Norges miljø- og biovitenskapelige universitet

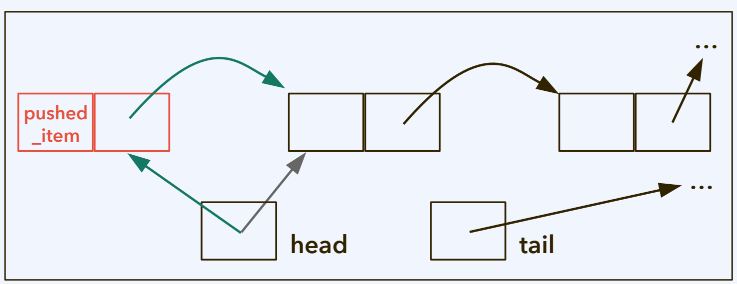
INF205
Resource-efficient programming

2 Data structures

2.5 Templates
2.6 Graph data structures
2.7 Tailored containers

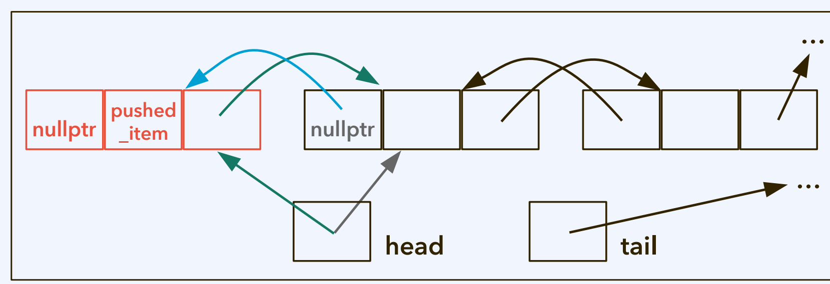
From singly linked to doubly linked list

```
// 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).

```
// see example code sequence-performance
// 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;
}
```



Thanks to Hanna Lye Moum and Nora Mikarlsen for fixing a bug in the original version.

Sequential data structures: Operations

- **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.

Application: Stacks and queues

- **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 (push) new elements at the tail of the list only
    - Detach (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 free capacity.

Noregs miljø- og biovitskaplege universitet

2 Data structures

2.5 Templates

C++ standard template library

The standard template library (STL) provides typical **container** data structures. They are **templates**: They can contain any type of fundamental data items or objects as their elements. The **element type** is specified in angular brackets.

```
// declare a list of int values
std::list<int> my_list();
```

```
// declare a list of BookIndex objects
std::list<BookIndex> my_list();
```

- **vector**<**T**> is a **dynamic array** for type **T** elements. (Free capacity: Tail only.)
- **deque**<**T**> ("double ended queue"): **Dynamic array** with capacity both ends.

- **forward_list**<**T**> is a **singly linked list** data structure for type **T**.
- **list**<**T**> is a **doubly linked list** data structure for type **T**.

- **set**<**T**> is a container where each **key** (element) occurs only (at most) once.
- **map**<**T**, **V**> contains **key**-**value** pairs, which each key occurring at most once.
- **multimap**<**T**, **V**> contains **key**-**value** pairs; keys may occur multiple times.

- **array**<**T**, **n**> is a **static array** for type **T**, with array size **n**, similar to **T[]** arrays.

Parameterized class definitions

We have already seen the STL templates: The **same container implementation** can be used for **different types of contained objects**, such as **list<float>** and **list<double>**. We can define our own class templates in this way:

```
template<typename T> class SinglyLinkedListNode
{
public:
  T& get_item() { return this->item; }
  SinglyLinkedListNode<T>* get_next() const { return this->next; }
  void set_item(T in_item) { this->item = in_item; }

private:
  T item;
  SinglyLinkedListNode<T>* next = nullptr;
  void set_next(SinglyLinkedListNode<T>* in_next) { this->next = in_next; }
};
```

attention with split between header and object file; think about "what the compiler will do"

attention with initializations

(example code **list-template**)

While there is only one **source code** for each template, **object code** is normally generated separately for each concrete version of it. (But not for the template!)

Templates for functions and methods

Norwegian University of Life Sciences

The same sort of syntax applies for parameterized function and method declarations and definitions. This includes cases with multiple parameters.

```
template<typename T>
  void SinglyLinkedList<T>::push_front(
    const T& pushed_item
) {
  SinglyLinkedListNode<T>* new_node
             = new SinglyLinkedListNode<T>;
  new_node->set_item(pushed_item);

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

Code above:
From **list-template** example.

```
template<typename SeqnT, typename ElmnT>
  void test_sequence(
    SeqnT* sqn, int n, int m,
    ElmnT a, ElmnT b, ostream* os
) {
  ...
}

template<typename SeqnT, typename ElmnT>
  float test_with_time_measurement(
    SeqnT* sqn, int iterations, ElmnT a, ElmnT b
) {
    int sequence_length = 1000001;
    int deletions = 10;
    test_sequence(sqn, 100000, 10, a, b, &cout);
}
```

Case distinctions in templates

The standard library header <**type_traits**> includes parameterized flags that can be used to make case distinctions, *e.g.*,

- **is_arithmetic**<T>::value, **is_signed**<T>::value, *etc.*;
- **is_pointer**<T>::value, **is_class**<T>::value, **is_array**<T>::value, *etc.*;
- **is_same**<T, S>::value, to check whether T and S are the same type.

In **list-template**, solutions for initializing the property "**T** item" would include:

```
T item = T();
```

```
template<typename T>
    const T initial_value = T();
...
T item = initial_value<T>;
```

```
SinglyLinkedListNode<T>() {
   if constexpr(is_arithmetic<T>::value) this->item = 3;
   else if constexpr(is_pointer<T>::value) this->item == nullptr;
   else if constexpr(is_same<T, string>::value) this->item = "uninitialized";
   else this->item = T();
}
```

Only with the solution on the right we can make more high-level design distinctions depending on the nature of the type T used for parameterizing.

Generic programming

Norwegian University of Life Sciences

Extensive reliance on templates is also called **generic programming** (GP), which can be seen as its own programming paradigm, building on object-oriented programming but going beyond it; "by implementing programs generically, a single implementation can be used for many different types".[1]

We have seen: C++ supports such design by (1) inheritance and (2) templates.

From C++20 onward, **concepts** are introduced as GP language constructs. They describe requirements for a type (*e.g.*, it must provide an operator such as "<<", a particular method, or we must be able to add it to an integer, …).

```
// old style: does not make clear what
// we expect from the class SeqnT
template<typename SeqnT, ...>
  void test_sequence(SeqnT* sqn, ...)
{ ... }
```

```
// new style, where we would define Sequence
// as a concept (and not as an abstract class)
template<Sequence SeqnT, ...>
  void test_sequence(SeqnT* sqn, ...)
{ ... }
```

[1]L. Escot, J. Cockx, Proc. *ACM Prog. Lang.* **6**: 625–649, doi:10.1145/3547644, **2022**.

Noregs miljø- og biovitskaplege universitet

2 Data structures

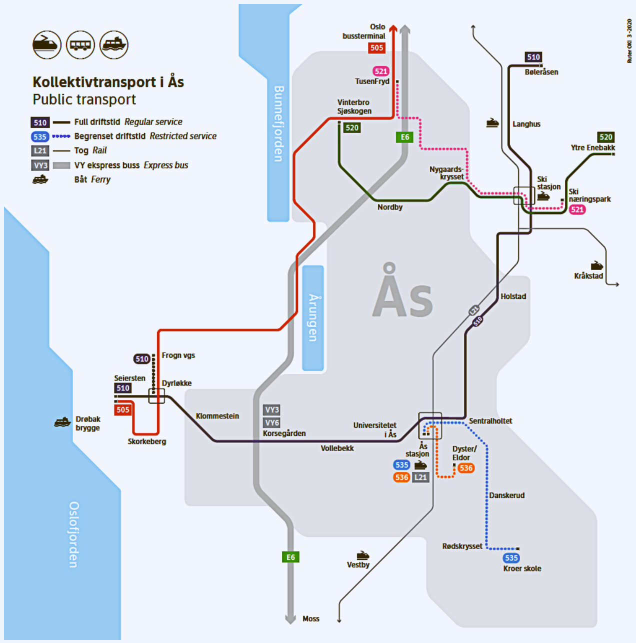
2.5 Templates

2.6 Graph data structures

INF205

12th October 2022

Non-sequential containers



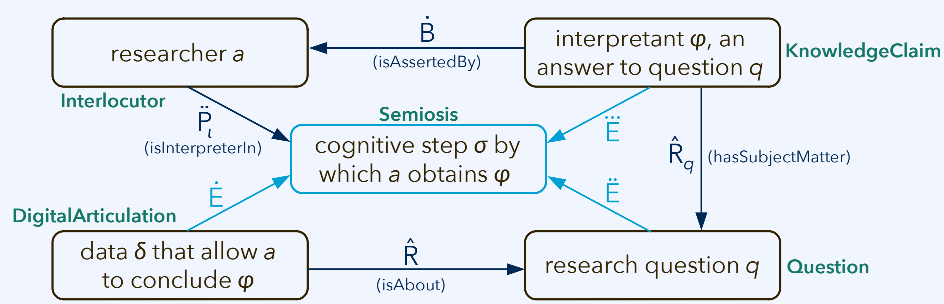
Sequential data structures arrange their items in a linear shape. Sometimes that is not the best solution, or it is not appropriate at all. **Why?**

The most frequent container data structures with a different, non-sequential shape are **graphs**, including 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 may also be labelled in order 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**.

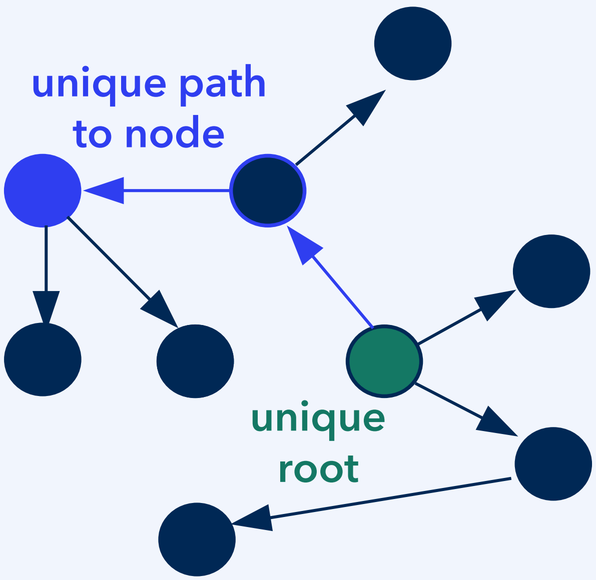
Trees are often used as sorted data structures, for efficiency reasons.



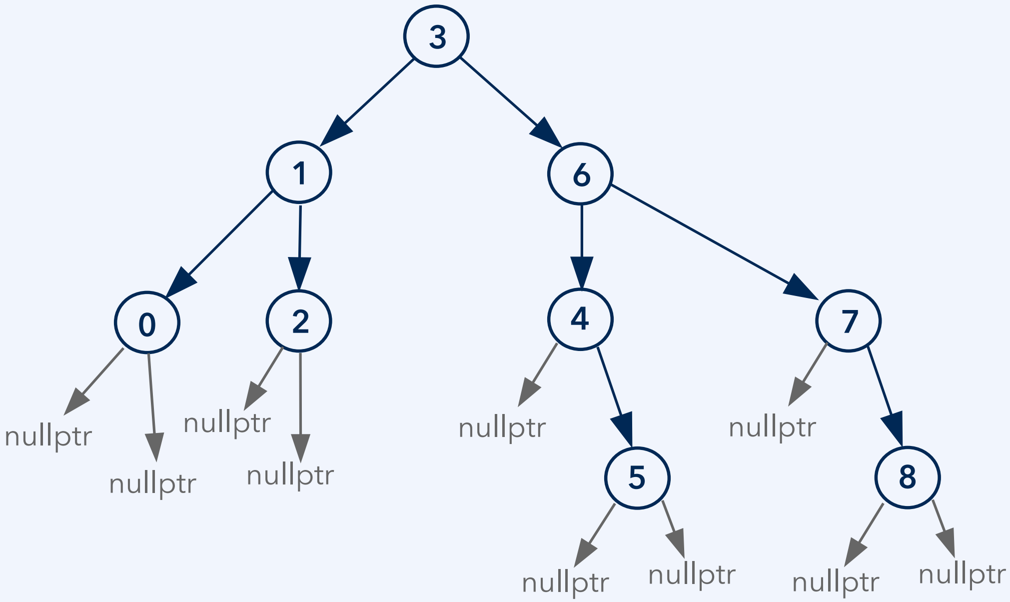
Tree data structures

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

tree (a kind of graph)



a binary search tree

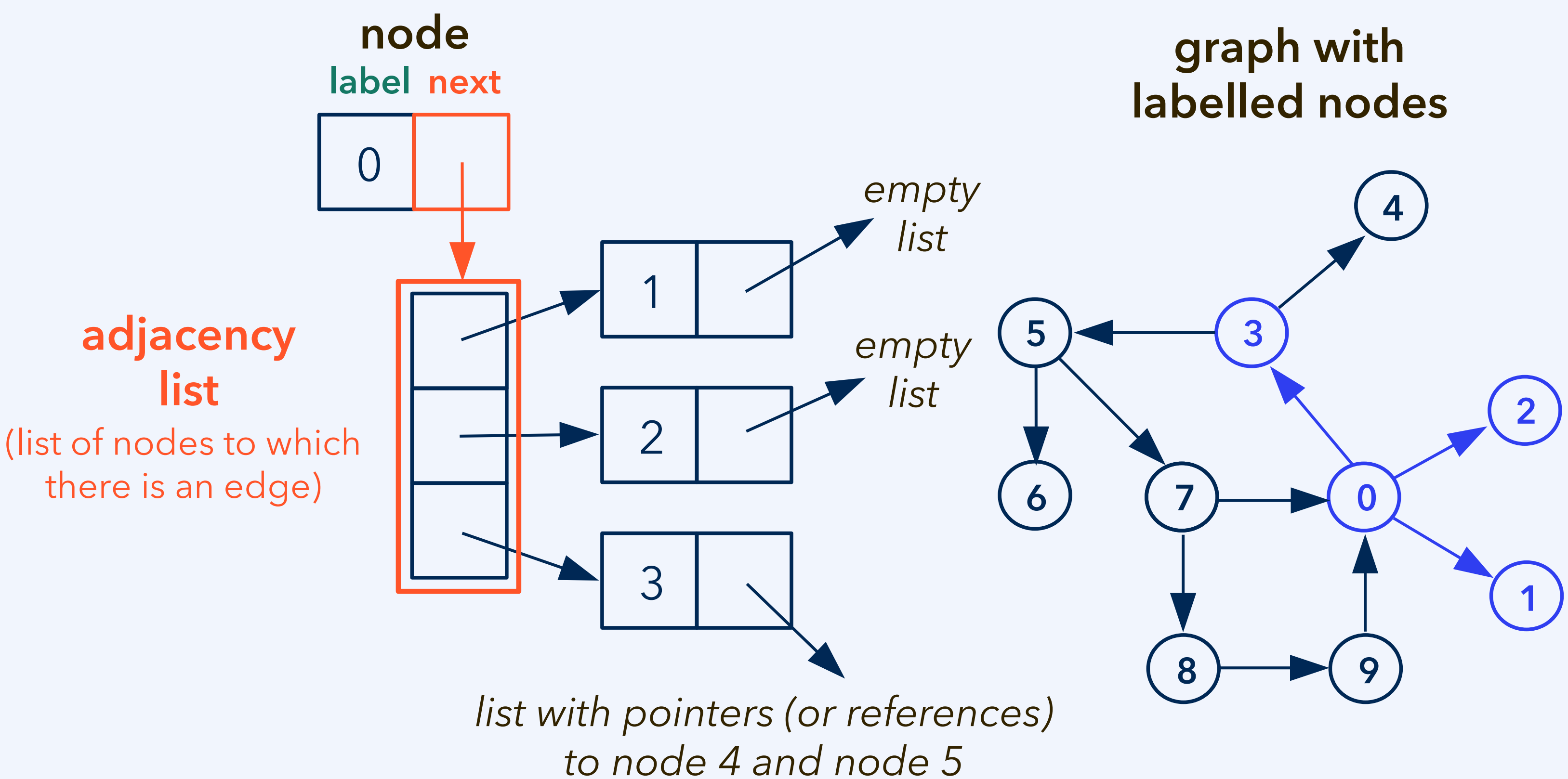


Definition ("tree"; in the literature, also: "out-tree" or "rooted tree")

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

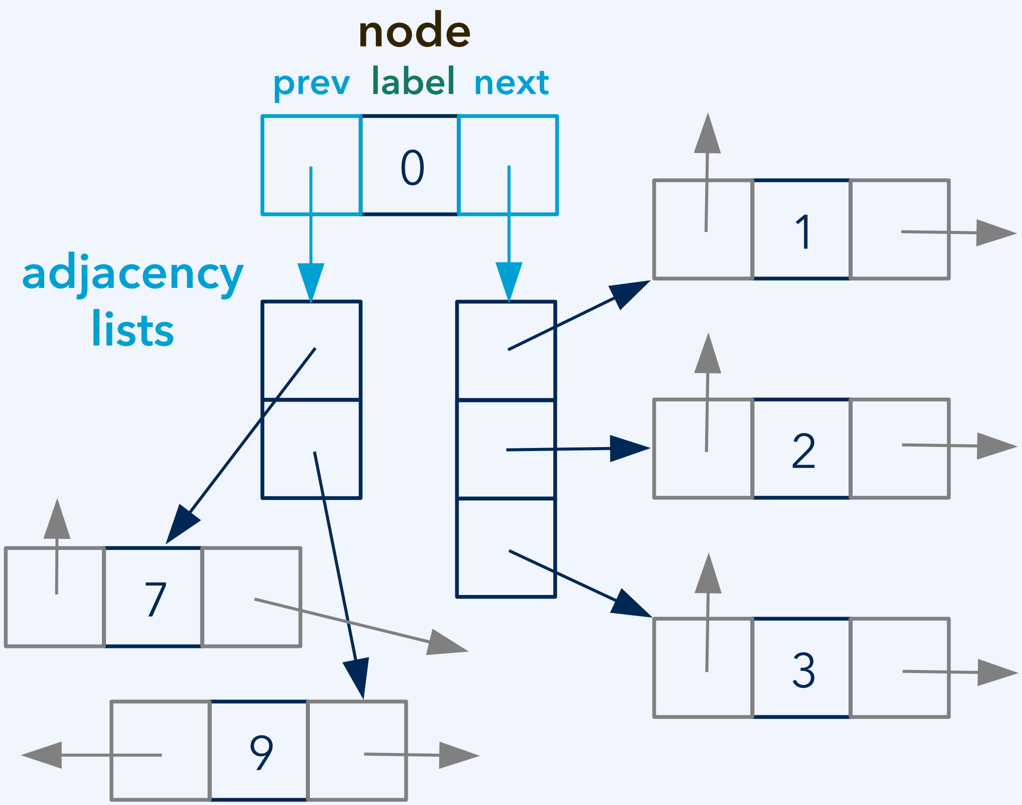
Adjacency lists: Singly linked

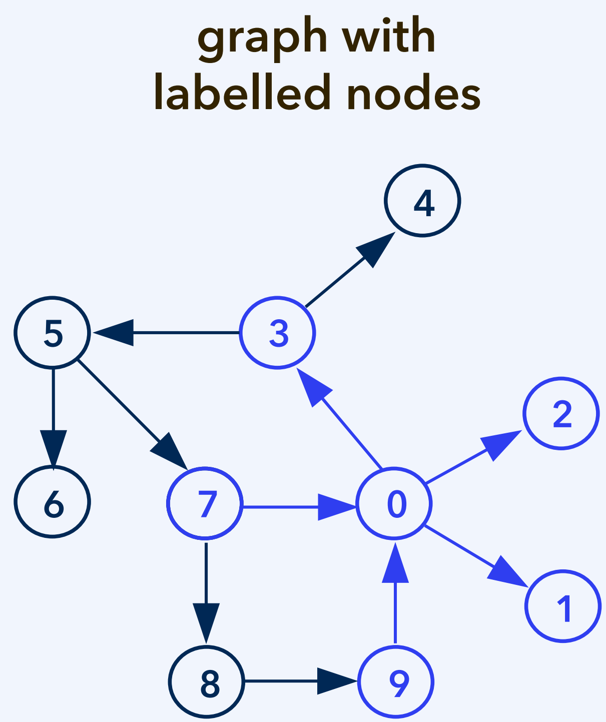
In a graph, one node can be connected to multiple other nodes. An **adjacency list** (with various possible implementations) can be used to manage these links.



Adjacency lists: Doubly linked

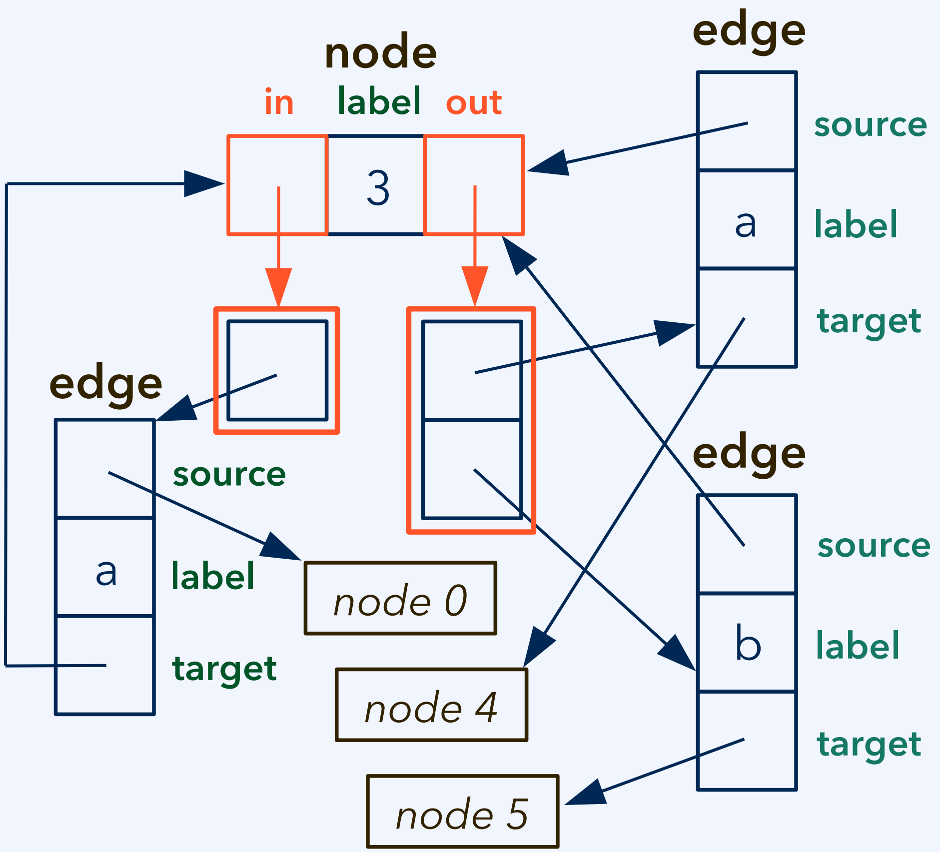
Instead of singly linked data structures, **doubly linked** data structures can also be used; *e.g.*, with an additional adjacency list pointing to predecessor nodes.

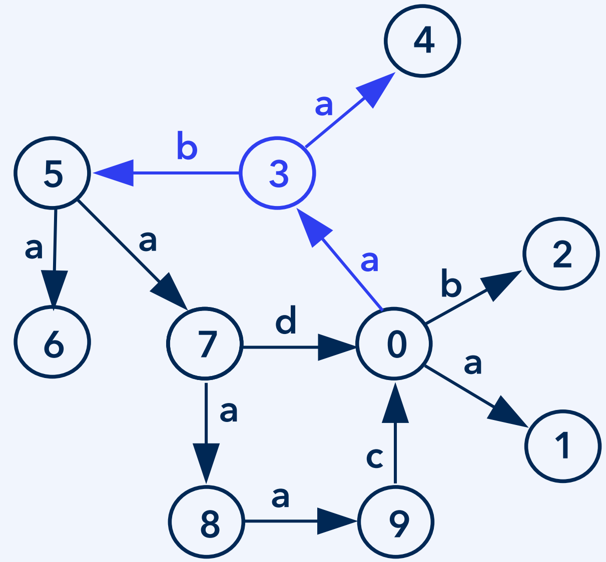




Incidence lists

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.



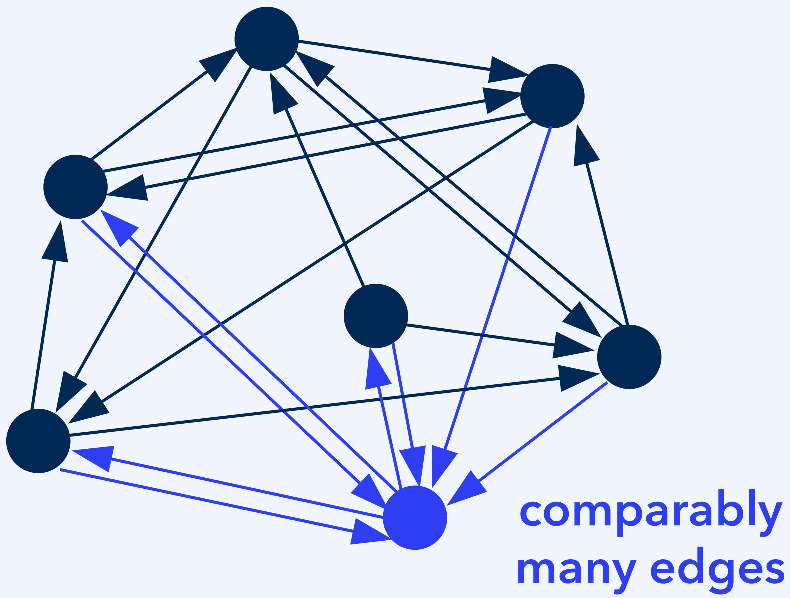


Sparse graphs vs. dense graphs

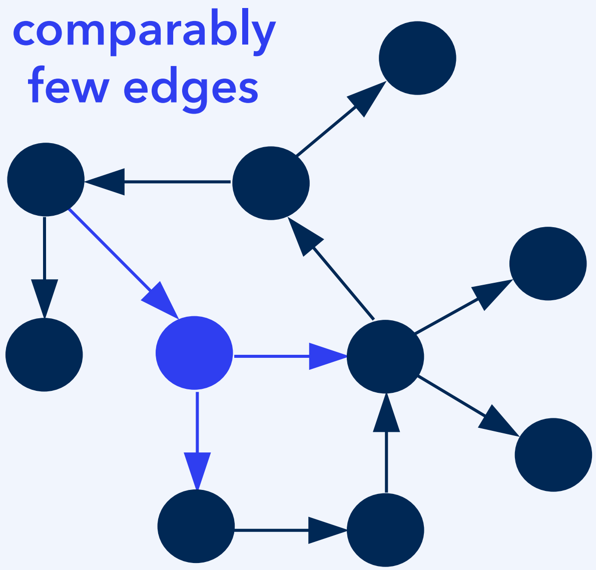
Norwegian University of Life Sciences

Neighbour lists, implemented as **adjacency or incidence lists**, are most suitable for **sparse graphs**. Matrix-like data structures are best for **dense graphs**.

dense graphs

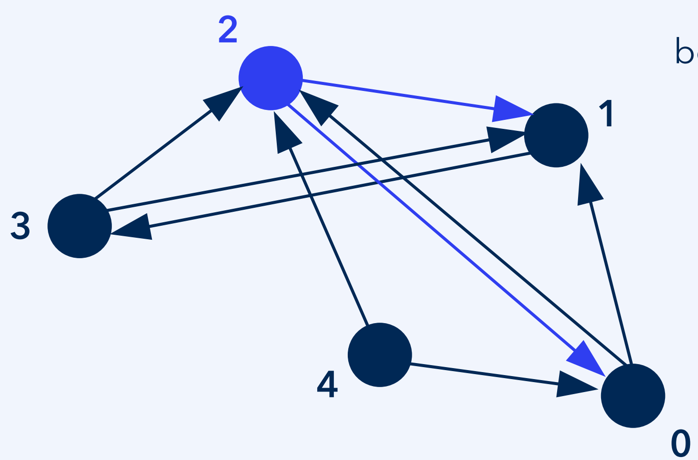


sparse graphs



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},	**out of node 0**
{false,false, false, true, false},	**out of node 1**
{true, true, false, false, false},	**out of node 2**
{false,true, true, false, false},	**out of node 3**
{true, false, true, false, false} };	**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**.

What are typical problems for graphs?

Most important computational problems for graph data structures:

Traversal of the nodes in the graph, including searching. Two canonical ways:

- **Depth first search** (DFS), always goes into depth as far as possible:
  - Push edges to a **stack**; pull from stack to visit the next node.
- **Breadth first search** (BFS), visits nodes in the order they are detected:
  - Push edges to a **queue**; pull from stack to visit the next node.

Reduction to a tree with a given root node (**spanning tree**), for example, using DFS or BFS for a "depth-first" or "breadth-first" tree. Also the **shortest paths**, if nodes have different "distances" from each other (edge weights).

Looking for **paths in a graph**: This includes cycles (same start and end node), the shortest paths (see above), or Hamilton paths/cycles (once at every node).

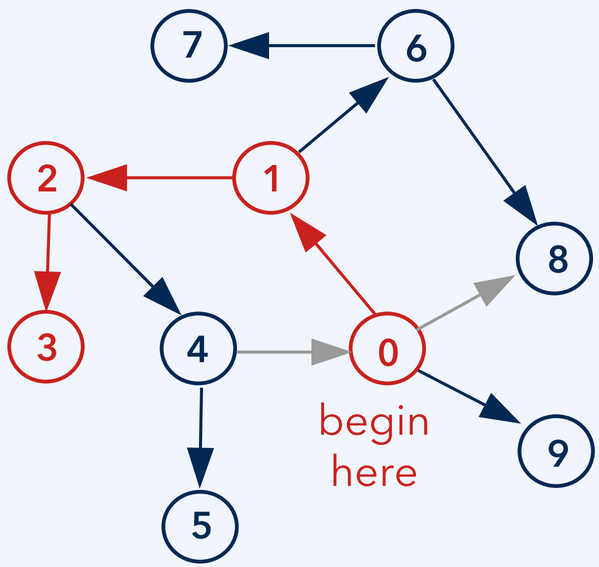
Related: **Connected components** and strongly connected components.

Graph traversal and spanning trees

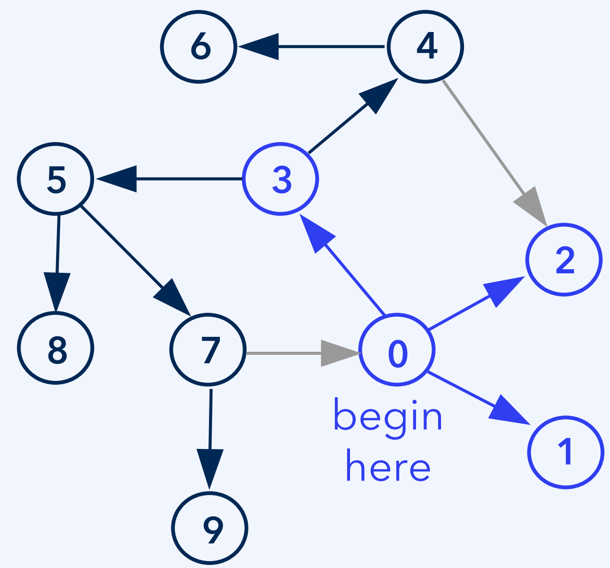
Traversal of trees and graphs: Depth-first search and breadth-first search

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)

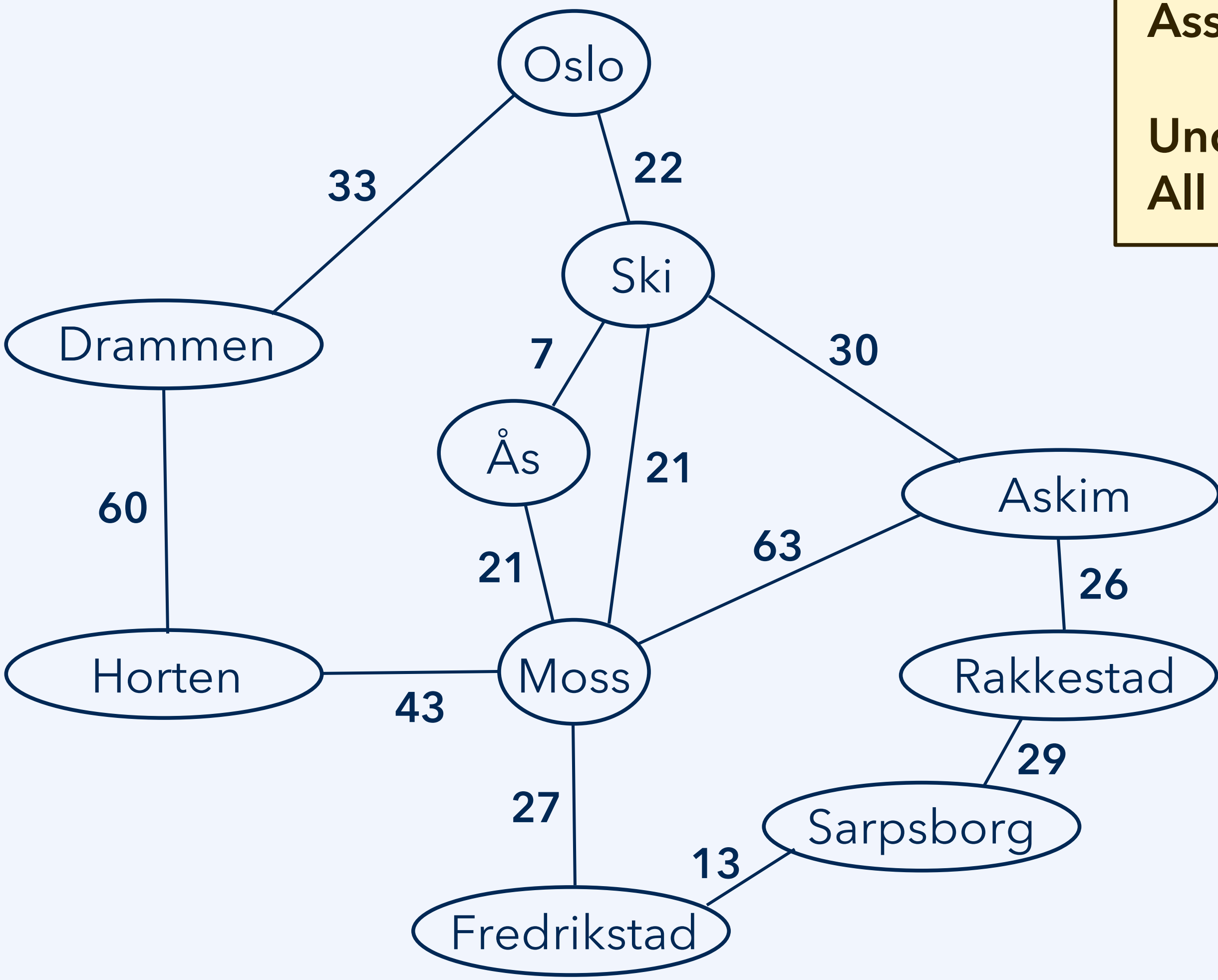


breadth-first search (BFS)



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

Shortest paths: Dijkstra's algorithm



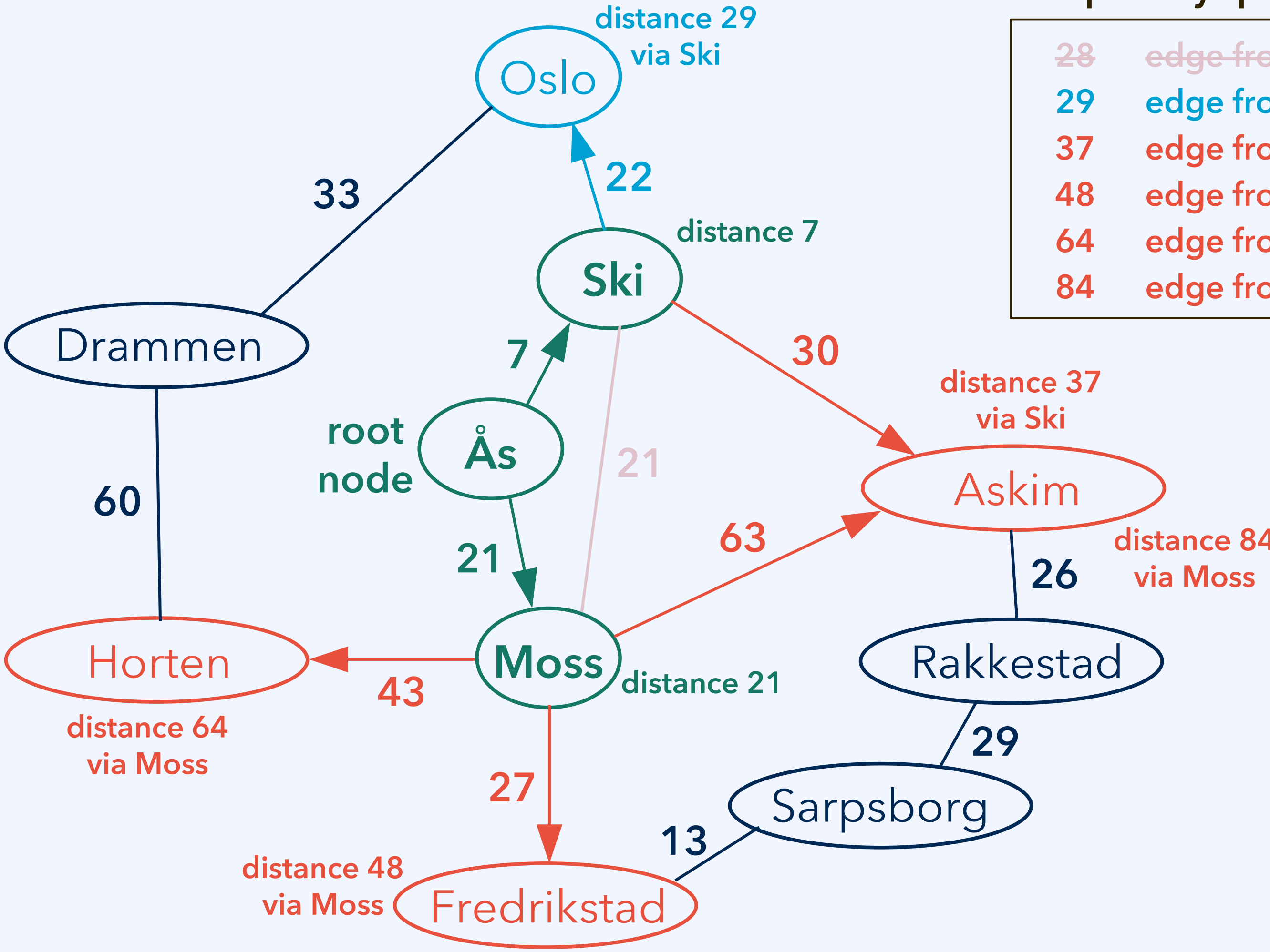
Assumptions:

Undirected graph
All distances are positive

In each iteration, visit the detected node closest to the root.

Process all edges to which that node is incident, detecting any undetected neighbours, and updating tentative distances.

Shortest paths: Dijkstra’s algorithm

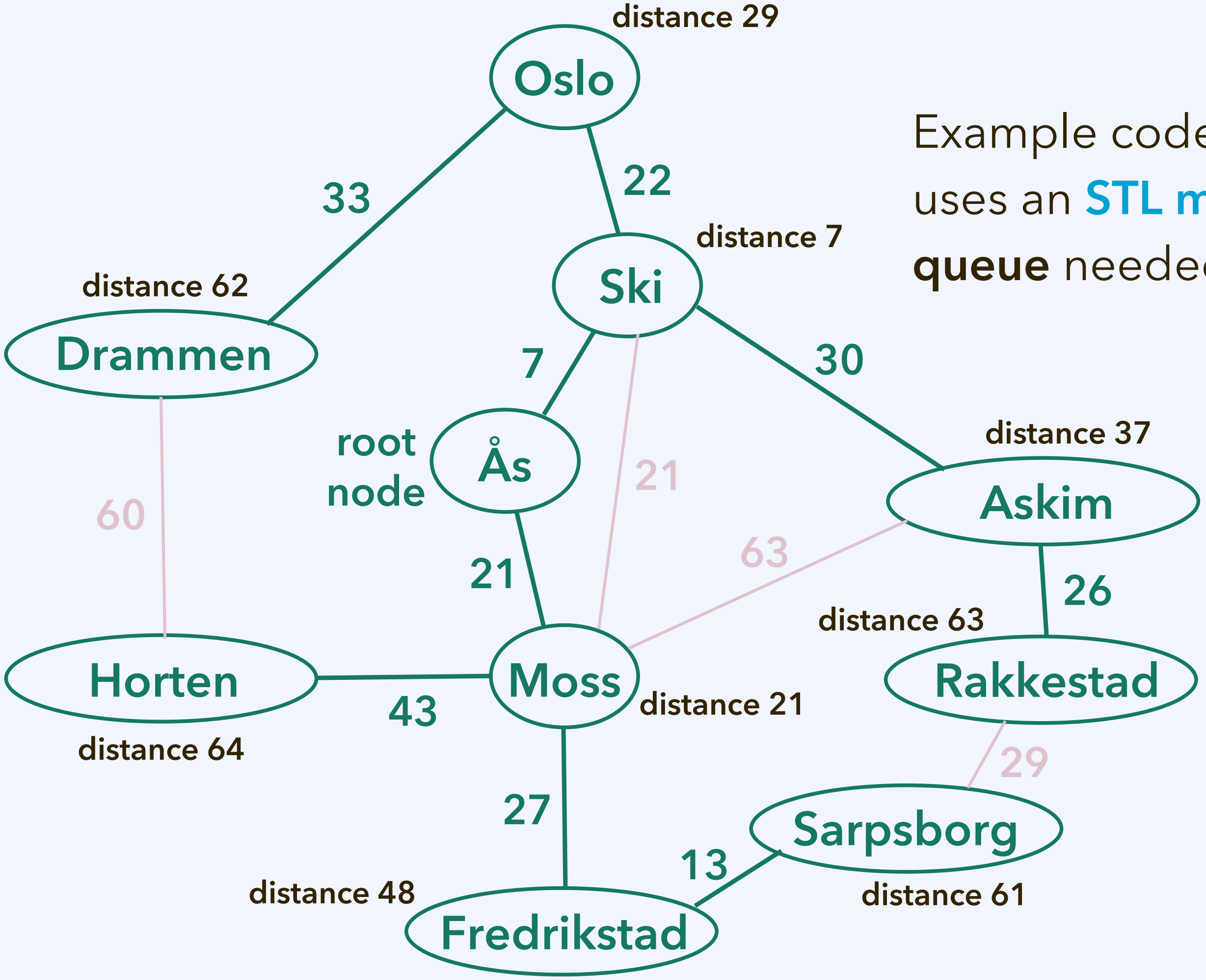


priority queue data structure

~~28~~	~~edge from Ski to Moss~~
29	edge from Ski to Oslo
37	edge from Ski to Askim
48	edge from Moss to Fredrikstad
64	edge from Moss to Horten
84	edge from Moss to Askim

Shortest paths: Dijkstra's algorithm

Norwegian University of Life Sciences



Example code **incidence-list-graph** uses an **STL multimap** as the **priority queue** needed in Dijkstra's algorithm.

A **tailored data structure** is used both for the graph and for the tree that contains all the shortest paths.

Noregs miljø- og biovitskaplege universitet

2 Data structures

2.5 Templates

2.6 Graph data structures

2.7 Tailored containers

Ownership, "rule of five" or "of three"

Container objects take **ownership** (*i.e.*, lifetime and deallocation responsibility) for all or some of the data that they contain. Ownership of data in memory is unique: When the container is deallocated, its owned data are deallocated.

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**.

"**Rule of five:**" Implement **(1)** destructor, **(2)** copy constructor, **(3)** copy assignment operator, **(4)** move constructor, **(5)** move assignment operator.

"**Rule of three:**" **(1)** destructor, **(2)** copy constructor, **(3)** copy assignment operator.

At least implement **(1) the destructor**!
If (2) and (3) are not there, forbid copying.

Examples (for "rule of five"):

- **UndirInclistGraph** in **incidence-list-graph**;
  this is an incidence list based implementation of an undirected graph.
- **DynamicArray** in **sequence-performance**.

Copy constructor and assignment

The **copy constructor** is called if a new object is created (hence, *constructor*) and initialized to have the same content as another object (hence, *copy*).

```
UndirInclistGraph* g = new UndirInclistGraph;
… // read content of g from file
UndirInclistGraph h = *g;
delete g;

… // do something with h
```

This line calls the **default constructor** UndirInclistGraph::UndirInclistGraph()

This line calls the **copy constructor** UndirInclistGraph::UndirInclistGraph(const UndirInclistGraph& original)

This line calls the **destructor** UndirInclistGraph::~UndirInclistGraph()

The implementation must ensure that we can still use h correctly, even though it was copied from *g which is now deallocated

Often the copy constructor needs to create a **deep copy** of the owned data: They are copied in memory, rather than just copying pointers (**shallow copy**).

The **copy assignment** operator (see examples) is called in cases just as above, but when the object to which we copy *already exists* (no need for constructor).

Norwegian University of Life Sciences

Move constructor and assignment

The **move constructor** and, similarly, the **move assignment** operator, are called when content is assigned to a new object just before the old object is deleted.

A typical use case is returning a local object from a function as its return value. We can then often *avoid the expensive deep copying* of the owned content.

```
// move constructor:
// we simply take over the content from the old array

DynamicArray::DynamicArray(DynamicArray&& old)
{
  // take over content from old (no deep copying!)
  this->values = old.values;
  this->capacity = old.capacity;
  this->logical_size = old.logical_size;

  // remove from old so that it does not get deleted
  old.values = nullptr;
  old.capacity = 0;
  old.logical_size = 0;
}
```

```
// copy constructor:
// afterward, the content must exist in memory twice

DynamicArray::DynamicArray(const DynamicArray& original)
{
  this->values = new int[original.capacity]();
  this->capacity = original.capacity;

  // deep-copy the content of original
  if(original.values != nullptr)
    std::copy(original.values,
              original.values + original.logical_size, this->values);

  this->logical_size = original.logical_size;
}
```

Noregs miljø- og biovitskaplege universitet

Conclusion

Project group formation

Edit

Project group formation: Reminder

10 Oct at 8:41

Martin Thomas Horsch

All sections

Dear students,

reminder: The deadline for self-organizing into programming project groups is this Wednesday (12th October) lecture time. At that point, all who are not in a group with either two or three members will be arranged into groups. But there will still be the possibility for all to rearrange their groups until end of the month.

The suggested problems, from which you can choose one for your group work, will be released by 26th October lecture time. The group work will then run from tutorial time in calendar week 43 (27th October) for five weeks, until calendar week 48. Group work presentations will be scheduled in the time window between week 48 and week 50.

If you have your own idea on what you would like to work on, that is possible and encouraged as long as it is aligned with the INF205 learning outcomes: If it demonstrates that you reach the learning outcomes, it is good. But we will in this case need a meeting some time between now and the end of the month to approve it - better don't wait too long and let as discuss it ASAP.

Best wishes,

Martin

Search entries or author

Unread

Write a reply...

Norges miljø- og biovitenskapelige universitet

INF205
Resource-efficient programming

2 Data structures

2.5 Templates
2.6 Graph data structures
2.7 Tailored containers