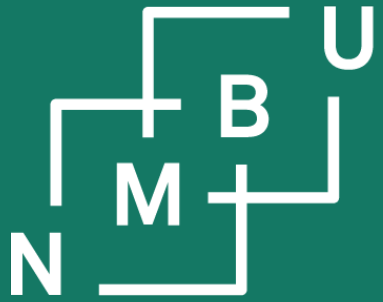
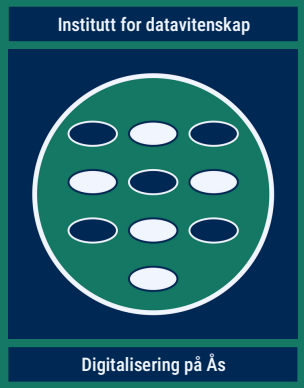
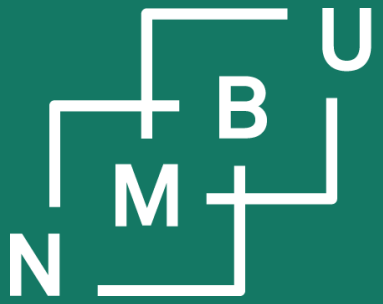


Norges miljø- og biovitenskapelige universitet

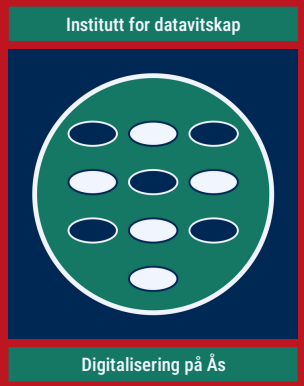


INF205
Resource-efficient programming

3 Data structures and libraries



Noregs miljø- og biovitskaplege universitet



3 Data structures

3.5 Linked data structures

3.6 Graphs and trees

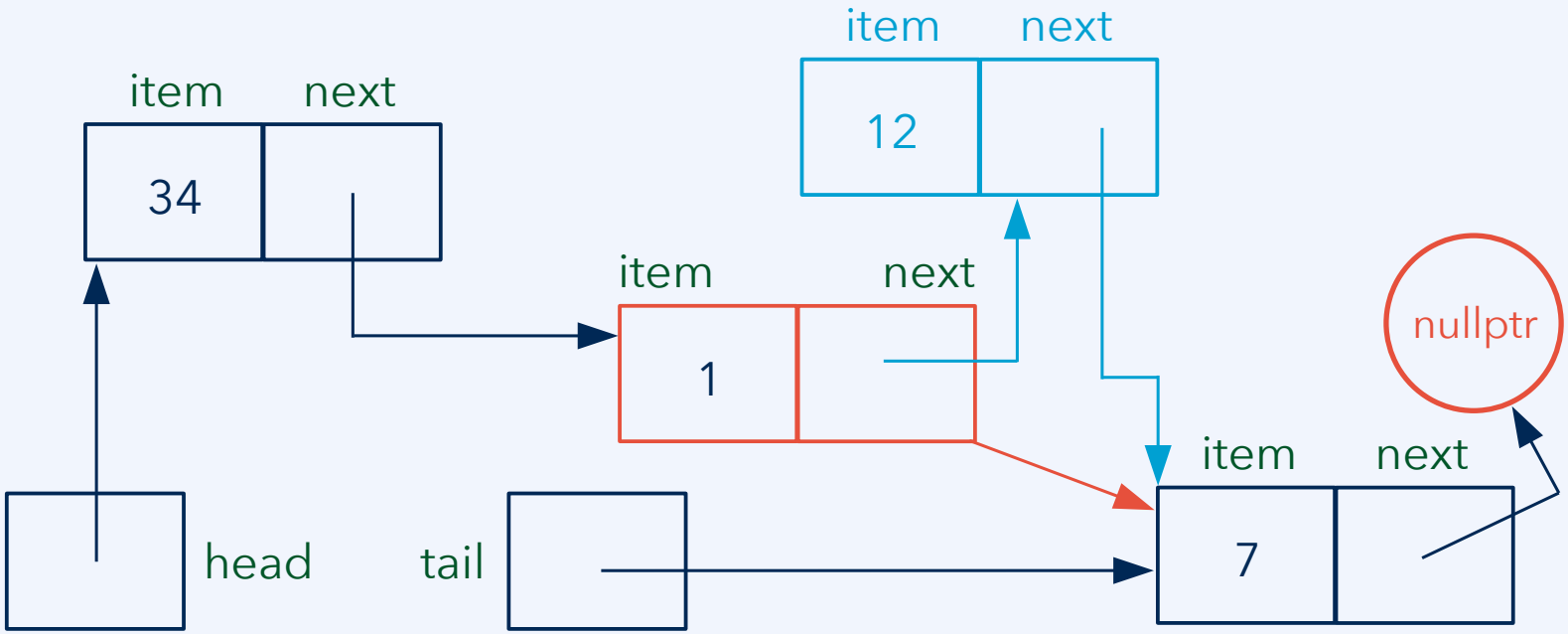
INF205

26th February 2025

Linked lists (singly linked)

Linked lists are **dynamic data structures**. Their elements are **not contiguous in memory**. Therefore, pointer arithmetics and increments (p++) cannot be used. Instead, the linked list consists of **nodes**.

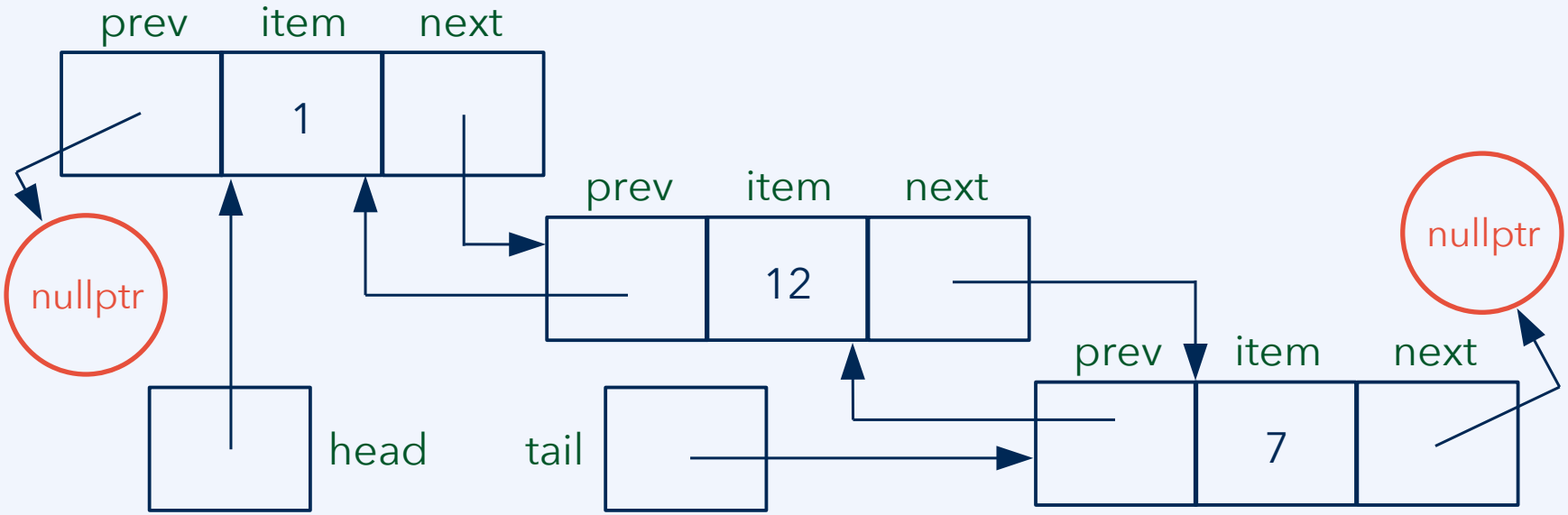
Example task: Insert 12 after node x, to which we already have a reference.



Linked lists (doubly linked)

In a **doubly linked list**, each node also contains a reference (or pointer) to the **previous node**. This facilitates traversal in **both directions and inserting** a new data item **before** any given node (rather than only after it), all in constant time.

Singly linked lists require two variables per data item (item and next).
Doubly linked lists require three variables per data item (prev, item, and next).

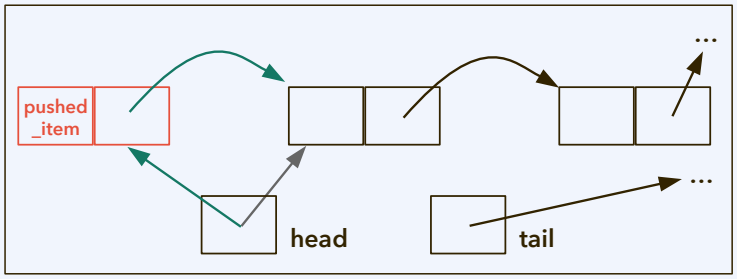


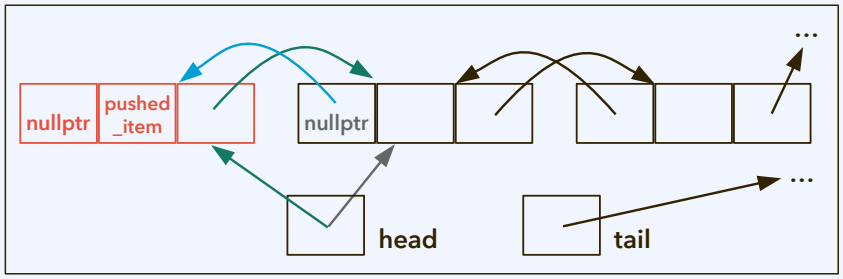
From singly to doubly linked

```
// add an item at the beginning of the list
void SinglyLinkedList::push_front(
  const int& pushed_item
){
  SinglyLinkedListNode* new_node
                  = new SinglyLinkedListNode;
  new_node->set_item(pushed_item);

  if(this->empty()) this->tail = new_node;
  else
    new_node->set_next(this->head);



  this->head = new_node;
}
```

For every link forward (next), there is now also a link backward (prev).

```
// add an item at the beginning of the list
void DoublyLinkedList::push_front(
  const int& pushed_item
){
  DoublyLinkedListNode* new_node
                  = new DoublyLinkedListNode;
  new_node->set_item(pushed_item);

  if(this->empty()) this->tail = new_node;
  else {
    new_node->set_next(this->head);
    this->head->set_prev(new_node);
  }
  this->head = new_node;
}
```





Discussion: Deleting or copying a list

Say we have a **linked-list** object x, consisting of the fields “head” and “tail”.

- The object runs out of scope and is getting deallocated. We did not define a constructor, so the default constructor is used.
  - What will this do? Is it what we want?
- We assign the value of x to another list object y, writing “y = x”. By default, this means that the property values “head” and “tail” are copied.
  - Is this reasonable? What could the user be expecting instead?

Or: We have a **singly-linked-list node** object, consisting of “item” and “next”.

- What will happen upon deallocation by default? What should happen?
- What will happen upon copying by default? What should happen?

Overview: Sequential data structures

copying-and-moving.zip

- **Read/write** access to a data item at position k
  - For a dynamic array, O(1) time; fast access by pointer arithmetics
  - For a singly linked list, O(k) time, *i.e.*, O(n) in the average/worst case
  - For a doubly linked list, O(min(k, $n - k$)), which is still effectively O(n)

- **Iterating** over the data, *i.e.*, proceeding from one item to the next one
  - O(1) both for dynamic arrays and for linked lists

- **Deleting** a data item at position k
  - For a dynamic array, O(1) at the end, O($n - k$) in general
  - For a singly linked list, O(1) at the head, or if we have a reference to the element at position k–1; otherwise, in general, O(k)
  - For a doubly linked list, O(1) at the head or tail, or if we have a reference to that region of the list; in general, O(min(k, $n - k$))

Remark: For linked lists, insertion/deletion as such takes constant time, once the node has been localized. However, getting to the node can take O(n) time.

Overview: Sequential data structures

copying-and-moving.zip

- **Read/write** access to a data item at position k
  - For a dynamic array, O(1) time; fast access by pointer arithmetics
  - For a singly linked list, O(k) time, *i.e.*, O(n) in the average/worst case
  - For a doubly linked list, O(min(k, $n - k$)), which is still effectively O(n)

- **Iterating** over the data, *i.e.*, proceeding from one item to the next one
  - O(1) both for dynamic arrays and for linked lists

- **Inserting** an additional data item at position k
  - For a dynamic array, O(n) in the worst case, *i.e.*, whenever the capacity is exhausted; with free capacity, O(1) at the end, O($n - k$) elsewhere
  - For a singly linked list, O(1) at the head or tail, or if we have a reference to the element at position k–1; Otherwise, in general, O(k)
  - For a doubly linked list, O(1) at the head or tail, or if we have a reference to that region of the list; in general, O(min(k, $n - k$))

Remark: For linked lists, insertion/deletion as such takes constant time, once the node has been localized. However, getting to the node can take O(n) time.

Stacks and queues

Example: **copying-and-moving.zip**

Queue

- nn, nb kø *m.*

Definition: A queue is a sequential (list-like) dynamic data structure that functions by the principle *first in, first out (FIFO)*.

- A queue class must provide a method for appending an element, usually called ***push*** and done on one end of the queue (*e.g.*, ***enqueue*** or ***push_back***), and a method for detaching an element, usually called ***pop*** and done at the other end of the queue (*e.g.*, ***dequeue*** or ***pop_front***).
- Singly and doubly linked lists are well suitable for implementing a queue, since both ***push*** and ***pop*** can be realized in constant time. However, a singly linked list should only be used if the data structure includes an explicit reference to the tail node; otherwise, the whole list needs to be traversed just to reach the tail, taking O(*n*) time.
- Dynamic arrays are less suitable for this purpose, requiring O(*n*) time for the ***push*** and ***pop*** operations in the long run (as their capacity gets exhausted).

- **Queues** function by the principle **"first in, first out" (FIFO)**
  - Can be implemented using a singly linked list (with a tail reference):
    - Attach (enqueue / push) new elements at the tail of the list only
    - Detach (dequeue / pop) elements from the head of the list only

Stacks and queues

Example: **copying-and-moving.zip**

- **Stacks** function by the principle **"last in, first out" (LIFO)**
  - Can be implemented using a singly linked list:
    - Attach (push) new elements at the head of the list only
    - Detach (pop) elements from the head of the list only
  - Can be implemented using a dynamic array:
    - Attach (push) new elements at the end of the array only
    - Detach (pop) elements from the end of the array only
- **Queues** function by the principle **"first in, first out" (FIFO)**
  - Can be implemented using a singly linked list (with a tail reference):
    - Attach (enqueue / push) new elements at the tail of the list only
    - Detach (dequeue / pop) elements from the head of the list only

All these operations can be carried out in constant time;
in case of the push operation for the dynamic array, subject to capacity.

Linked data structures as containers

Example: **copying-and-moving.zip**

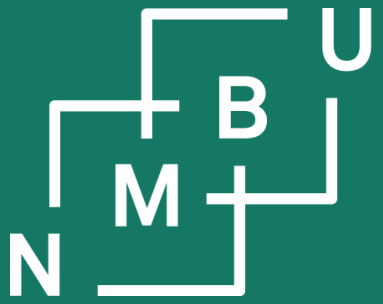
Container

- nn konteinar, container *m.*
- nb konteiner, container *m.*

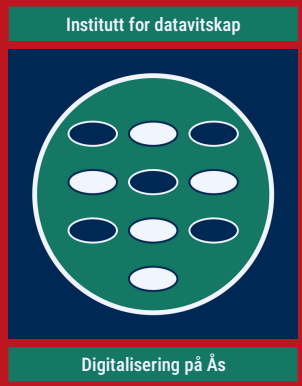
Definition (Stroustrup): "A class with the main purpose of holding objects is commonly called a *container*."

- Container objects take *ownership*, *i.e.*, responsibility for allocating and deallocating any contained data. The programmer needs to take care of this whenever there are data subject to manual memory management (new and delete) in a self-designed container.
- Examples include the standard template library (STL) containers (list, map, set, vector, *etc.*). Other than for educational purposes as an exercise, it does not make sense to reimplement these standard data structures by hand.
- Many problems require special, tailored container data structures in order to be solved efficiently. It is then part of the development work to both design and implement the required data structure.

See also: Object-oriented programming, pointer, queue.



Noregs miljø- og biovitskaplege universitet



3 Data structures

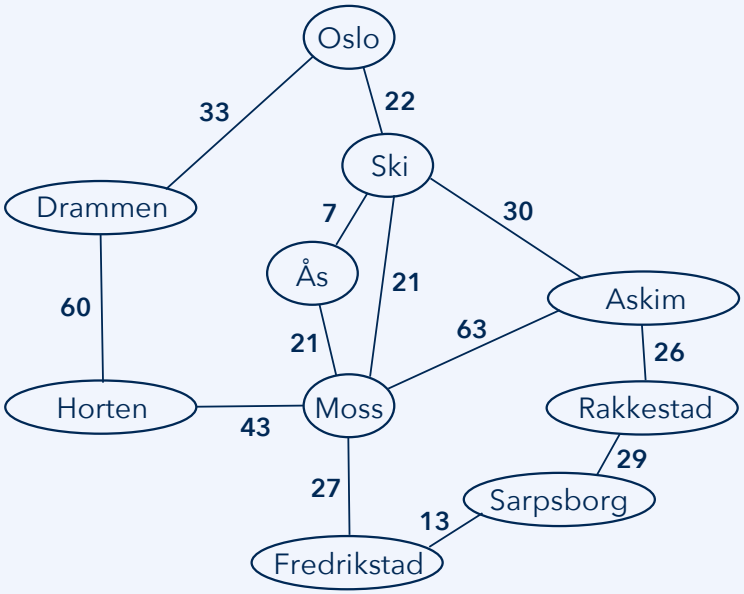
3.5 Linked data structures

3.6 Graphs and trees

INF205

26th February 2025

Graphs as non-sequential linked data structures



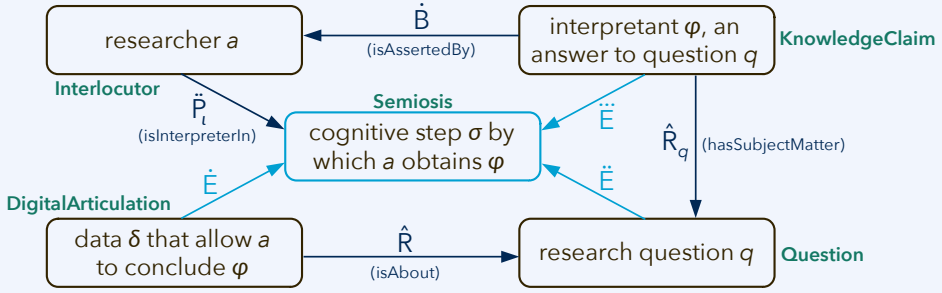
Sequential data structures arrange their items in a linear shape. Sometimes that is not the best solution, or it is not appropriate at all.

Linked data structures with a non-sequential shape are **graphs**, which includes the important special case of tree data structures.

A graph G = (V, E) is defined by its **nodes V**, which are also called vertices, and **edges E** that connect one node to another. Nodes and edges can be *labelled* to give the graph a meaning.

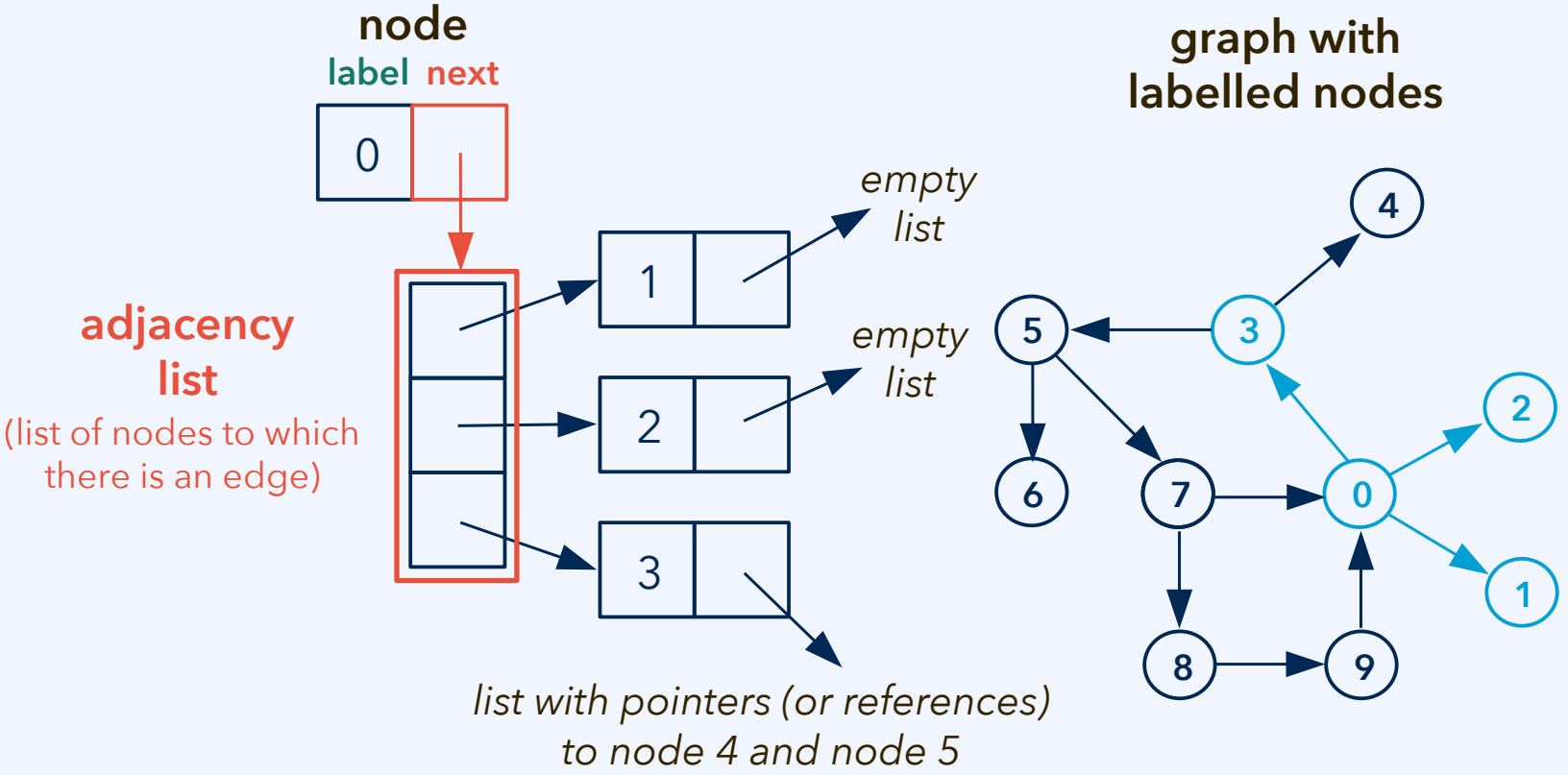
Graphs can be used to represent relations between objects, such as distances on a map, or as a **knowledge graph**.

Trees are often used as sorted data structures, for efficiency reasons.



Implementation: Adjacency lists

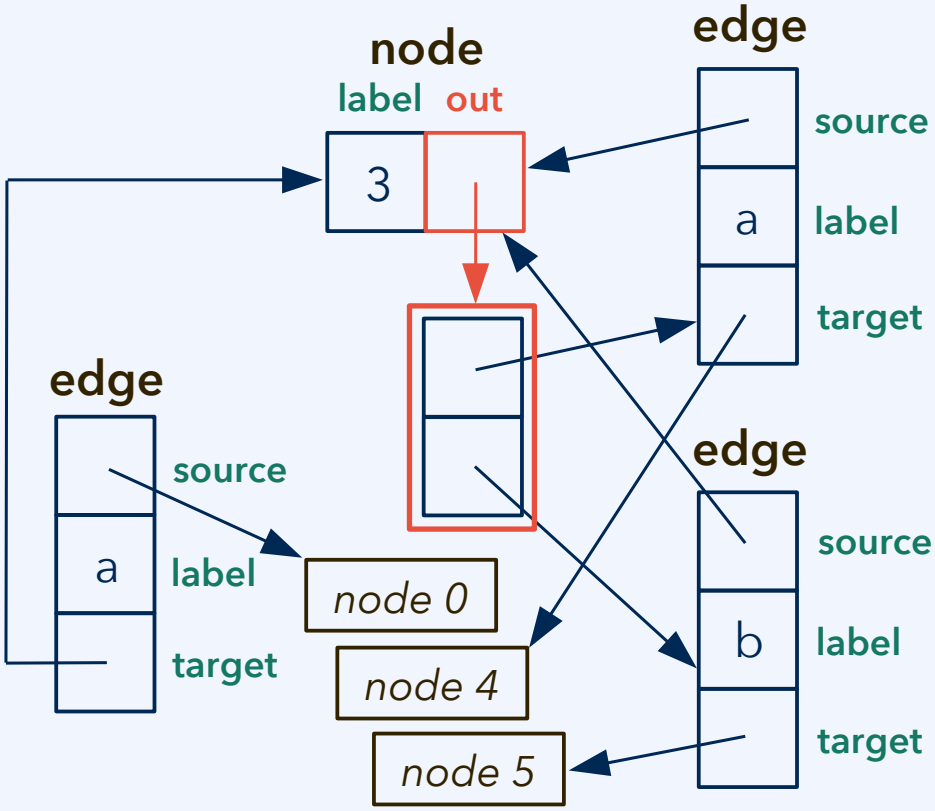
In a graph, one node can be connected to multiple other nodes. An **adjacency list** (with various possible implementations) can be used to manage these links.

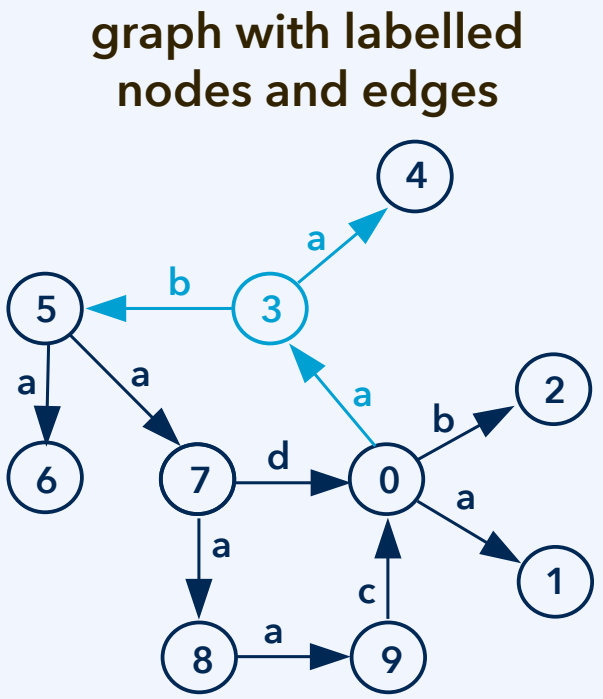


Doubly-linked version of this: Two lists, for incoming and for outgoing edges.

Implementation: Incidence lists

An **incidence list** is a list of edges to which a node is incident. For adjacency lists or incidence lists, various data structures can be used, *e.g.*, dynamic arrays.

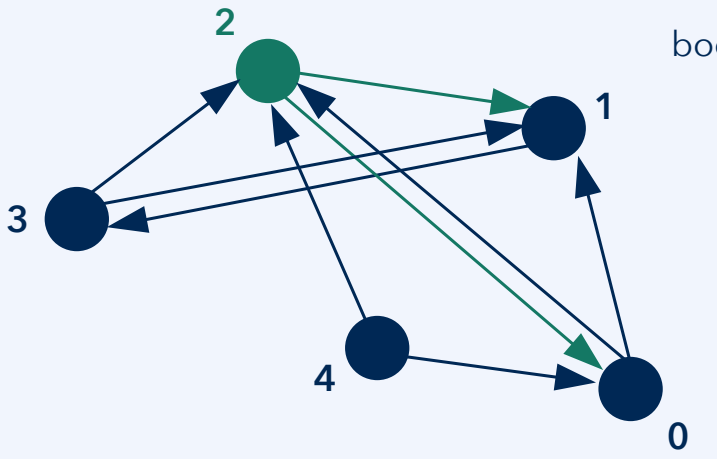




Doubly-linked version of this: Two lists, for incoming and for outgoing edges.

Implementation: Adjacency matrix

Matrix-like data structures include two-dimensional arrays, *i.e.*, arrays where the individual elements are accessed by double indexing. The most relevant use for graphs is the **adjacency matrix**. (Also possible: An incidence matrix.)



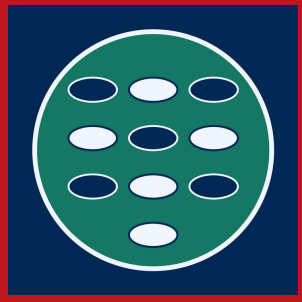

```
bool adj[5][5]={   {true, true,  true,  false, false},
                   {false,false, false, true,  false},
                   {true, true,  false, false, false},
                   {false,true,  true,  false, false},
                   {true, false, true,  false, false}  };
```

out of node 0
out of node 1
out of node 2
out of node 3
out of node 4

For a sparse graph, the vast majority of entries in the 2D array/matrix is “false”. Adjacency matrices are commonly only used when expecting a **dense graph**.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

Knowledge graphs: Use of graph structures for knowledge representation

INF205

26th February 2025

Knowledge bases: TBox and ABox

A **knowledge base** for linked data consists of two components:

> **Definition:** A **knowledge base**, given by $K = (T, A)$, consists of an ontology T, describing universals, and a set of assertions A describing concrete instances of these universals.

ABox = knowledge graph

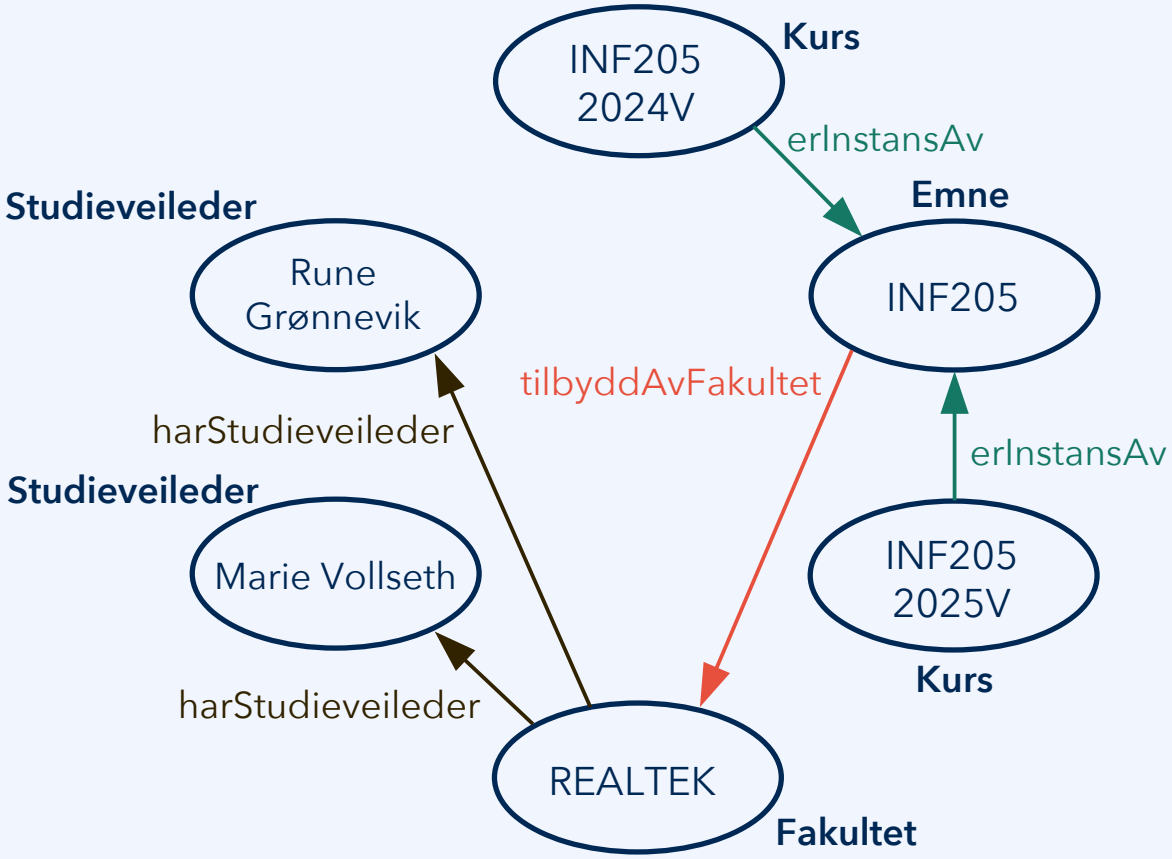
	entity		(sometimes: attribute)
particular:	**individual** (object)	**relationship**	**property**

	entity type	relationship type	(sometimes: attribute type)
universal:	**concept** (class)	**relation** (in OWL: ObjectProperty)	**attribute** (in OWL: DatatypeProperty)

TBox = ontology

Semantics represented as a graph

Modern knowledge bases represent knowledge about the state of affairs as **knowledge graphs**. These graphs are understood as part of one **semantic web**.



Knowledge graphs contain **individuals** (objects) as nodes.

They contain **relations** (binary predicates) as edges.

They may also visually represent the instantiation of **concepts** (classes).

Querying graph databases using SPARQL

SPARQL is a recursive acronym: "SPARQL Protocol and RDF Query Language." An interface that can handle SPARQL queries is called a *SPARQL end point*.

The **syntax of SPARQL** is reminiscent of SQL, but at its core is given by

- **RDF triples**, using RDF schema and OWL, in **TTL notation**;
- Some elements of the triples are **wildcards**, *i.e.*, free variables.

```
SELECT ?x ?y
WHERE {
  ?x erInstansAv ?emnekode.
  ?emnekode tilbyddAvFakultet ?fakultet.
  ?fakultet harStudieveilder ?y.
}
```

"Produce a table with course instances in the first column and the study advisors from the faculty responsible for the course in the second colum."

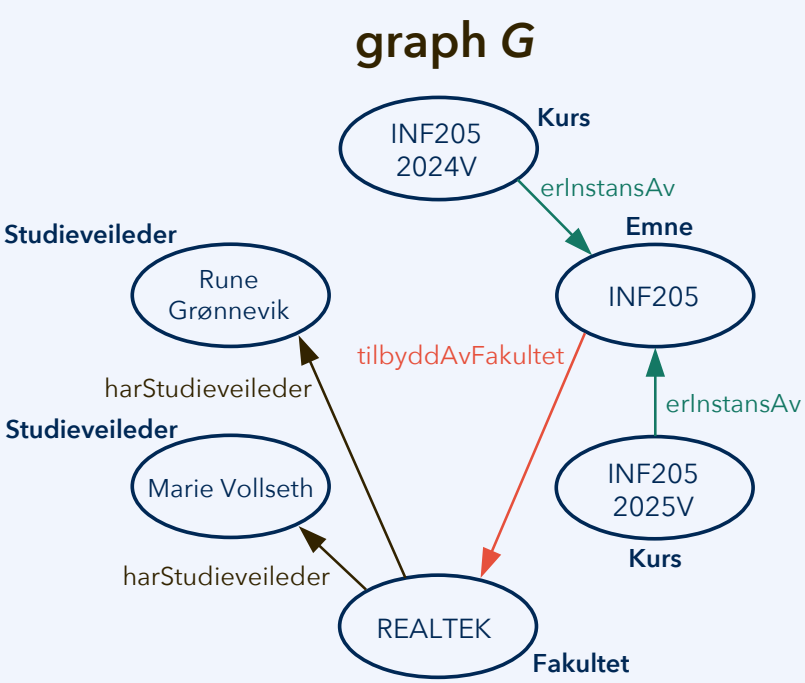
The **semantics of SPARQL** is given by the correct response to a SPARQL query, which consists of a *table with all matching valuations of the selected wildcards*.

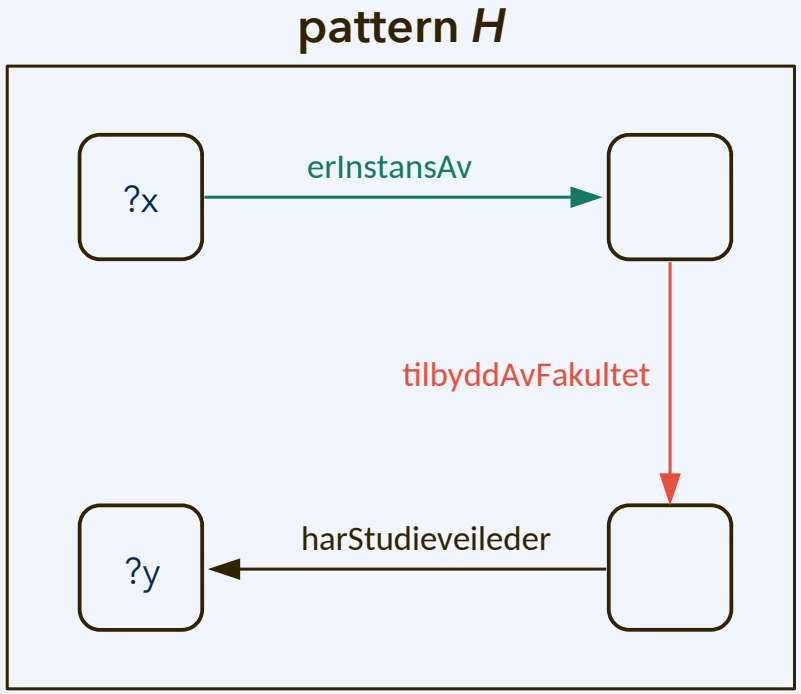
SPARQL querying therefore corresponds to the **subgraph matching problem** from graph theory: It looks for occurrences of a pattern within a larger graph.

Querying graph databases

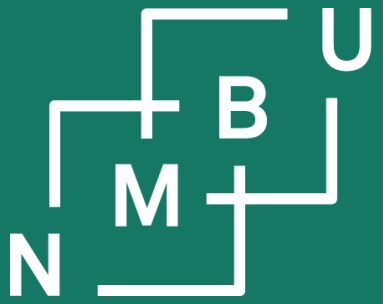
Subgraph matching problem (NP-complete):

Given a graph *G* and a pattern *H*, does G contain a subgraph isomorphic to *H*?

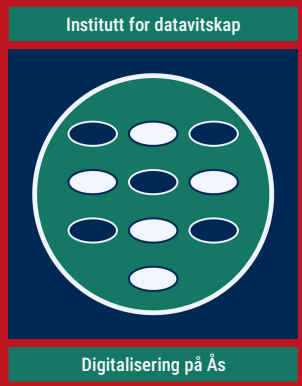




SPARQL querying therefore corresponds to the **subgraph matching problem** from graph theory: It looks for occurrences of a pattern within a larger graph.



Noregs miljø- og biovitskaplege universitet



Trees: Acyclic graphs

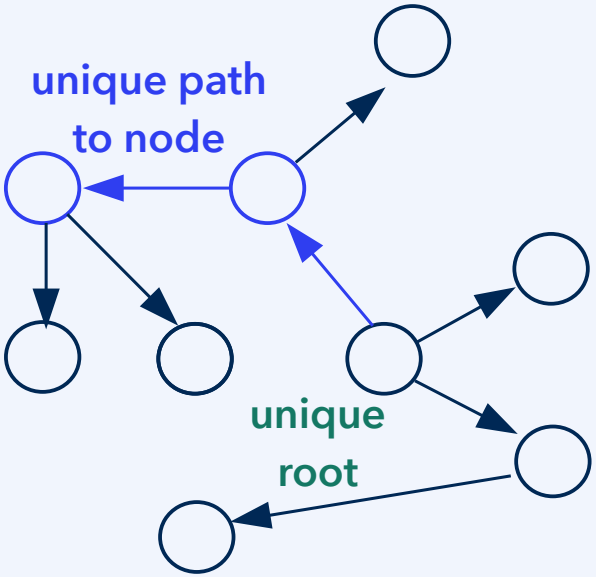
INF205

26th February 2025

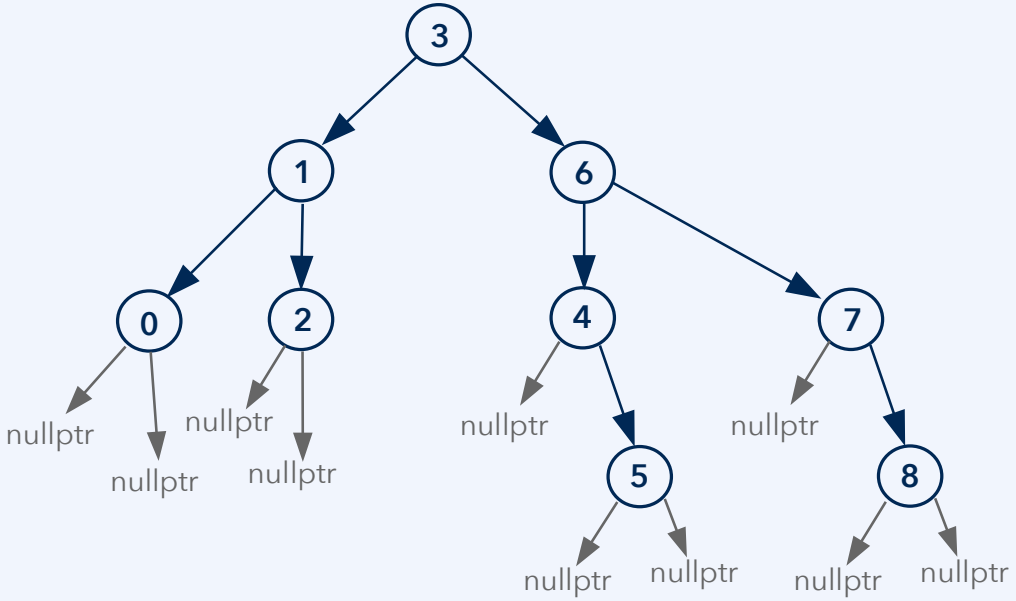
Tree data structures

Trees are a special kind of graph; or graphs are a generalization of trees:

tree (a kind of graph)



a binary search tree



Definition ("tree"; in the literature, also: "out-tree" or "rooted tree")

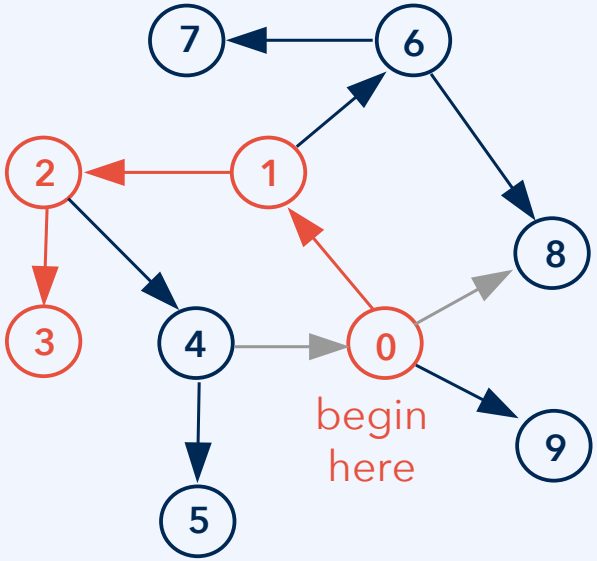
A tree is a graph with a root and a unique path from the root to each node.

Graph traversal

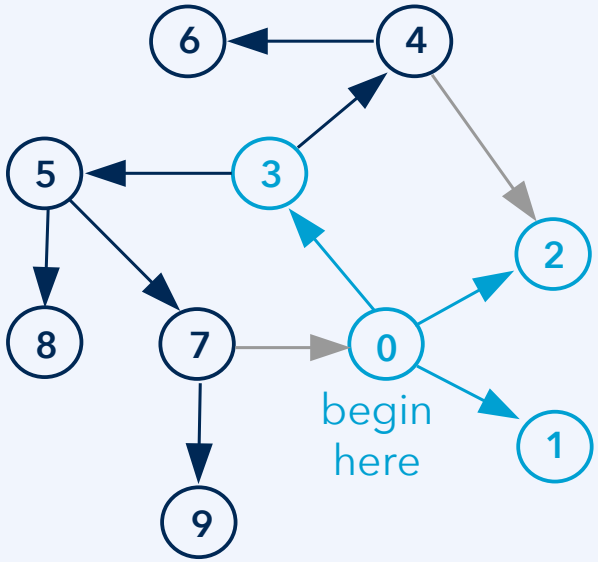
Traversal of graphs: Depth-first search and breadth-first search

DFS always proceeds from the most recently detected node (LIFO).
BFS always proceeds from the node that was detected earliest (FIFO).

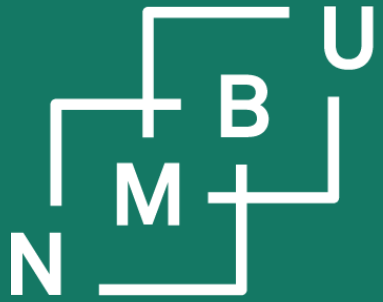
depth-first search (DFS)



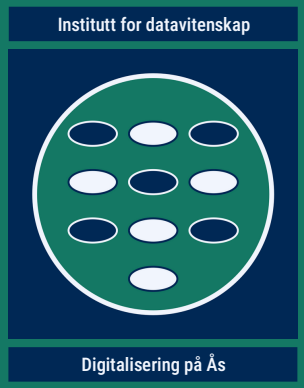
breadth-first search (BFS)



Note: Only elements *to which there is a path from the initial node* can be found.



Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

3 Data structures and libraries