Graphs

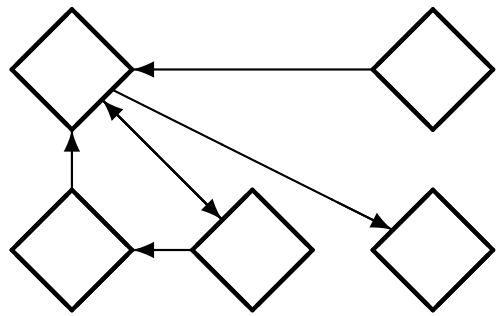
November 26, 2018

- Graphs
- Implementing Graphs
- Traversing Graphs

Graphs

1: Graphs

By the term *graph*, we shall mean a structure such as:



Thus:

> *A graph is a* collection of nodes *(sometimes called* vertices*), connected by* edges *(sometimes referred to as* arcs*).*

Adjacency Tables

2: Implementing Graphs

2.1: Adjacency Tables

One way to represent graphs is using a *two dimensional array*.
If node **i** has an edge to node **j** in our graph then:

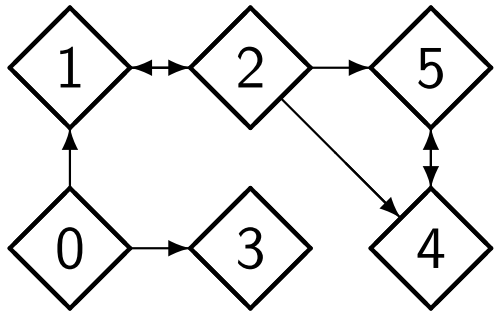
$$\texttt{graph[i][j]} = \textbf{true}$$

otherwise:

$$\texttt{graph[i][j]} = \textbf{false}$$

Example

Example: The following graph:



can be represented by the following *adjacency table*:

	0	**1**	**2**	**3**	**4**	**5**
0	false	true	false	true	false	false
1	false	false	true	false	false	false
2	false	true	false	false	true	true
3	false	false	false	false	false	false
4	false	false	false	false	false	true
5	false	false	false	false	true	false

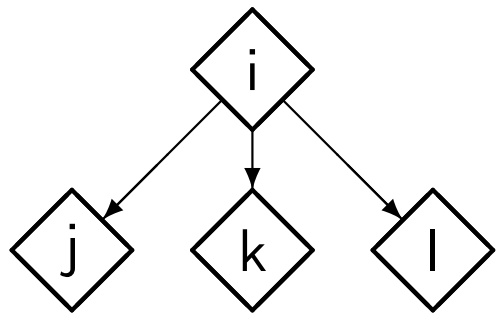
Adjacency List

2.2: Adjacency List

Adjacency lists are sometimes also referred to as *associative lists*.
Here we use a one dimensional array.
Each entry in the array is a *set of vertices*.
Each set describes the graph vertices accessible from a given vertex.

Adjacency List

If vertex **i** is *only* joined (via an edge) to the vertices **j**, **k** and **l**:

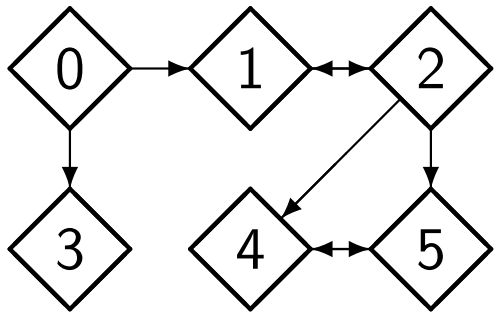


then:

$$\texttt{graph[i]} = \{\texttt{j}, \texttt{k}, \texttt{l}\}$$

Example

Example: The graph:



can be represented by the following adjacency list:

0	$\{1, 3\}$
1	$\{2\}$
2	$\{1, 4, 5\}$
3	$\emptyset$
4	$\{5\}$
5	$\{4\}$

Notes

2.3: Notes

2.3.1: Non-Integer Nodes

If we use Hashtables we may label vertices using other objects
This is particularly useful when we use graphs to model, e.g.:

- parse trees
- lazy data structures (in functional programming languages)
- relational data structures (in databases and logic programming languages)
- states of machines (e.g. Turing machines and push-down automaton)
- etc.

Space Efficiency

2.3.2: Space Efficiency

For sparse graphs (i.e. many more nodes than edges), adjacency tables *waste* a lot of space.

- need $(\text{number of nodes})^2$ memory cells in order to store the array
- for large graphs (e.g. number of vertices $\geq 2^{10}$), we do not have enough space to store an adjacency table!

From now on, we shall assume that *all* our graphs will be modeled using *adjacency lists*.

Traversing Graphs

3: Traversing Graphs

3.1: Methods

Many graph algorithms require us to visit *all* nodes of the graph e.g. in determining what nodes are *path-accessible* from a given node or *searching* for an appropriate node

There are *two* main ways in which we may traverse graphs:

- depth-first traversal
- breadth-first traversal

Depth-first

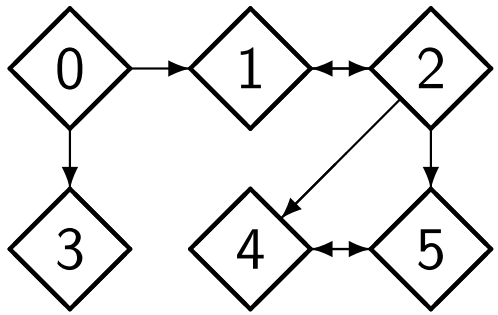
3.1.1: Depth-First Traversal

The idea of a *depth-first traversal* is to follow a chain of edges as far as we can into the graph (i.e. we go as deep as the rabbit hole allows).
When we can follow the edges no more (i.e. we can go no deeper), we backtrack to find previously visited nodes out of which other edge chains (i.e. paths) may lead.
To avoid following the same path more than once, it is imperative that we *mark nodes* as visited when we reach them.
Thus, each node needs a *flag* associated with it[1].

[1] In the following example we can check a node's status by checking the "traversal" list, but this need not generally be so.

Depth-first

Example: Consider the graph:

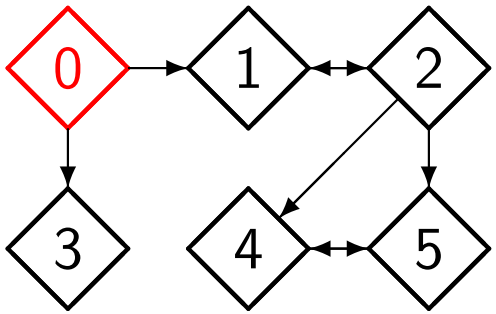


which may be represented by the following adjacency list:

0	false	$\{1, 3\}$
1	false	$\{2\}$
2	false	$\{1, 4, 5\}$
3	false	$\emptyset$
4	false	$\{5\}$
5	false	$\{4\}$

Depth-first

- Start at node **0** — mark as visited:

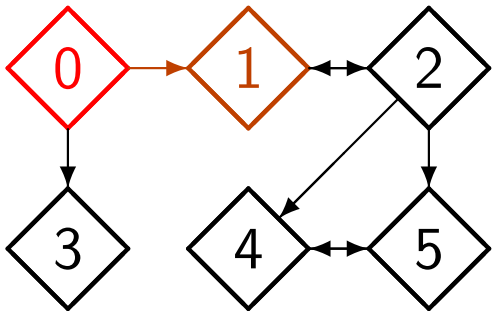


0	*true*	$\{1, 3\}$
1	false	$\{2\}$
2	false	$\{1, 4, 5\}$
3	false	$\emptyset$
4	false	$\{5\}$
5	false	$\{4\}$

- Traversal list: **[0]**
- Backtrack list: **[0]**

Depth-first

- Node **1** has not yet been visited:

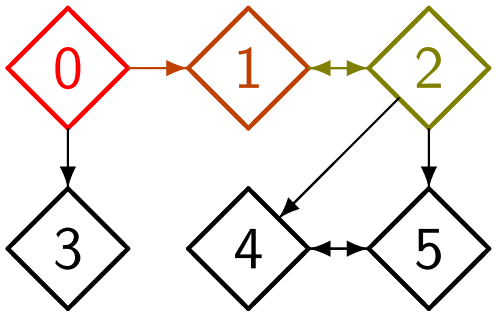


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	false	$\{1, 4, 5\}$
3	false	$\emptyset$
4	false	$\{5\}$
5	false	$\{4\}$

- Traversal list: **[0, 1]**
- Backtrack list: **[0, 1]**

Depth-first

- Node **2** has not yet been visited:

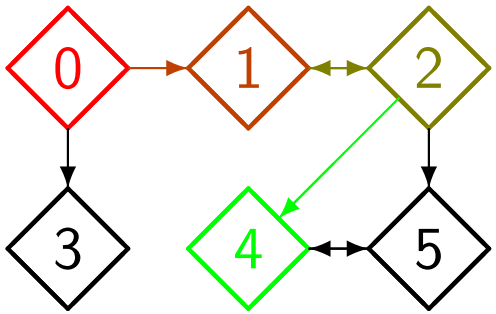


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	*true*	$\{1, 4, 5\}$
3	false	$\emptyset$
4	false	$\{5\}$
5	false	$\{4\}$

- Traversal list: **[0, 1, 2]**
- Backtrack list: **[0, 1, 2]**

Depth-first

- The next node to be visited is node **4**

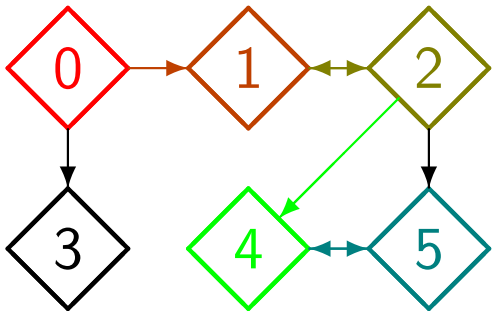


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	*true*	$\{1, 4, 5\}$
3	false	$\emptyset$
4	*true*	$\{5\}$
5	false	$\{4\}$

- Traversal list: **[0, 1, 2, 4]**
- Backtrack list: **[0, 1, 2, 4]**

Depth-first

- Node **5** has not yet been visited:

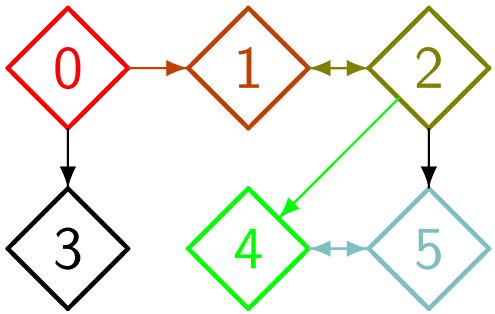


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	*true*	$\{1, 4, 5\}$
3	false	$\emptyset$
4	*true*	$\{5\}$
5	*true*	$\{4\}$

- Traversal list: **[0, 1, 2, 4, 5]**
- Backtrack list: **[0, 1, 2, 4, 5]**

Depth-first

- Backtrack to node **4**

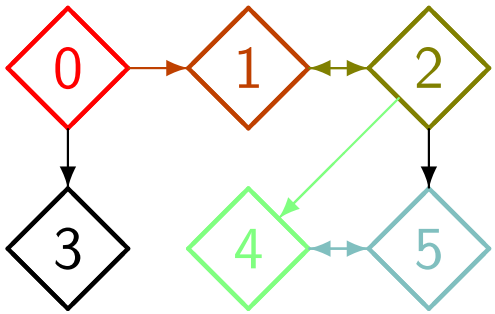


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	*true*	$\{1, 4, 5\}$
3	false	$\emptyset$
4	*true*	$\{5\}$
5	*true*	$\{4\}$

- Traversal list: **[0, 1, 2, 4, 5, 4]**
- Backtrack list: **[0, 1, 2, 4]**

Depth-first

- Backtrack to node **2**.

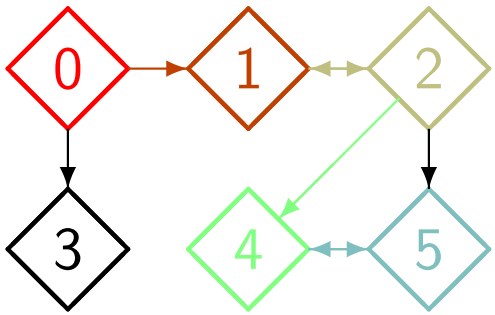


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	*true*	$\{1, 4, 5\}$
3	false	$\emptyset$
4	*true*	$\{5\}$
5	*true*	$\{4\}$

- Traversal list: **[0, 1, 2, 4, 5, 4, 2]**
- Backtrack list: **[0, 1, 2]**

Depth-first

- Nodes **1**, **4** and **5** all visited so backtrack to node **1**

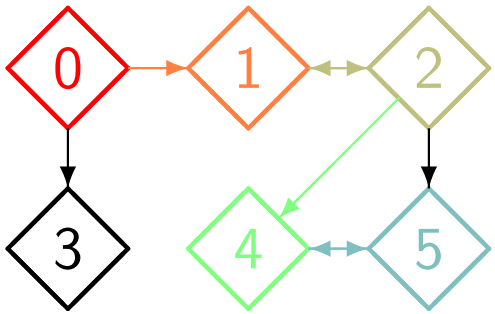


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	*true*	$\{1, 4, 5\}$
3	false	$\emptyset$
4	*true*	$\{5\}$
5	*true*	$\{4\}$

- Traversal list: **[0, 1, 2, 4, 5, 4, 2, 1]**
- Backtrack list: **[0, 1]**

Depth-first

- Again no new nodes to be visited so backtrack to node **0**

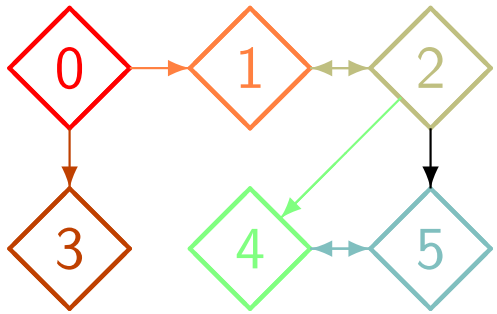


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	*true*	$\{1, 4, 5\}$
3	false	$\emptyset$
4	*true*	$\{5\}$
5	*true*	$\{4\}$

- Traversal list: **[0, 1, 2, 4, 5, 4, 2, 1, 0]**
- Backtrack list: **[0]**

Depth-first

- Node **3** has not yet been visited:

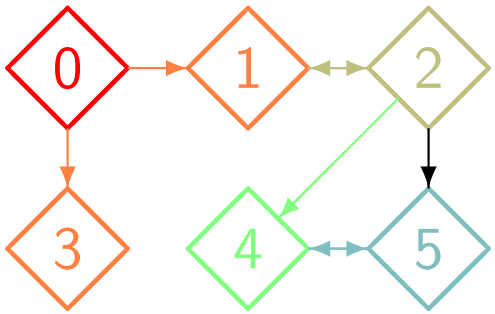


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	*true*	$\{1, 4, 5\}$
3	*true*	$\emptyset$
4	*true*	$\{5\}$
5	*true*	$\{4\}$

- Traversal list: **[0, 1, 2, 4, 5, 4, 2, 1, 0, 3]**
- Backtrack list: **[0, 3]**

Depth-first

- Backtrack to node **0**:



0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	*true*	$\{1, 4, 5\}$
3	*true*	$\emptyset$
4	*true*	$\{5\}$
5	*true*	$\{4\}$

- Traversal list: **[0, 1, 2, 4, 5, 4, 2, 1, 0, 3, 0]**
- Backtrack list: **[0]**

Depth-first

- Nodes **1** and **3** are accessible from node **0**, but they have both been visited.
  They are no other nodes to consider at this level so we attempt to backtrack.
  In doing this, our backtrack list becomes empty.
  We now search through our array looking for a node that has not yet been visited. No such node exists so our algorithm ends.

Thus, we see that a *depth-first traversal* for this graph is:

$$[0, 1, 2, 4, 5, 4, 2, 1, 0, 3, 0]$$

or, ignoring revisits:

$$[0, 1, 2, 4, 5, 3]$$

Breadth-first

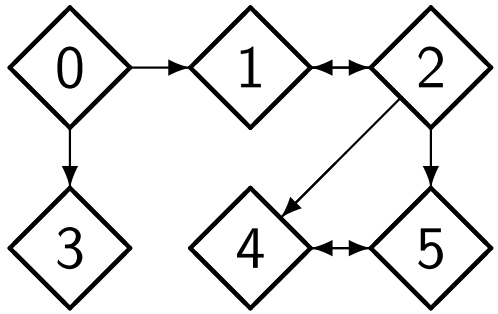
3.1.2: Breadth-First Traversal

The idea of the *breadth-first traversal* is to first visit the *immediate successors* of a given vertex, and then to successively look at each of their immediate successors (which have not yet been visited)[2].

[2] Again we use a "visited" flag, although that is not strictly necessary if we maintain a "traversal" list.

Breadth-first

Example: Consider the graph:



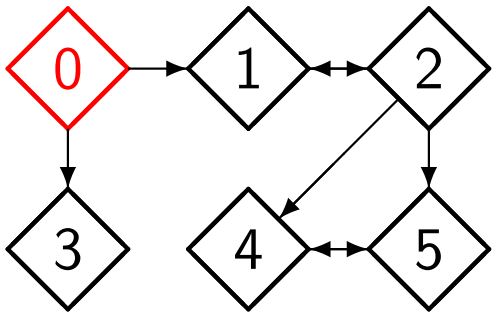
which may be represented by the following adjacency list:

0	false	$\{1, 3\}$
1	false	$\{2\}$
2	false	$\{1, 4, 5\}$
3	false	$\emptyset$
4	false	$\{5\}$
5	false	$\{4\}$

We shall also maintain a traversal list and a to do list.

Breadth-first

- start at node **0**, thus it is marked as visited:

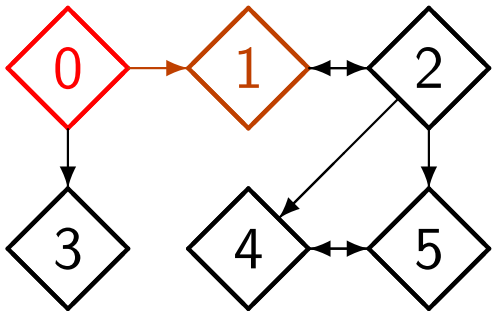


0	*true*	$\{1, 3\}$
1	false	$\{2\}$
2	false	$\{1, 4, 5\}$
3	false	$\emptyset$
4	false	$\{5\}$
5	false	$\{4\}$

- Traversal list: **[0]**
- To do list: **[1, 3]**

Breadth-first

- Node **1** has not yet been visited

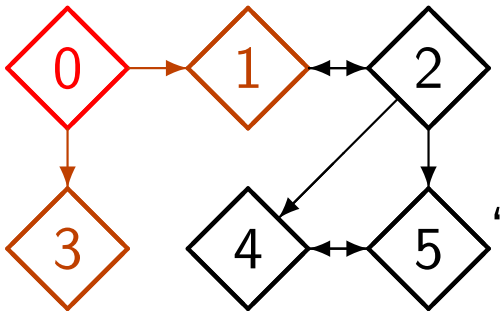


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	false	$\{1, 4, 5\}$
3	false	$\emptyset$
4	false	$\{5\}$
5	false	$\{4\}$

- Traversal list: **[0, 1]**
- To do list: **[3, 2]**

Breadth-first

- Node **3** has not yet been visited

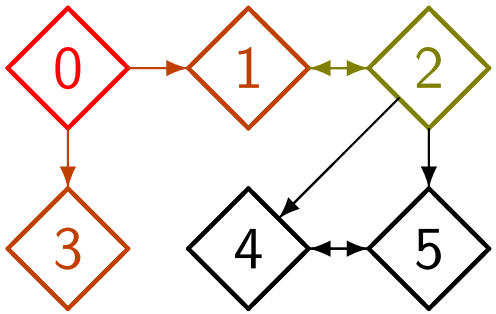


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	false	$\{1, 4, 5\}$
3	*true*	$\emptyset$
4	false	$\{5\}$
5	false	$\{4\}$

- Traversal list: **[0, 1, 3]**
- To do list: **[2]**

Breadth-first

- Node **2** has not yet been visited

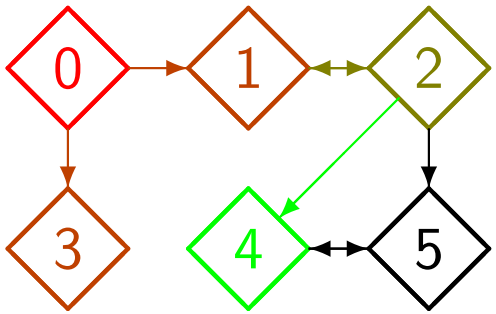


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	*true*	$\{1, 4, 5\}$
3	*true*	$\emptyset$
4	false	$\{5\}$
5	false	$\{4\}$

- Traversal list: $[0, 1, 3, 2]$
- To do list: $[4, 5]$

Breadth-first

- Node **4** has yet to be visited

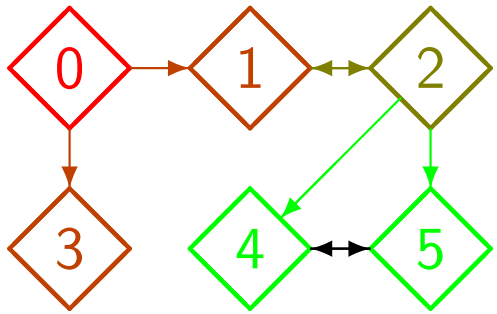


0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	*true*	$\{1, 4, 5\}$
3	*true*	$\emptyset$
4	*true*	$\{5\}$
5	false	$\{4\}$

- Traversal list: **[0, 1, 3, 2, 4]**
- To do list: **[5]**

Breadth-first

- Node **5** has yet to be visited



0	*true*	$\{1, 3\}$
1	*true*	$\{2\}$
2	*true*	$\{1, 4, 5\}$
3	*true*	$\emptyset$
4	*true*	$\{5\}$
5	*true*	$\{4\}$

- Traversal list: **[0, 1, 3, 2, 4, 5]**
- To do list: **[]**

Breadth-first

- Our *to do list* is empty.
  We now search through our nodes looking for one that has yet to be visited.
  No such nodes exist, so our algorithm ends.

Thus, we see that a *breadth-first traversal* for this graph is:

$$[0, 1, 3, 2, 4, 5]$$

Breadth First Traversals

3.2: Implementation
3.2.1: Breadth First
3.2.1 (i): Visit to a Single Node

First we define a method that encapsulates the actions required to visit a single node. This method will

- check that the node has not already been visited
- note that this node has been visited
- add any unvisited children of this node to the "to do" list

Has node been visited?

3.2.1 (ii): Has the node been visited?

Maintain a set of visited nodes

```
// if we have already visited or are due to visit this node
// there is nothing to do
if (isOnVisitList(node)) return;
```

Visit node

3.2.1 (iii): Visit node

```
// add this node to the traversal
traversal.add(node);
```

Add successors to the to do list

3.2.1 (iv): Add unvisited successors to the “to do” list

```
// add successors to the “to do” list
for (T next:  getNeighbours(node)) {
    if (!isOnVisitList(next)) {// unless they have been visited
        toDo.add(next);
    }
}
```

Full version

3.2.1 (v): Full Version

```
private void visitNode(T node) {
    if (isOnVisitList(node)) return;
    traversal.add(node);
    for (T next:  getNeighbours(node)) {
        if (!isOnVisitList(next)) {
            toDo.add(next);
        }
    }
}
```

Visit the whole to do list

3.2.1 (vi): Visit all the "To Do" Nodes

While there are nodes in the "to do" list we need to:

- get the next node from the "to do" list;
- visit that node.

Visit all to do nodes

3.2.1 (vii): Visit all the “to do” nodes

```
private void traverseToDoList() {
    while (toDo.size() > 0) {
        T node = toDo.remove();
        visitNode(node);
    }
}
```

Visit all nodes

3.2.1 (viii): The Complete Traversal

```
private Queue<T> toDo = new ArrayDeque<T>();
private List<T> traversal = new ArrayList<T>();

public List<T> traverse() {
    T next = getUnvisitedNode();
    while (next != null) {
        toDo.add(next);
        traverseToDoList();
        next = getUnvisitedNode();
    }
    return traversal;
}
```

Finding the next unvisited node

3.2.1 (ix): Finding the next unvisited node

```
protected T getUnvisitedNode() {
    for (T node:  getNodes()) {
        if (!isOnVisitList(node)) {
            return node;
        }
    }
    return null;
}
```

Depth First Traversal

3.2.2: Depth First

A visit to a node should:

- check whether the node has already been visited
- visit the node
- visit all the node's descendants

Implementation: left as an exercise

End of graphs lecture