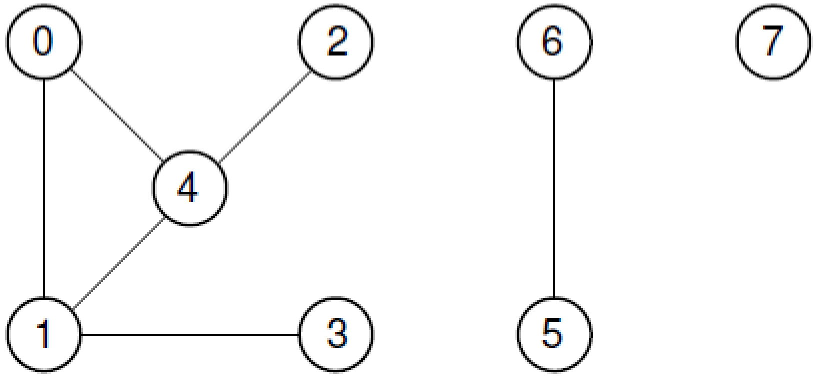
Graph connectivity

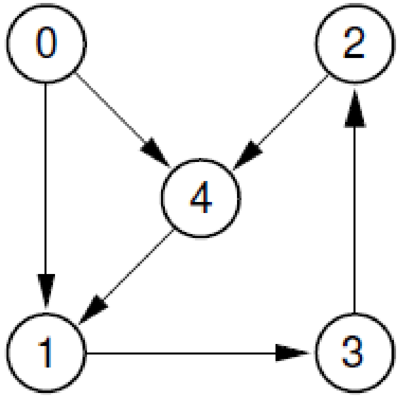
Graph G=(V,E).

A *path* (*walk*) is a sequence of edges $\{e_1, e_2, \ldots, e_l\}$ such that for each i the end-point vertex of e_i is a start-point of e_{i+1}.

Alternative representation: a sequence of vertices $\{v_1, v_2, \ldots, v_{l+1}\}$.

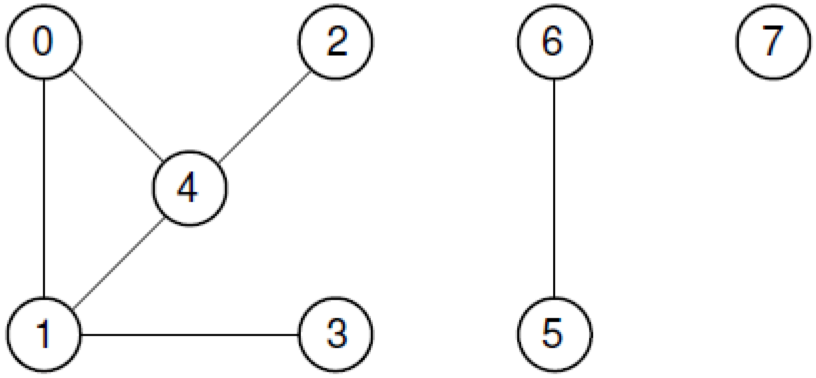
The number of edges = *length* of the path.

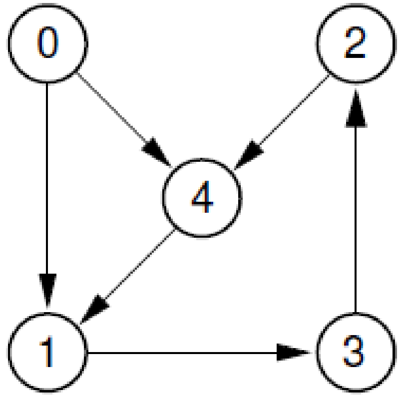




Graph connectivity

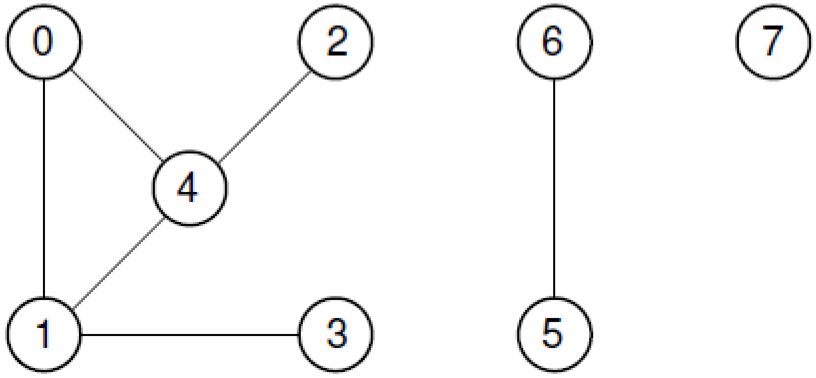
- A path $\{v_1, v_2, \ldots, v_{l+1}\}$ is a *cycle* iff $v_1 = v_{i+1}$.
- A vertex v is *reachable* from the vertex u on G iff there is a path on G from u to v .

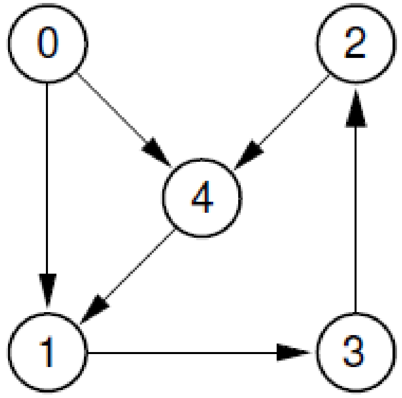




Graph connectivity

- A graph is called *connected* iff for each pair of vertices $\{u, v\}$ there is a path between u and v.
- The maximally connected subgraphs of G are called *connected components*.



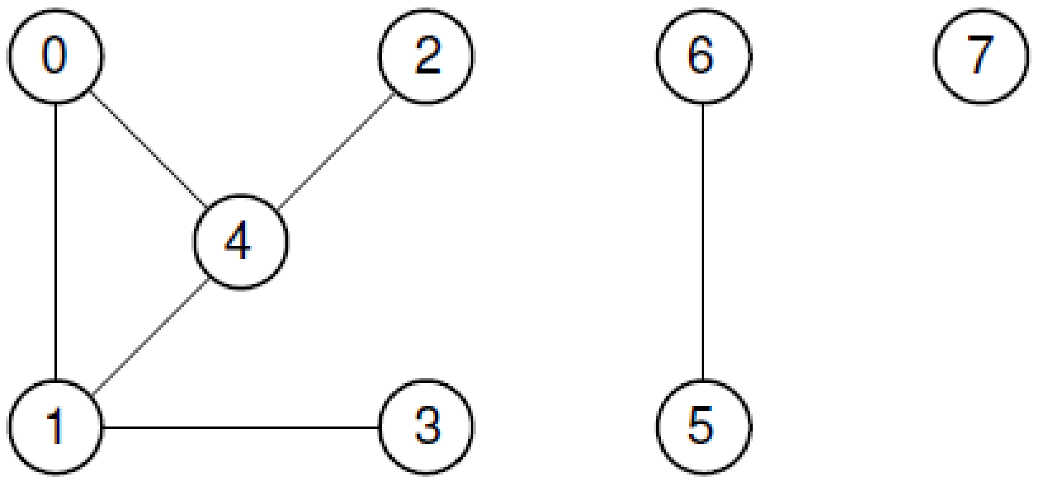


Graph connectivity

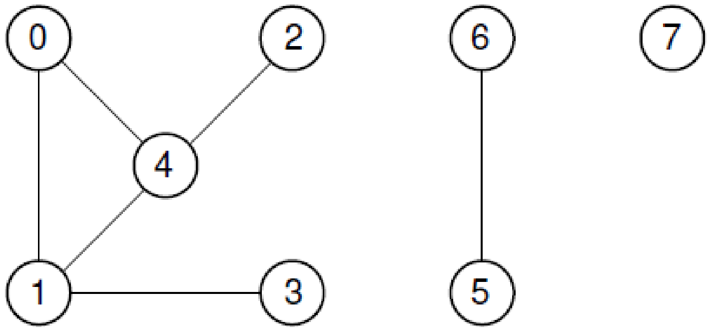
Problem

Given a graph $G(V, E)$, detect all its connected components.

1. $\{0, 1,2,3,4\}$
2. {5,6}
3. {7}



Graph connectivity

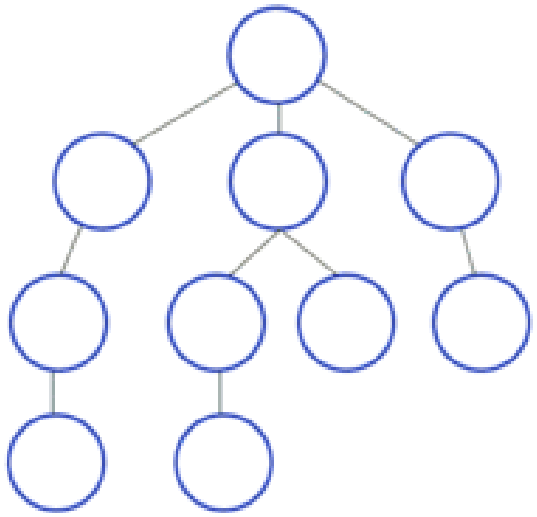


<u>Solution</u>

1. Mark all vertices as 'unvisited'.
2. While there is an unvisited vertex *s*:
3. Initialize a new component C_k.
4. Start DFS/BFS from *s*.
5. Visiting a vertex, put it into C_k.

DFS: Depth-First Search

Visiting a vertex v, recursively visit (start DFS) each of its unvisited neighbors.



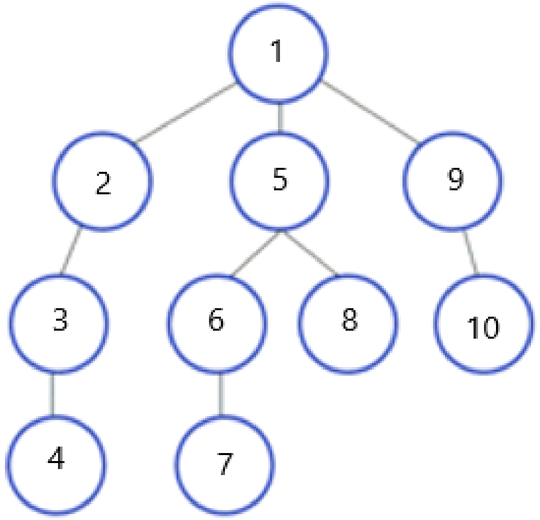
https://en.wikipedia.org/wiki/Depth-first_search

```
DFS(v)
Mark v as 'visited'
For each u in Adj(v):
    if u is unvisited:
        DFS(u)
```

DFS: Depth-First Search

Visiting a vertex v, recursively visit (start DFS) each of its unvisited neighbors.

```
DFS(v)
Mark v as 'visited'
For each u in Adj(v):
    if u is unvisited:
        DFS(u)
```



https://en.wikipedia.org/wiki/Depth-first_search

DFS: Depth-First Search

For graph exploration, we often need to perform some processing before / after recursive DFS.

```
DFS(v)
PreVisit(v)
Mark v as 'visited'
For each u in Adj(v):
   if u is unvisited: DFS(u)
PostVisit(v)
```

DFS: explicit stack implimentation

Recursive implementation can lead to ‘stack overflow’ error ☹

=> An alternative implementation using explicit *stack*.

Stack: abstract data structure

Stack = abstract data structure with two principal operations:

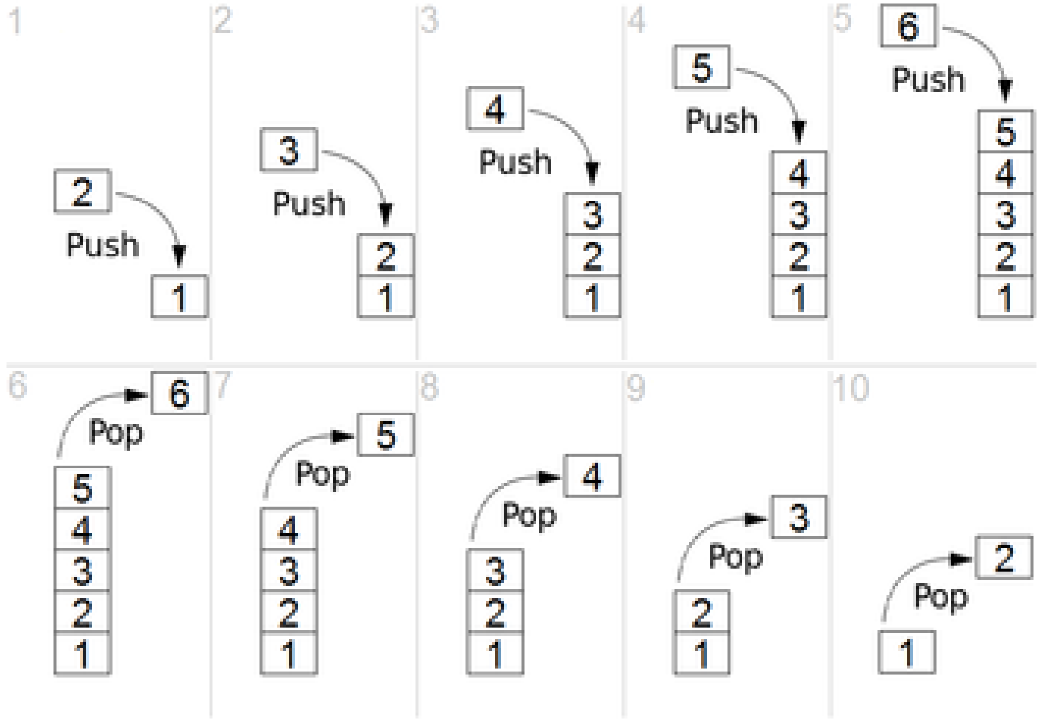
- Push(*item*)
- Pop()

LIFO = Last-In, First-Out

Stack: abstract data structure

Stack = abstract data structure with two principal operations:

- `Push(`*`item`*`)`
- `Pop()`
- [Get the top item]



https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Stack: abstract data structure

```
// Stack abtract class
template <typename E> class Stack {
private:
  void operator =(const Stack&) {}     // Protect assignment
  Stack(const Stack&) {}         // Protect copy constructor

public:
  Stack() {}                        // Default constructor
  virtual ˜Stack() {}               // Base destructor

  // Reinitialize the stack.  The user is responsible for
  // reclaiming the storage used by the stack elements.
  virtual void clear() = 0;

  // Push an element onto the top of the stack.
  // it: The element being pushed onto the stack.
  virtual void push(const E& it) = 0;

  // Remove the element at the top of the stack.
  // Return: The element at the top of the stack.
  virtual E pop() = 0;

  // Return: A copy of the top element.
  virtual const E& topValue() const = 0;

  // Return: The number of elements in the stack.
  virtual int length() const = 0;
};
```

https://people.cs.vt.edu/~shaffer/Book/

Stack: implementation

A stack data structure can be implemented in different ways:

- array based
- linked-list based

Stack: array-based implementation

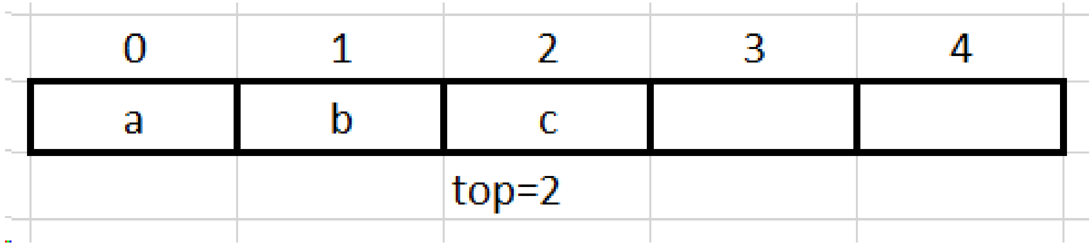
```
template <typename E> class ads_stack_array: public Stack<E>
```

Version 1: STL array

```
{
protected:
        std::vector<E> arr;
}
```

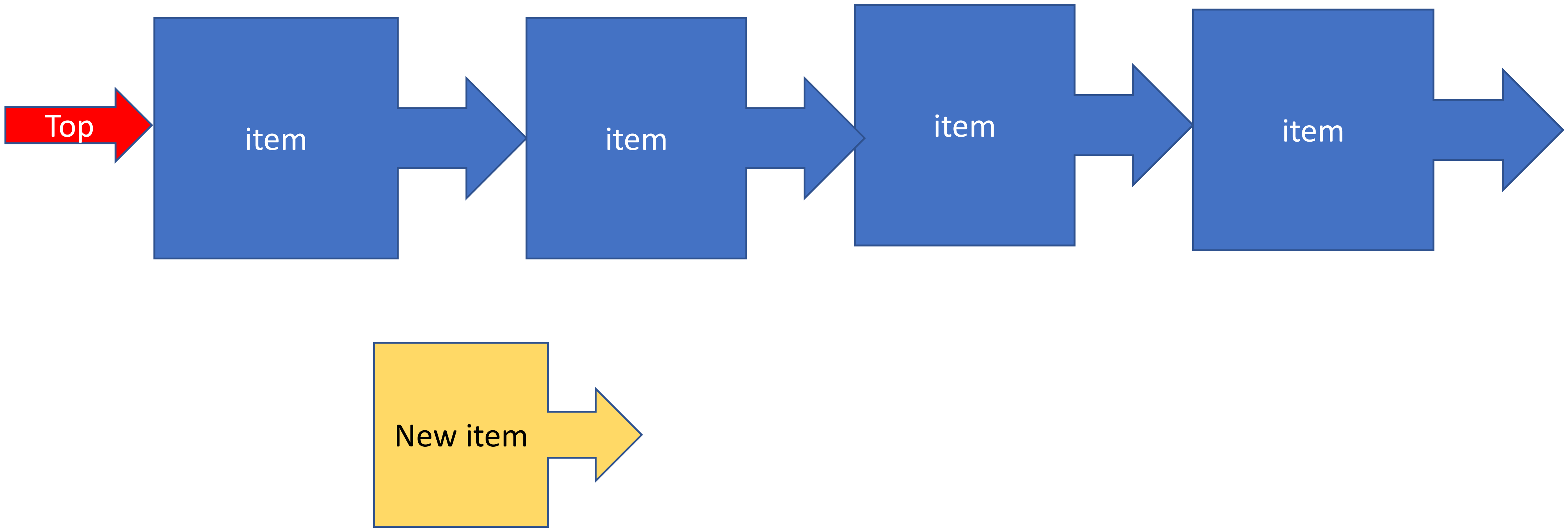
Version 2: static array

```
{
protected:
        E* arr;
        size_t top; // index of the top item
         void resize();
}
```



Stack: dynamic list-based implementation

A dynamic list data structure with top (=head) pointer.



DFS: explicit stack implementation

```
StackDFS(G)
Select s ∈ V
Push(s)
While (stack is not empty):
    v = Pop()
    if v is unvisited:
        Mark v as 'visited'
        For each u in Adj(v):
            Push(v)
```

DFS: example