Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205

Resource-efficient programming

3 Data structures

3.1 Object orientation
3.2 Inheritance
3.3 Linked data structures

3.4 Containers
3.5 Graph data structures
3.6 Streams and file I/O

Weekly glossary concepts

Norwegian University of Life Sciences

What are essential concepts from the previous lecture?

Let us include them in the **INF205 glossary**.[1]

???

container

queue

???

stack (data structure)

???

[1]https://home.bawue.de/~horsch/teaching/inf205/glossary-en.html

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

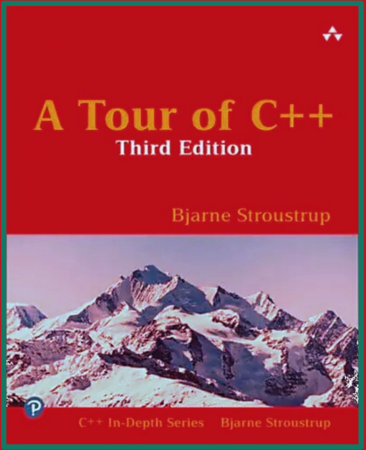
3 Data structures

3.1 Object orientation

3.2 Inheritance

3.3 Linked data structures

3.4 Containers



Sections 6.1 – 7.2

Core Guidelines:

C.31 – C.33, T.1, T.2, T.62, T.83

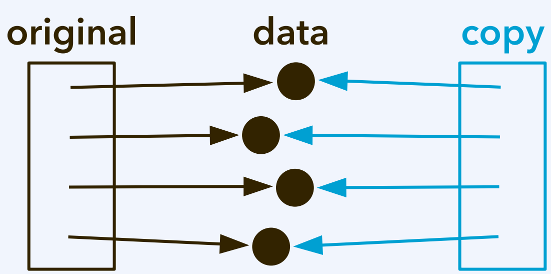
(+ more "C" & "T")

NMBU Norwegian University of Life Sciences

Rule of three: Shallow vs. deep copy

Shallow copy:

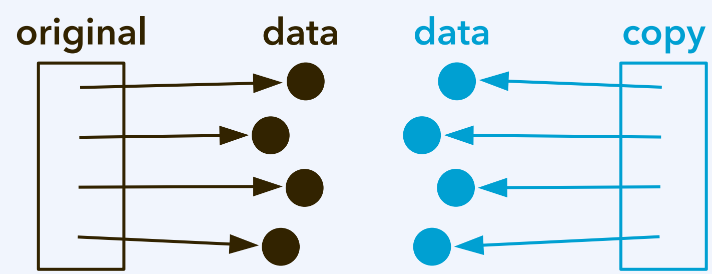
Standard copying, such as if there is no handwritten copy constructor or copy assignment operator, will simply **copy the value of pointers**, *not the content* to which they point.



After shallow copying, the content will exist **once in memory**. This can be appropriate when the content is **not owned** but just pointed at.

Deep copy:

Standard copying, such as if there is no handwritten copy constructor or copy assignment operator, will simply **copy the value of pointers**, *not the content* to which they point.



After deep copying, content exists **twice in memory**. Design following the concept of a "container" that **uniquely "owns"** its content requires deep copying.

Rule of three: (2) Copy constructor

Norwegian University of Life Sciences

The **copy constructor T::T(const T& orig)** is called when the following two are done at the same time: **(1) allocation** of an object, so that a constructor needs to be called, and its **(2) initialization** to the value of a pre-existing object that continues to exist.

Examples for when the **copy constructor** is called:

```
// default constructor
T tfirst;
…
// copy constructor
T tsecond = tfirst;
```

```
void func(T param) { … }

int main() {
   T tobject;
   …
   // copy constructor
   func(tobject);
}
```

after running the copy constructor, the same content must exist in memory twice!

```
class T
{
public:
   T() { this->p = new S[1000](); }
   T(const T& original) {
      this->p = new S[1000]();
      std::copy(
         original.p, original.p+1000,
         this->p
      );
   }
   …
}
```

std::copy can be used for data that are contiguous in memory

1. Create space for the duplicate.
2. Now write the duplicate into it.

Rule of three: (3) Copy assignment operator

The **copy assignment operator** technically is an overloaded "=" operator:

```
T& T::operator=(const T& rhs) { … }
```

Difference from the copy constructor:

- Object already exists, hence *no initial allocation* of memory for content.
- But *deallocate pre-existing content if necessary*.

```
// default constructor
T tfirst, tsecond;
…
// copy assignment
tsecond = tfirst;
```

A **copy assignment** is done whenever we copy the value of one variable to another, **both existed** before, and **both continue to exist**.

```
class T
{
public:
  T() { this->p = new S[1000](); }
  T& operator=(const T& rhs) {
    if(&rhs == this) return *this;

    std::copy(
      rhs.p, rhs.p+1000, this->p
    );
    return *this;
  }
  …
}
```

after running the copy assignment, the same content must exist in memory twice!

Rule of three and rule of five

Container objects take **ownership**, *i.e.*, lifetime and deallocation responsibility. 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**.

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.

Most often you will then also need to implement **(0)** a constructor.

"**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**.

Rule of five: (4) Move constructor

The **move constructor** is called when the content of an *old object* can be shifted to a *new object* that is *allocated and initialized* (*e.g.*, *before we deallocate the old object*).

```
T::T(T&& old) { … }
```

```
T func(…) {
  T tfirst;
  …
  return tfirst;
  // the destructor will be called
}

int main() {
  // but before, call the move constructor
  T tsecond = std::move( func(…) );
}
```

Typical use case: Efficient **handover of content** returned by a function.

```
class T
{
public:
  T() { this->p = new S[1000](); }

  T(T&& old) {
    this->p = old.p;
    old.p = nullptr;
  }
  …

private:
  S* p …
}
```

A **shallow copy** of the pointer to the content is good enough; after the action, the content *exists in memory only once*!

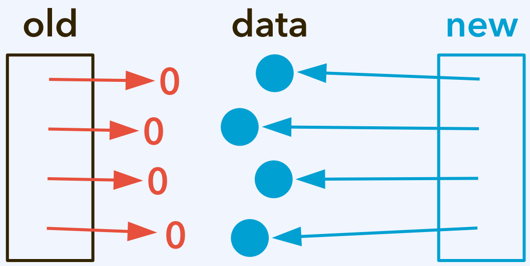
Attention: Right after the move constructor for "this", the destructor of "old" might be called.

Remove all pointers to the content from old, so that it does not get deallocated!

Move constructor: Why can it be advantageous?

Move constructor + destructor:

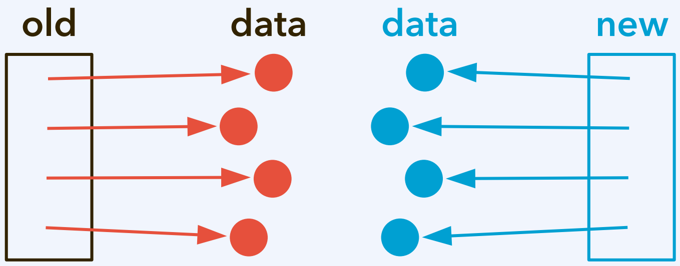
The move constructor is used to *make a new container own the data* without copying the data. A **shallow copy** is made, and the *data are detached* from the old container.



The shallow copy is an inexpensive operation. If the data exist **once in memory** both before the operation and after, *why copy them* from one place to another?

Copy constructor + destructor:

If there is *no move* constructor, or the compiler does not enforce a move, first all the content is copied (**deep copy**); the old container is probably *deallocated right after*.



This is an expensive operation whenever there is a substantial amount of data. All data are *copied*, *unnecessarily*, since at the end they still exist only **once in memory**.

Example file: copying-and-moving.zip

Rule of five: (5) Move assignment operator

The move assignment operator relates to the move constructor the same way as the copy assignment operator relates to the copy constructor.

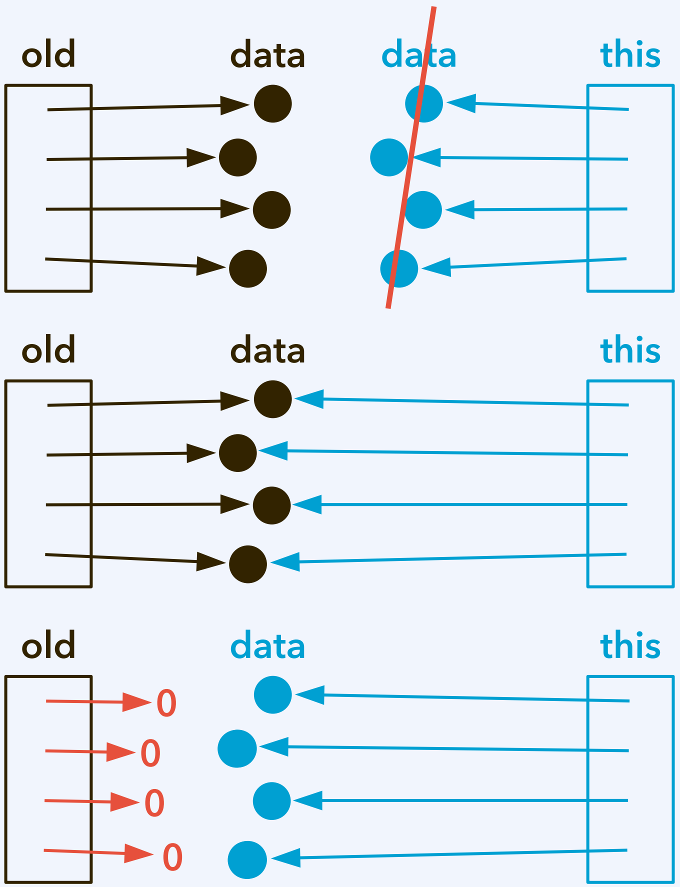
```
T& T::operator=(T&& old) { … }
```

```
T func(…) {
  T tfirst;
  …
  return tfirst;
  // the destructor will be called
}

int main() {
  T tsecond;
  …
  // but before, call the move assignment operator
  tsecond = std::move( func(…) );
}
```

constructor called

tsecond exists already



Example file: copying-and-moving.zip

Templates: 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

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!)

Example file: list-template.zip

Templates for functions and methods

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;
}
```

```
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);
}
```

Example file: list-template.zip

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

Sign-up for the third worksheet

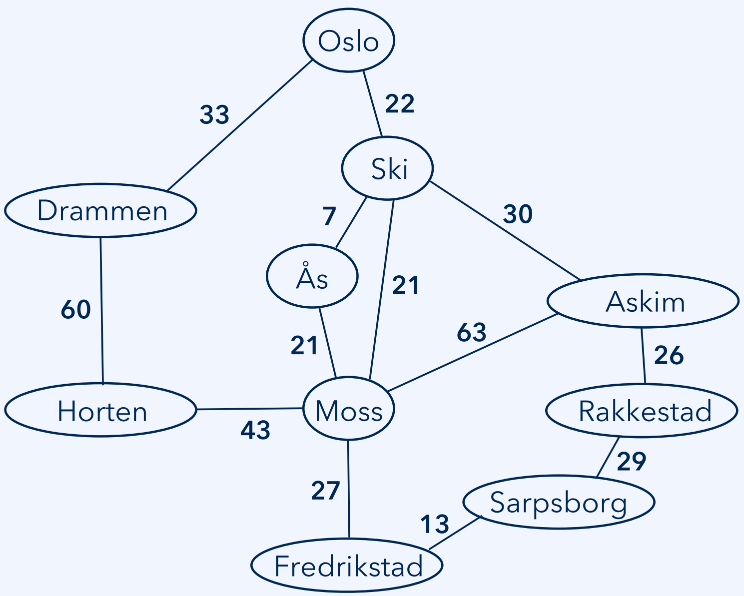
Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

3 Data structures

Graphs as non-sequential linked data structures



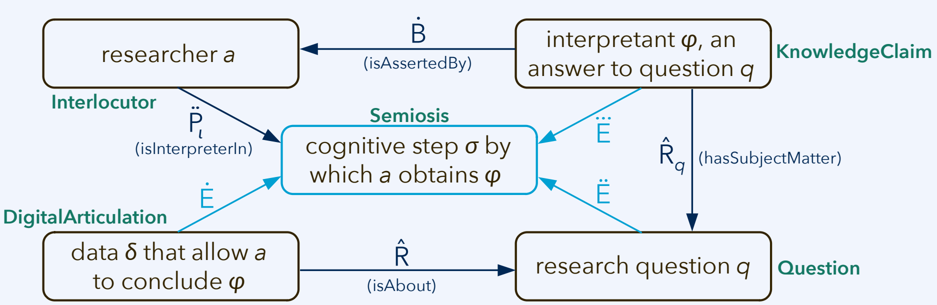
Sequential data structures arrange their items in a linear shape. Sometimes that is not the best solution, or it is not appropriate at all.

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.

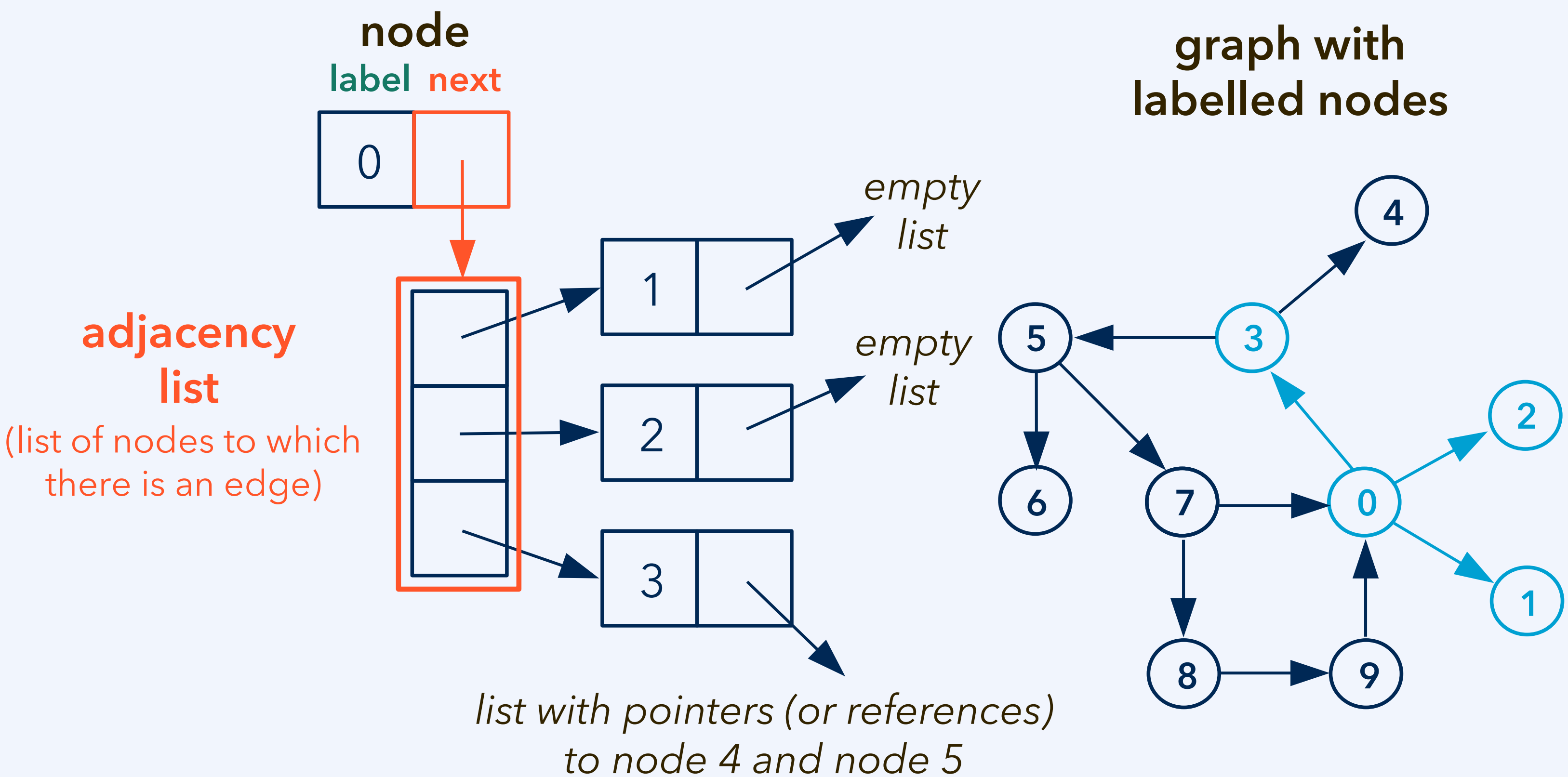
Graphs can be used to represent relations between objects, such as distances on a map, or as a **knowledge graph**.

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



Data structure implementation: Adjacency lists

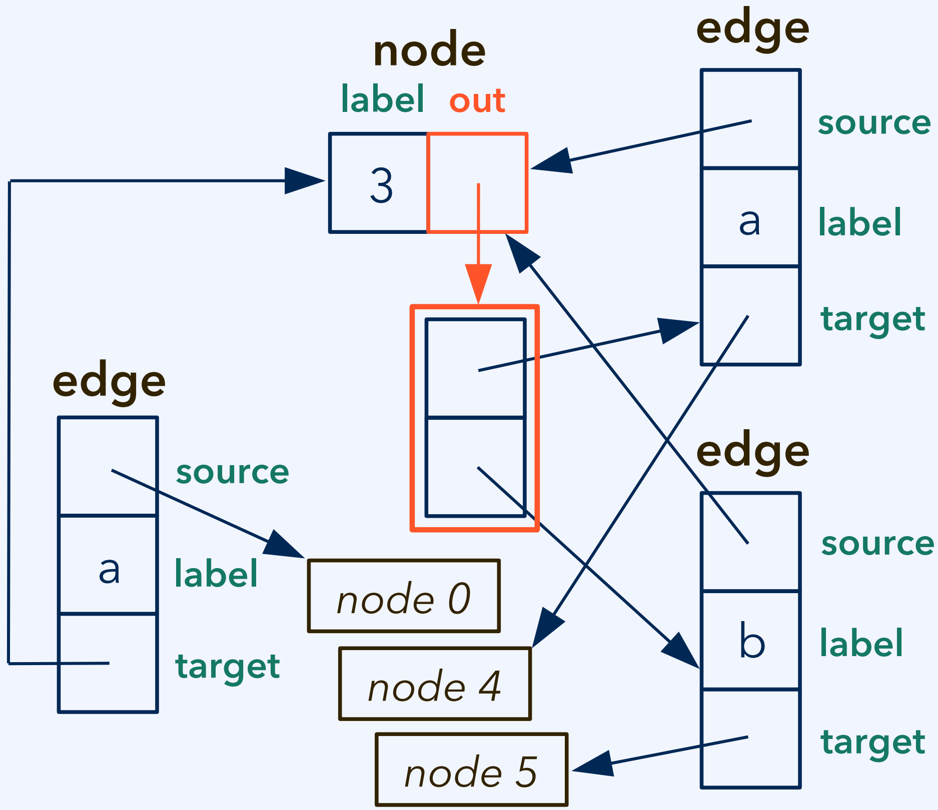
In a graph, one node can be connected to multiple other nodes. An **adjacency list** (with various possible implementations) can be used to manage these links.

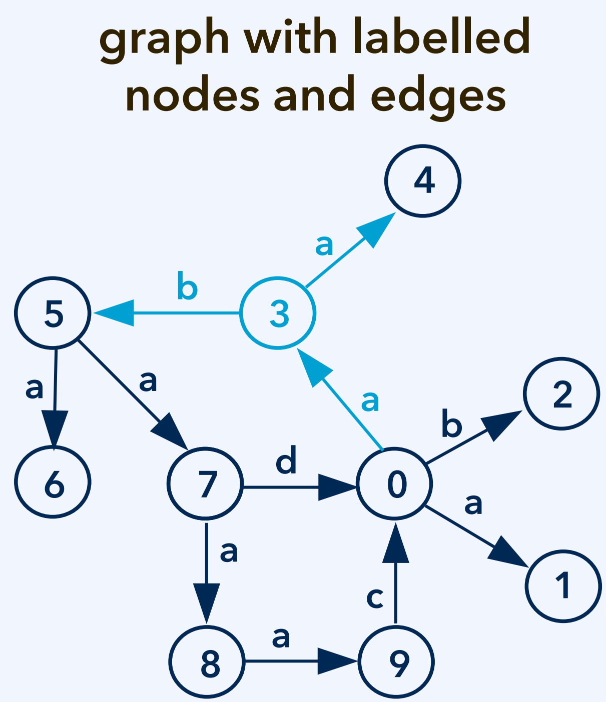


Doubly-linked version of this: Two lists, for incoming and for outgoing edges.

Data structure implementation: <u>Incidence lists</u>

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.

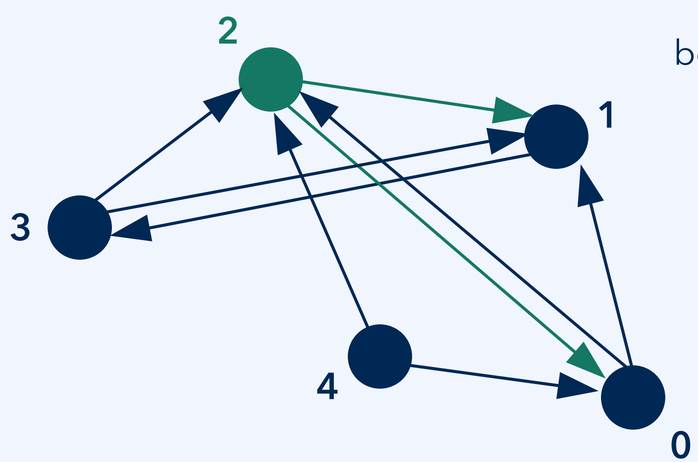




Doubly-linked version of this: Two lists, for incoming and for outgoing edges.

Data structure 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},`	**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**

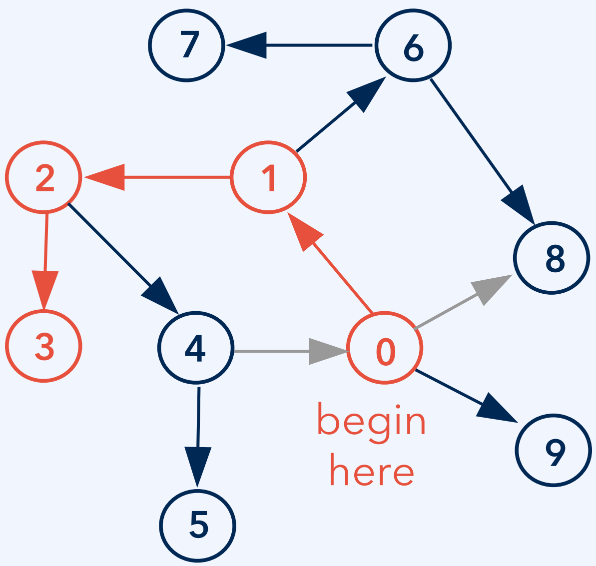
For a sparse graph, the vast majority of entries in the 2D array/matrix is “false”. Adjacency matrices are commonly only used when expecting a **dense graph**.

Graph traversal

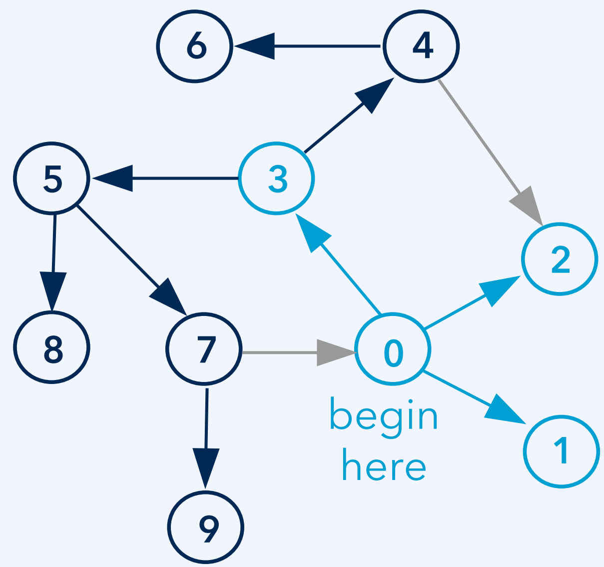
Traversal of graphs: Depth-first search and breadth-first search

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)

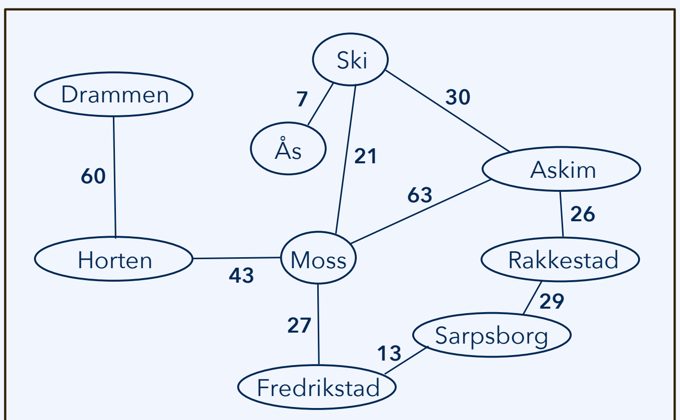


Note: Only elements *to which there is a path from the initial node* can be found.

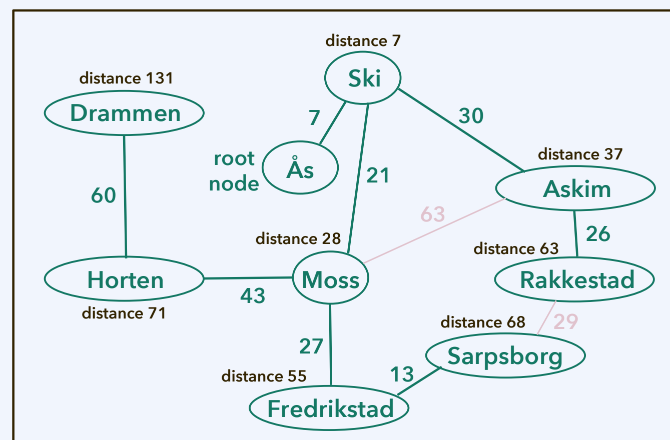
Graph traversal: Shortest paths (see also INF221)

For an example showing travel minutes between locations according to ruter, see the example code **incidence-list-graph.zip**.

The data structure employed in the code are **incidence lists** (for undirected graphs). Dijkstra's algorithm is implemented.

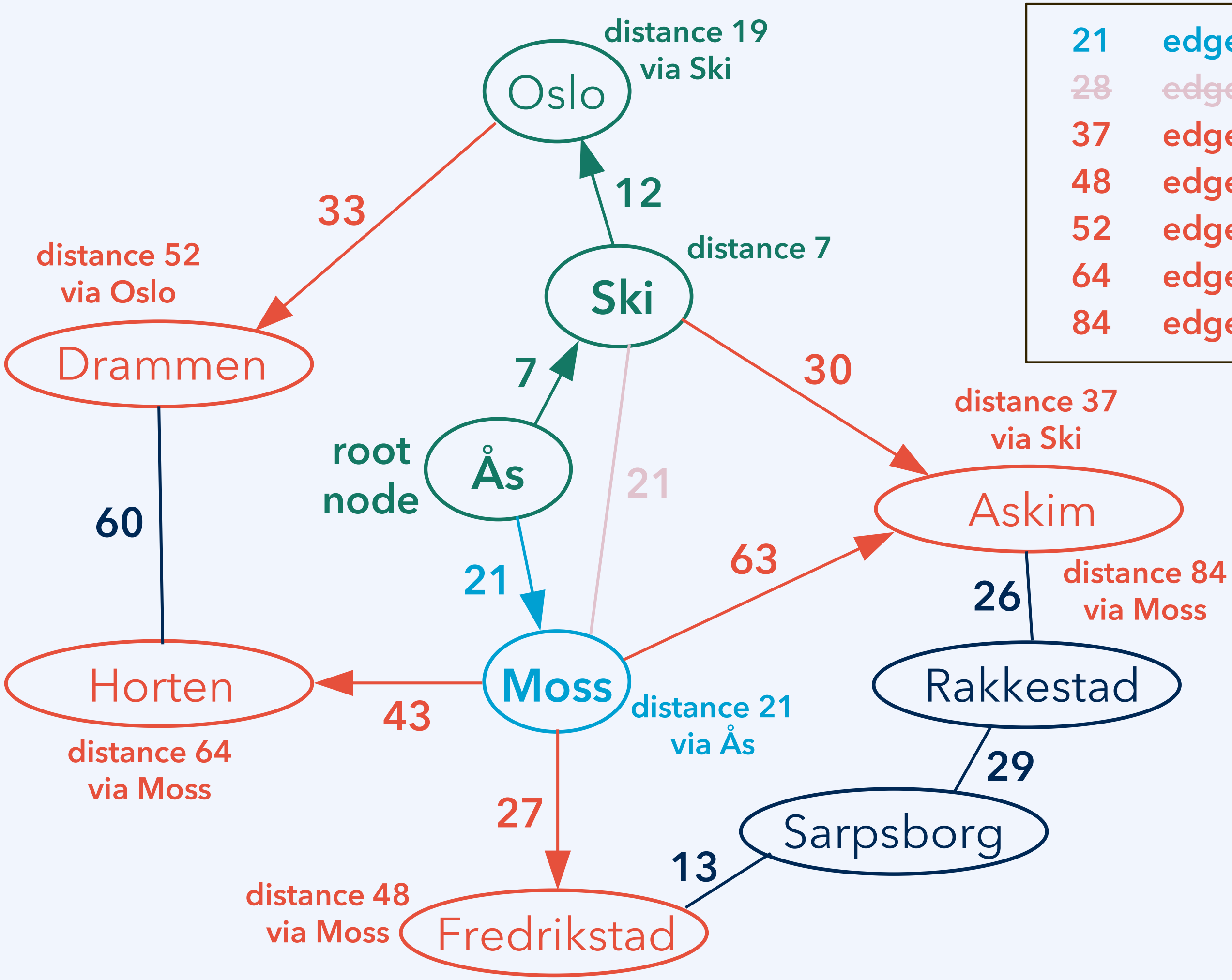


Dijkstra's algorithm



Example file: incidence-list-graph.zip

Graph traversal: Shortest paths (see also INF221)



priority queue data structure

21	edge from Ås to Moss
~~28~~	~~edge from Ski to Moss~~
37	edge from Ski to Askim
48	edge from Moss to Fredrikstad
52	edge from Oslo to Drammen
64	edge from Moss to Horten
84	edge from Moss to Askim

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

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

Example file: incidence-list-graph.zip

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

3 Data structures

3.1 Object orientation

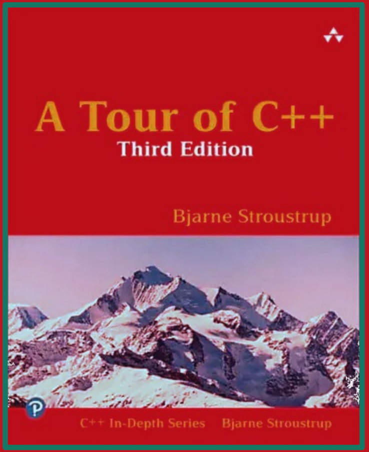
3.2 Inheritance

3.3 Linked data structures

3.4 Containers

3.5 Graph data structures

3.6 Streams and file I/O



6.4, 11.7, 11.9

Core Guidelines:

C.168

I/O operator overloading

See example code **io-operator-overloading.zip** for the following.

Assume that for some **class C**, we have defined methods that write content to a stream, or that analogously read from a stream.

```
void C::out(ostream* target) const {
  *target << … ;
}
```

```
void C::in(istream* source) {
  *source >> … ;
}
```

You can convert this to overloaded I/O operator definitions:

```
ostream& operator<<(
  ostream& str, const C& x
) const {
  x.out(&str);
  return str;
}
```

```
istream& operator>>(istream& str, C& x)
{
  x.in(&str);
  return str;
}
```

Now you can use the operator **<<** and the operator **>>** on objects of type C just like for numbers, *etc.*

Advice: Input & output methods/operators should use the same serialization.

Example file: io-operator-overloading.zip

File input/output

Norwegian University of Life Sciences

We **must serialize the data** in order to store them in a file!

To transfer data through a communication channel as a message, the data items and their parts need to be serialized (ordered) in a well-defined way that is understood both by the sender and the receiver.

- As a contiguous chunk of memory, *if the exchange is memory-based*.
- As a file, *if file I/O is the mechanism* by which data are exchanged.

File stream objects can be used in order to read or write a file.

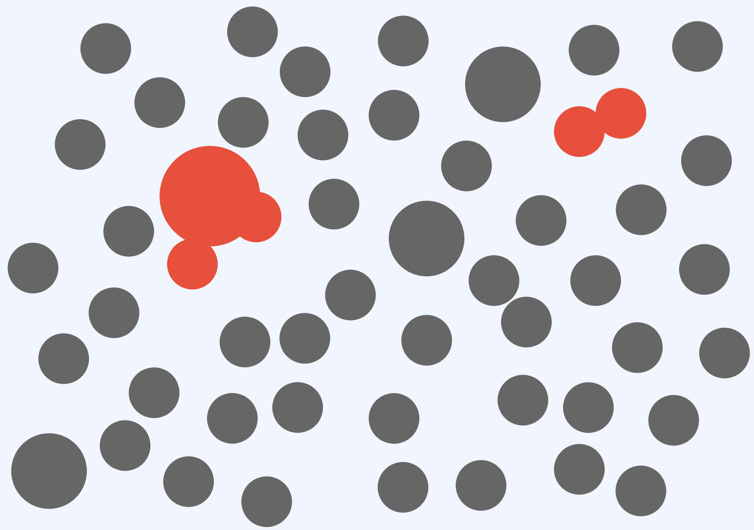
```
// open in-filestream
std::ifstream infile(argv[1]);      // file name given as command-line argument argv[1]

// read graph object from file
graph::UndirInclistGraph g;
infile >> g;
infile.close();
```

Example file: run-graph-example.cpp
(in **incidence-list-graph.zip** archive)

Example code: Spheres in a box

N spheres of different types (with different radii) are positioned in a 3D system.



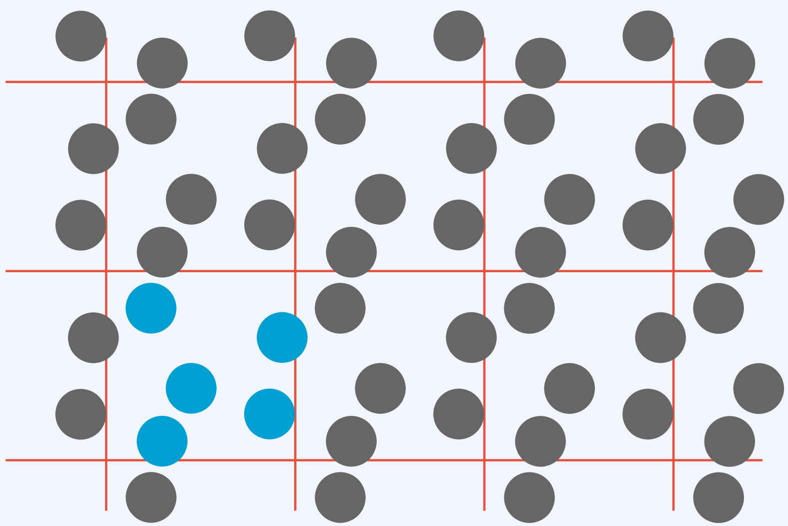
Evaluate the number of overlaps.

This can be a (very) simple model of a liquid or solid. It is much easier to implement, and faster to compute, than more realistic molecular models.

Example file: repulsive-spheres.zip

Conventions for the box in molecular simulation

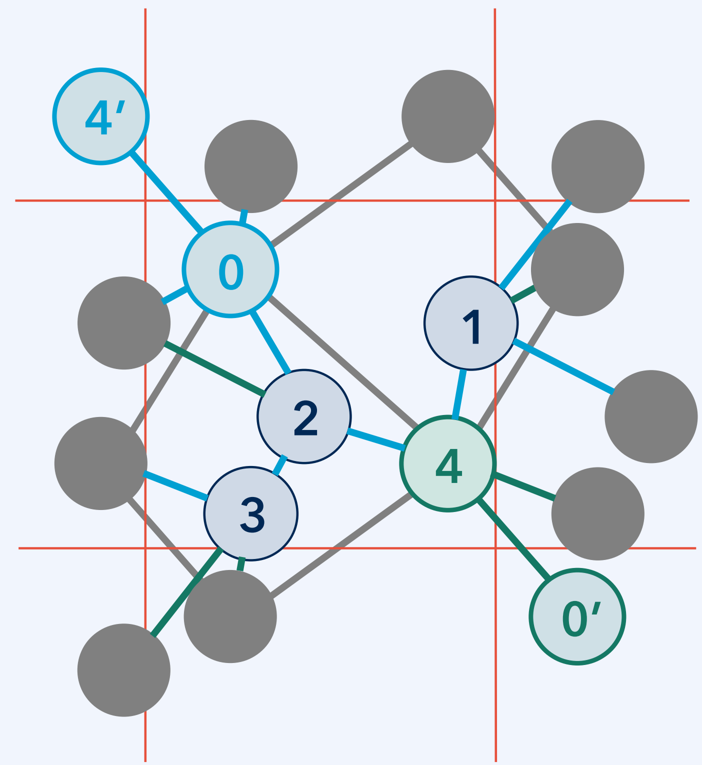
Periodic boundary condition (PBC)



PBC: Assume that the simulation box repeats periodically in all directions.

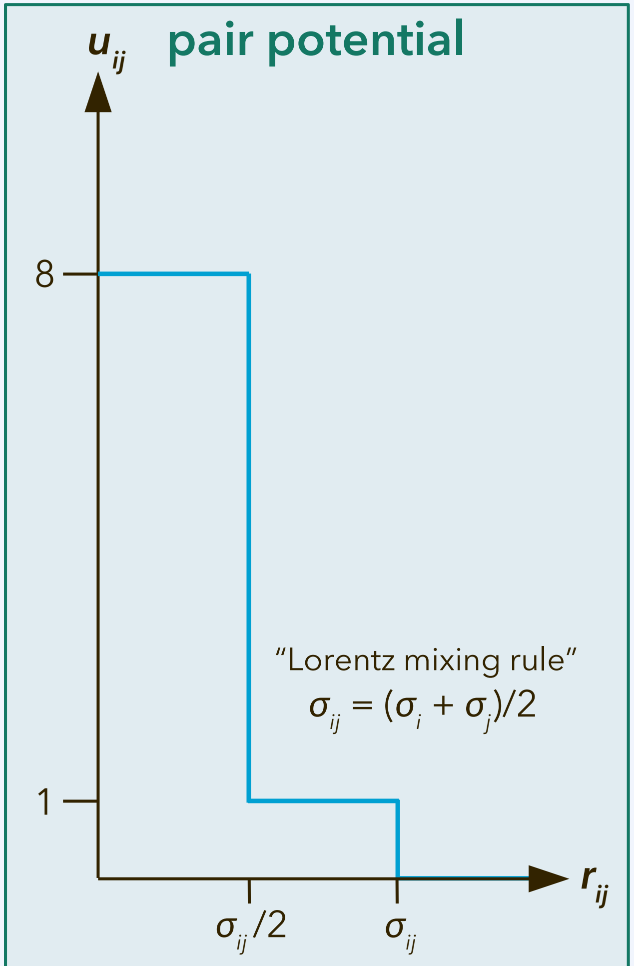
MIC: Each particle interacts only with closest replica of each other particle.

Minimum image convention (MIC)



interact, count for potential (*e.g.*, overlaps)
interact, don't count for potential
don't interact

Example code: Repulsive spheres with soft shielding



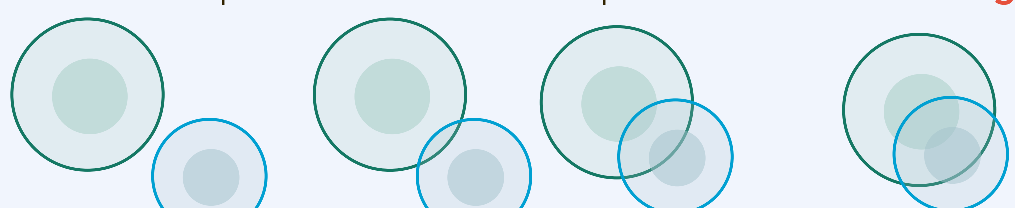

```
int Sphere::check_overlap(const Sphere* other, const double box_size[3]) const
{
  // square distance between the centre of i and the centre of j
  double square_distance = 0.0;
  for(int d = 0; d < 3; d++) {
    double dist_d = other->coords[d] - this->coords[d];

    // apply minimum image convention
    if(dist_d > 0.5*box_size[d]) dist_d -= box_size[d];
    else if(dist_d < -0.5*box_size[d]) dist_d += box_size[d];

    square_distance += dist_d*dist_d;
  }

  // is the square distance smaller than the square of the sum of radii?
  double sum_of_radii = 0.5 * (this->size + other->size);
  int overlap = 0;
  if(square_distance < 0.25*sum_of_radii*sum_of_radii) overlap = 8;  // soft shielding
  else if(square_distance < sum_of_radii*sum_of_radii) overlap = 1;  // normal overlap
  return overlap;
}
```

no overlap | **one** overlap | counts as **eight**



$\mathrm{dist}(i, j) \geq (\sigma_i + \sigma_j)/2$ | anything in between | $\mathbf{dist}(i, j) < (\sigma_i + \sigma_j)/4$

See implementation in **repulsive-spheres.zip**, sphere.cpp, line 51.

Example code: Generating a configuration

See generator code. Main scenario parameter:

- N, the number of spherical particles in the system; default: $N = 4096$

Additional benchmark scenario parameters (change only if you have a reason):

- packing fraction ξ, *i.e.*, total sphere volume / box volume; default: $\xi = 0.5$
- ratio ζ_{max} between largest and smallest sphere diameter; default: $\zeta = 3$

Remark: If the spheres are all of the same size, the densest packing (without any overlaps) has the packing fraction 0.7405.

This had been known as one of the "Hilbert problems."

THEOREM 1.1 (The Kepler conjecture). *No packing of congruent balls in Euclidean three space has density greater than that of the face-centered cubic packing.*

This density is $\pi/\sqrt{18} \approx 0.74$.

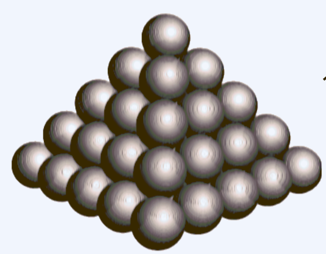


Figure 1.1: The face-centered cubic packing

The proof of this result is presented in this paper. Here, we describe the

T. C. Hales, "A proof of the Kepler conjecture," *Ann. Math.* **162**(3): 1065–1185, doi:10.4007/annals.2005.162.1065, **2005**.

Example file: generator.cpp (in **repulsive-spheres.zip** archive)

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

Conclusion

Weekly glossary concepts

Norwegian University of Life Sciences

What are essential concepts from this lecture?

Let us include them in the **INF205 glossary**.[1]

operator overloading

???

???

graph

rule of five

???

[1]https://home.bawue.de/~horsch/teaching/inf205/glossary-en.html

Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205

Resource-efficient programming

3 Data structures

3.1 Object orientation
3.2 Inheritance
3.3 Linked data structures

3.4 Containers
3.5 Graph data structures
3.6 Streams and file I/O