

# DATA STRUCTURES & ALGORITHMS

# MOTIVATION

---

- Its really about **how to solve problems** more than how to program.  
Fundament of software engineering.
- An **efficient** algorithm may be the difference between solving in seconds vs. years, or being able to run within memory limits or not.
- You rarely create new algorithms or data structures. **Likely be building upon fundamentals like arrays, lists, trees, hash tables and searching, sorting, traversing, path finding.**
- Your task is to know available algorithms and data structures and **understand how to choose.**

# LEARNING OBJECTIVES

---

At the end of the course, the participants will be able to:

- **Describe** and implement common data structures and associated algorithms
- **Explain** the underlying mathematical models of data structures and algorithms
- **Design** data structures and algorithms, and judge their performance and suitability for specific situations.

# WHO ARE WE?

---

**Assistant Professor** Cláudio Ângelo Gonçalves Gomes

**Background** See <https://clagms.github.io/> for details.

- Research into optimization of co-simulation algorithms, reachability analysis, etc...



**Teaching Associate Professor** Peter Høgh Mikkelsen

**Background** 10+ years professional experience in embedded hardware and software development and 10 years for experience teaching various HW / SW topics.



# CONTACT DETAILS

---

- The most convenient way – email:
  - Peter: [phm@ece.au.dk](mailto:phm@ece.au.dk)
  - Claudio: [claudio.gomes@ece.au.dk](mailto:claudio.gomes@ece.au.dk)
- Teaching Assistants' emails:
  - Carlos Ignacio Isasa Martin: [cisasa@ece.au.dk](mailto:cisasa@ece.au.dk)
- Or see us in our offices:
  - Peter: 5123 - 424
  - Claudio: 5123 - 425



# HOW TO GET A QUICK EMAIL REPLY

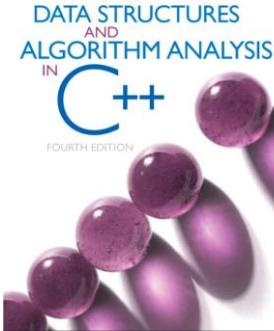
---

We really appreciate your questions! Follow the following guidelines:

- Include both teachers emails as well as the teaching assistant's emails (just copy the contacts below and paste them on your emails).
  - Rationale: The first person to reply to your email will include all the other staff in cc, thus allowing us to know that you've been helped.
- Do not send duplicate emails to different staff individually please (we end up doing duplicate work)
- Subject should be informative. Example: "Data structures and algorithms: Problem in Question 1a".
  - Rationale: we teach different courses and it's sometimes hard to figure out which course you are talking about in the email content.

Emails for copy paste:

"Peter Høgh Mikkelsen" <phm@ece.au.dk>; "Cláudio Ângelo Gonçalves Gomes" <claudio.gomes@ece.au.dk>;  
"Carlos Ignacio Isasa Martin" <cisasa@ece.au.dk>



# TEACHING MATERIAL

---

- **Book:** Data Structures & Algorithm Analysis in C++ by Mark A. Weiss, Pearson, 4th Edition, 2013. Errata: [https://users.cs.fiu.edu/~weiss/dsaa\\_c++4/errata.html](https://users.cs.fiu.edu/~weiss/dsaa_c++4/errata.html)
- **Bi-weekly** exercises sheets (6 total)
- Slides ← Very descriptive, start here when prepping lecture.
- Some articles
- Some additional videos (talks about the topics in a different way – additional)
- **Programming language:** C++
- **Programming environment:** what you feel comfortable with! (Visual Studio, VS Code, Vim/Shell, ...)

# ONLINE REFERENCES

---

- No need to memorize standard libraries.
- Internet is rich on material about course content
- C/C++:
  - <http://www.cplusplus.com/>
  - <https://en.cppreference.com>
- Author's website for code:
  - [https://users.cs.fiu.edu/~weiss/dsaa\\_c++4/code/](https://users.cs.fiu.edu/~weiss/dsaa_c++4/code/)
- General:
  - <https://stackoverflow.com/> (don't mindlessly copy paste!)

# COURSE WEB PAGES

---

- All information concerning this course including lecture notes, assignments announcements, etc. can be found on the course web pages at Brightspace ([www.brightspace.au.dk](http://www.brightspace.au.dk))
- You should check this site frequently for new information and changes. It will be your main source of information for this unit. The layout of the web pages should be fairly self explanatory
- **Please let us know if you don't have access**

# EDUCATIONAL FORM

---

- Interaction with the teachers: Tuesdays 08:15 – 11:50 in Shannon 009/013
- Form
  - Combination of overview lecture, student reflection exercise. **You need to prepare!**
  - Work with programming exercises
- Exam form
  - Written, “open book” three hours

# THE PERFECT STUDENT OF DST+ALG

---

Curiosity

Motivation

Responsibility

Critical Thinking

Communication

Adaptability

Collaboration

Resilience



# EXERCISES

---

- In addition to smaller reflection exercises, you will also work with exercises in the exercise part of lectures, **and on your own time if needed.**
- FOUR exercises must be handed in and approved through a peer reviewing process
- It is **VERY STRONGLY RECOMMENDED** to work with and complete all exercises.
  - Not just understanding a solution. You must be able to come up with it, and with alternative solutions if applicable.

# POINTERS, ITERATORS AND SEARCHING

# VARIABLES IN C++ MEMORY SYSTEM

---

There are two "kinds" of memory available to a process:

## **Stack:**

- Stores local variables of the function.
- Removed once the function ends!

## **Heap:**

- Contains dynamically allocated variables.
- Stays once the function ends, unless cleared before!

Read more: <http://www.learnCPP.com/cpp-tutorial/79-the-stack-and-the-heap/>

# POINTER

---

A pointer is a variable that holds the memory address of another variable of the same type

It supports dynamic memory allocation

**int x;** A variable storing an integer

**int \*foo;** A variable storing a pointer to an integer

**x= 123;**

**foo = &x;**  $\&x$  is the **address** of the variable  $x$

**if \*foo==123 ...**  $*foo$  is the **value** that  $foo$  points to (here the value of  $x$ )

# ASSIGNING A VALUE TO A *DEREFERENCED* POINTER

---

A pointer must have a value before you can *dereference* it (follow the pointer).

```
int *x;
```

```
*x=3;
```

ERROR!  
Overrode the value located  
in some address!

```
int foo;
```

```
int *x;
```

```
x = &foo;
```

```
*x=3;
```

this is fine  
x points to foo



# REFLECTION: COMPARING POINTERS

---

```
int j = 5;  
int i = 5;  
int *ptrj1 = &j;  
int *ptrj2 = &j;  
int *ptri = &i;
```

True/False:

```
if (ptrj1 == ptrj2) ?  
if (ptrj1 == ptri) ?  
if (&ptrj1 == &ptrj2) ?  
if (*ptrj1 == *ptri) ?
```

# SOLUTION: COMPARING POINTERS

```
—  
int j = 5;  
int i = 5;  
int *ptrj1 = &j;  
int *ptrj2 = &j;  
int *ptri = &i;
```

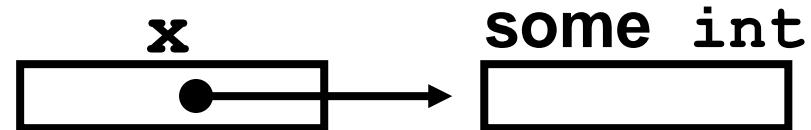
True/False:

if (ptrj1 == ptrj2) ?	True
if (ptrj1 == ptri) ?	False
if (&ptrj1 == &ptrj2) ?	False
if (*ptrj1 == *ptri) ?	True

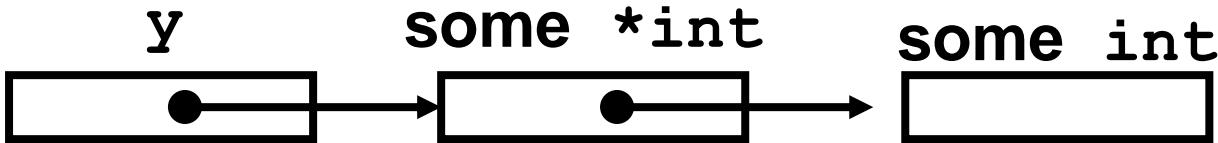
# POINTERS TO POINTERS

---

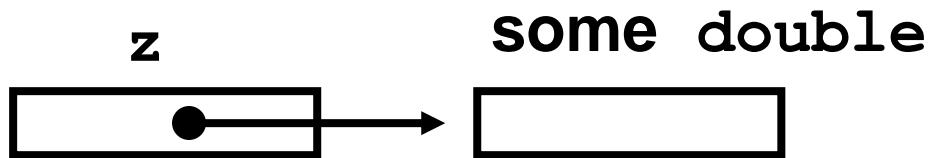
```
int *x;
```



```
int **y;
```



```
double *z;
```



# POINTERS AND ARRAYS

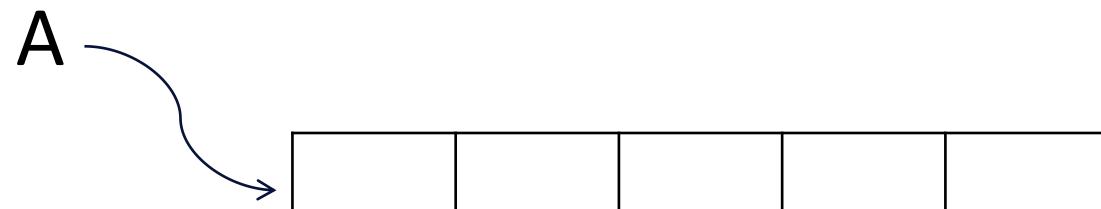
---

Array name is basically a *const pointer* pointing at the beginning of the array.

You can use the [] operator with pointers!

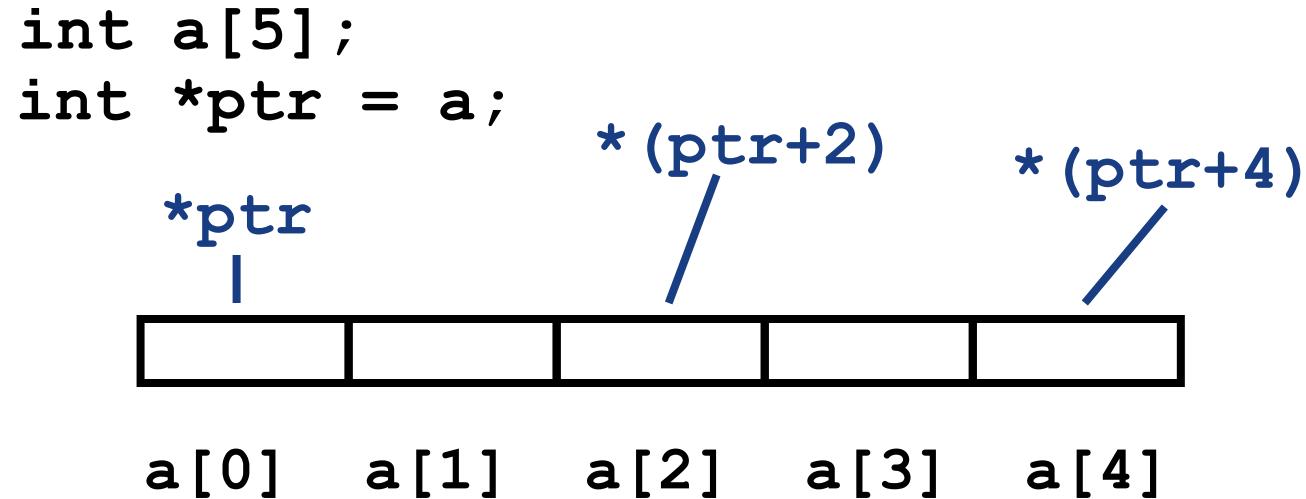
Example:

- `int A[5];`
- Creates a memory block of 5 integers on the stack (5x4bytes) where A (the pointer) points at the beginning of the array (i.e. A[0]).



# POINTER ARITHMETIC

- Integer math operations can be used with pointers: +, -, ++, --, +=, -=
- If you **increment** a pointer, it will be increased by the size of whatever it points to (the size of the type).
- Incrementing pointers will basically make it point to the “next” element in memory.



# REFLECTION

---

```
int a[5] = {0, 1, 2, 3, 4};
```

```
int * p = a;
```

The expressions below are true or false?

$*p == a[0]$

$*p+1 == a[1]$

$*(p+3) == a[3]$

# REFLECTION

---

```
int a[5] = {0, 1, 2, 3, 4};  
int * p = a;  
p++;
```

The expressions below are true or false?

`*p == a[0]`

`*p == a[1]`

# USING THE HEAP

---

All primitives are stored in the stack,  
All that is made without new command  
is stored in the stack. (except global variables and initialized char\* -splab!)  
Using the **new** keyword, we can now  
allocate memory on the **heap**.

```
int *i = new int;  
string *str = new string("hi there, heap is cozy!");  
int *arr = new int[5];
```

Deleting these objects can be done  
by using the **delete** keyword.

- delete i;
- delete str;
- delete [] arr;

Safe Delete:

```
if (0 != p) {  
    delete p;  
    p=0;  
}
```

DO NOT DELETE A  
PONTER NOT ALLOCATED  
BY new



# POINTER PITFALLS

---

Assigning values to uninitialized, null, or deleted pointers:

```
int *p;           int *p = NULL;           int *p = new int;  
*p = 3;          *p = 4;                 delete p;  
                  *p = 5;
```

All of these cases end up with **segmentation fault!**

# DANGLING POINTER

---

A pointer that points at nothing:

Example:

```
int *p, *q;  
p = new int;  
q = p;  
delete q;  
*p = 3; //illegal assignment!
```

- p and q point to the same location, q is deleted, results with p becoming a dangling pointer!

# MEMORY LEAKS

---

Memory leak is when you remove the reference to the memory block, before deleting the block itself. Example:

```
int *p = new int;  
p = NULL; //or a new other value
```

p points at memory location of type int.

p changed to point at null.

Result? **Memory leak!**

There are no pointers to allocated block, so no way to clear it.  
Must free memory block before changing reference.

A memory leak can diminish the performance of the computer by reducing the amount of available memory.

Eventually, in the worst case, too much of the available memory may become allocated

and all or part of the system or devices stops working correctly, the application fails, or the system slows down.

Use [valgrind](#) to test if your implementation has memory leaks.

[Valgrind User Manual](#), [The Valgrind Quick Start Guide](#), [Graphical User Interfaces](#)

# TEMPLATE CLASSES

---

A **C++ template** is literally a template or blueprint for creating a generic class or function of a specified type.

- A template is not a class.
- Using C++ templates avoids having to write nearly identical classes (int, float, etc.), but results in compiled code that is mostly as if we had written each version separately.

A C++ template uses "**Instantiation-style polymorphism**".

- For example, the template **MyClass<T>** isn't really a generic class that can be compiled to code.
- Only the result of instantiation of the template can be compiled.

As an example, the **string** class is an instantiation of the **basic\_string** class template with a **char** type.

- Strings are objects that represent sequences of characters.
- **basic\_string** supports all the operations of a sequence container.

```
typedef basic_string<char> string;
```

# C++ TEMPLATES

---

Templates allow functions and classes to be instantiated for generic types.

- Templates allow a function or class to work on many different data types without being rewritten for each one.

```
template <typename T1, typename T2>
class MyPair
{
private:
    T1 first;
    T2 second;
public:
    MyPair(T1 f, T2 s) : first(f), second(s) { }
    T1 getFirst() { return first; }
    T2 getSecond() { return second; }
};
```

Discuss: what's happening here?

```
MyPair<string, int> myDog("Dog", 36);
MyPair<double, double> myFloats(3.0, 2.18);
```

Allocated in stack or heap?



# STL – STANDARD TEMPLATE LIBRARY

---

Collections of useful classes for common data structures

Ability to store objects of any type (template)

Study of containers

Containers form the basis for treatment of data structures

Container – class that stores a collection of data

STL consists of 10 container classes:

- Sequence containers
- Adapter containers
- Associative containers



# STL CONTAINERS

---

## Sequence Container

- Stores data by position in linear order:
- First element, second element , etc:

## Associate Container

- Stores elements by key, such as name, social security number or part number
- Access an element by its key which may bear no relationship to the location of the element in the container

## Adapter Container

- Contains another container as its underlying storage structure

# STL CONTAINERS

---

- Sequence Container
  - Vector
  - Deque
  - List
- Adapter Containers
  - Stack
  - Queue
  - Priority queue
- Associative Container
  - Set, multiset
  - Map, multimap

We will study these later in the course

# VECTOR CONTAINER

---

Generalized array that stores a collection of elements of the same data type

Vector – similar to an array

- Vectors allow access to its elements by using an index in the range from 0 to  $n-1$  where  $n$  is the size of the vector

Vector vs array

- Vector has operations that allow the collection to grow and contract dynamically at the rear of the sequence

# VECTOR CONTAINER

---

Example:

```
#include <vector>

.
.
.

vector<int> scores (100);           //100 integer scores
vector<Passenger> passengerList(20); //list of 20 passengers
```

# THE VECTOR

---

A **vector** is a **template class** based on a dynamic array which means you can select its elements in arbitrary order as determined by the subscript value.

In addition, a vector supports the following operations that an array does not:

- Increase or decrease its length.
- Insert an element at a specified position without writing code to shift the other elements to make room.
- Remove an element at a specified position without writing code to shift the other elements to fill in the resulting gap.

*Vectors are used most often when a programmer wants to add new elements to the end of a list but still needs the capability to access, in arbitrary order, the elements stored in the list.*

# REFLECTION

---

```
std::string s1 = "a new string";
std::string s2 = s1;
assert(&s2==&s1); // True or False?
```

```
std::string * s3 = &s1;
std::string * s4 = s3;

assert(s4==&s1); // True or False?
assert(s4==s3); // True or False?
```

# REFERENCES AND POINTERS

---

In addition to **pointer** types, C++ provides **reference** types.  
Before that, we need to discuss what are **lvalues** and **rvalues**.

## Definition:

An *lvalue* is an expression that identifies a non-temporary object (variable identifier).

## Examples:

```
vector<string> arr( 3 );
const int x = 2;
int y;
...
int z = x + y;
string str = "foo";
```

In the examples, what comes in the left-hand side (`arr`, `str`, `x`, `y`, `z`) are **lvalues**.



# REFERENCES AND POINTERS

---

Variables (**lvalues**) store results of computations (**rvalues**).

## Definition:

An *rvalue* is an expression that identifies a temporary object or value not associated with any object.

In the examples, what comes in the right-hand side (2, "foo",  $x+y$ ) are **rvalues**.

## Examples:

```
vector<string> arr( 3 );
const int x = 2;
int y;
...
int z = x + y;
string str = "foo";
```

Discuss:  
Is there a rvalue for the following? If so, what is it?  
`vector<string> arr(3);`

# REFERENCES AND POINTERS

---

A **reference type** allows us to define a **new name** for an existing value. In classic C++, a **reference** can generally only be another name for an **lvalue**, defined by placing an & after some type:

```
string str = "hell";
string & rstr = str;           // rstr is another name for str
rstr += 'o';                 // changes str to "hello"
bool cond = (&str == &rstr);  // true; str and rstr are same object
string & bad1 = "hello";      // illegal: "hello" is not a modifiable lvalue
string & bad2 = str + "";     // illegal: str+"" is not an lvalue
string & sub = str.substr( 0, 4 ); // illegal: str.substr( 0, 4 ) is not an lvalue
```

Notice how operations can be performed over **references** too.

# REFERENCES AND POINTERS

---

*But why are references **useful**?*

The simplest case is to create **synonyms** for complicated expressions (notice the use of `auto` for **automatic typing**):

```
auto & whichList = theLists[ myhash( x, theLists.size( ) ) ];
if( find( begin( whichList ), end( whichList ), x ) != end( whichList ) )
    return false;
whichList.push_back( x );
```

Without the **reference**, the first line would actually **create a copy**, and the rest would operate over the copy:

```
auto whichList = theLists[ myhash( x, theLists.size( ) )];
```

# REFERENCES AND POINTERS

---

Another possibility is to **avoid unnecessary copies**. Suppose a function `findMax` returns the maximum in a collection.

Suppose further that the collection stores **large objects**. The assignment operator will by default make a copy:

```
auto x = findMax( arr );
```

We can avoid the copy by **obtaining a reference**. Notice that the syntax needs to be adjusted to receive and return references:

```
auto & x = findMax( arr );
```

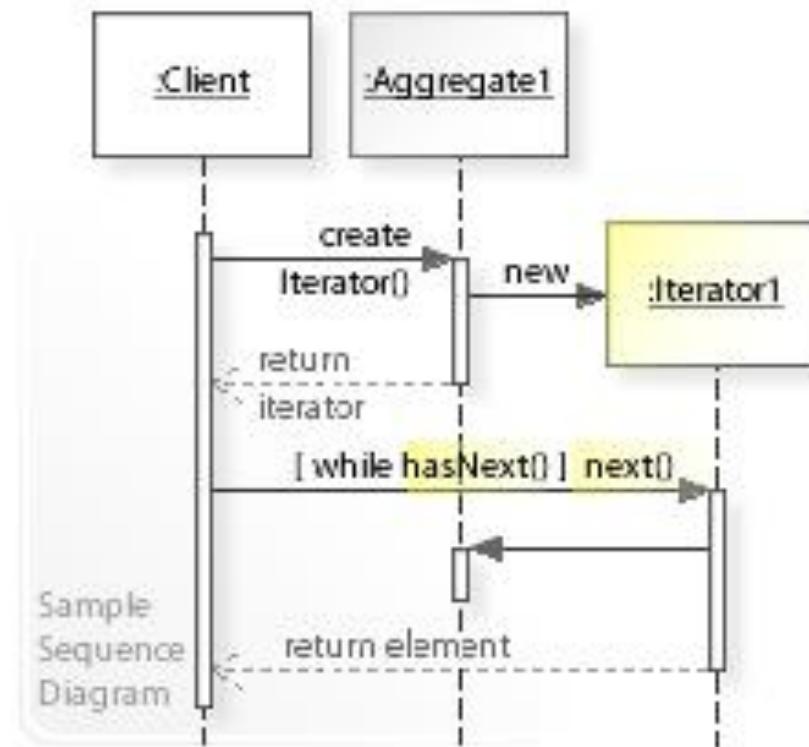
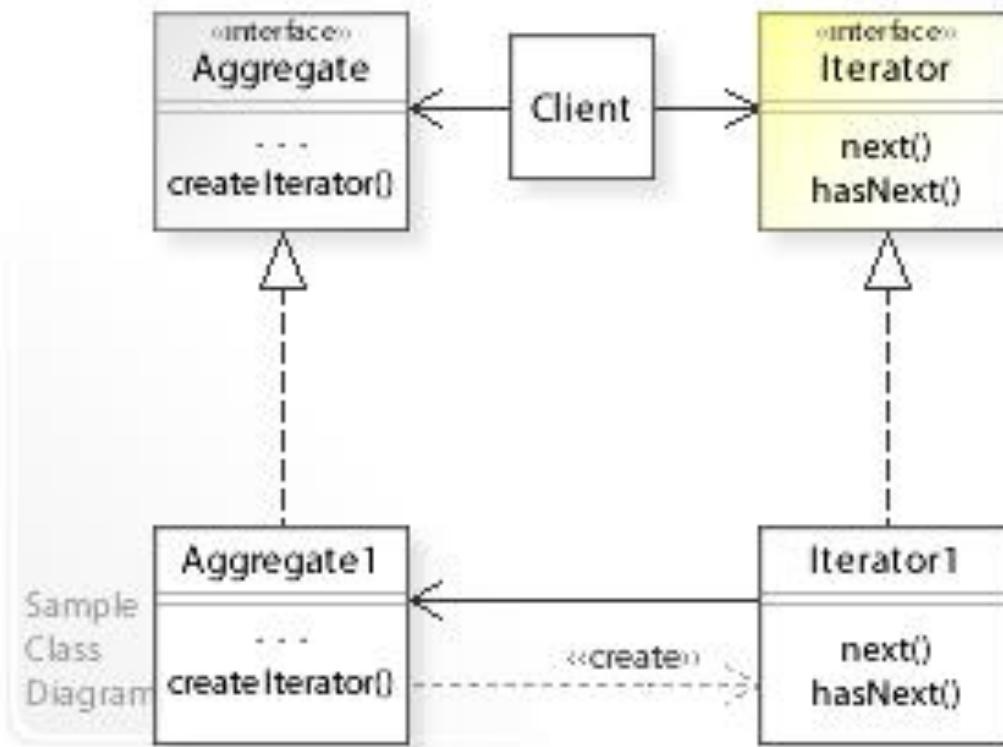
# REFLECTION

---

```
std::string s1 = "a new string";
std::string & s2 = s1;
assert(&s2==&s1); // True or False?
```

# ITERATOR PATTERN

---



[https://en.wikipedia.org/wiki/Iterator\\_pattern](https://en.wikipedia.org/wiki/Iterator_pattern)



# STL ITERATORS

---

- An object that can “iterate” over elements. May be all or part of a STL container.
- An iterator represents a certain position in a container.
- Operator \*

  - Returns the element of the actual position.

- Operator ++

  - Lets the iterator step forward to the next element.
  - Most iterators also allow stepping backwards by using operator --

# (CONT.)

---

Operator == and !=

- Returns whether two iterators represent the same position.
- Operator =
  - Assigns an iterator (the position of the element to which it refers)

# ITERATOR RELATED MEMBER FUNCTIONS

`begin( )`

- returns an iterator that represents the beginning of the elements in the container.

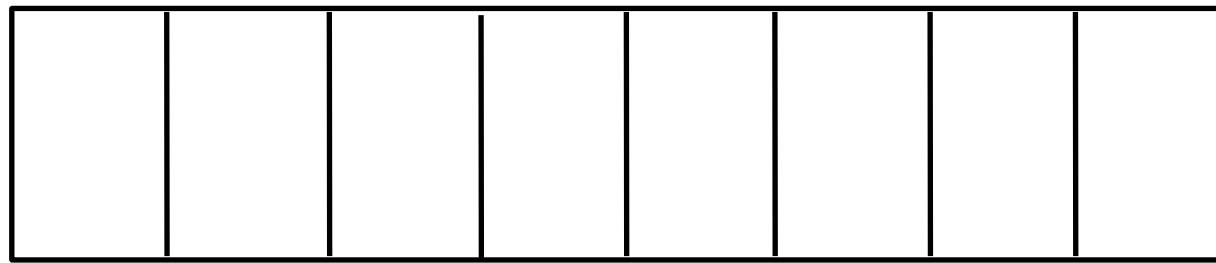
`end( )`

- returns an iterator that represents the end of the elements in the container.
- The end position is beyond the last element.

**begin ( )**



**end ( )**



# ITERATOR EXAMPLE

---

```
// C++ code to demonstrate the working of
// iterator, begin() and end()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{
    vector<int> ar = { 1, 2, 3, 4, 5 };

    // Declaring iterator to a vector
    vector<int>::iterator ptr;

    // Displaying vector elements using begin() and end()
    cout << "The vector elements are : ";
    for (ptr = ar.begin(); ptr < ar.end(); ptr++)
        cout << *ptr << " ";

    return 0;
}
```

# REFLECTION

---

```
vector<int> ar = {1, 2, 3, 4, 5};  
vector<int>::iterator ptr;  
ptr++;  
assert(*ptr==11); // True or False?
```

```
vector<int> ar = {1, 2, 3, 4, 5};  
vector<int>::iterator ptr = ar.begin();  
ptr++;ptr++;  
assert(*ptr==3); // True or False?  
  
ptr += 2;  
(*ptr) = 10;  
  
assert(ar[3] == 10); // True or False?  
assert(ar[4] == 10); // True or False?  
assert(ar[5] == 10); // True or False?
```

# LINEAR SEARCH

---

First, define the **computational problem** we are trying to solve.

## Array search problem:

*Given an unsorted list of entries  $A_0, A_1, A_2, \dots, A_{N-1}$  and a key  $x$ , find the smallest  $i$  such that  $x = A_i$ . If there is no such  $i$ , return error.*

# LINEAR SEARCH

---

- We implemented linear search as an **external function**. *Why?*
- The **range** of elements is defined by iterators (*end* is invalid) and *value* is the *key* to **search**.
- The iterator is **robustly** moved by `std::next()`.

```
1 using namespace std;
2
3 /**
4  * Performs the standard linear search using one comparison per item.
5 */
6 template <typename iterator, typename Object>
7 iterator linearSearch(iterator first, iterator last, const Object& value) {
8     iterator begin = first;
9     iterator end = last;
10    while(begin != end) {
11        if(*begin == value) {
12            return begin;
13        } else {
14            begin = std::next(begin);
15        }
16    }
17    return end;
18 }
```

linear\_search.h

# LINEAR SEARCH

---

The testing code does not offer many surprises.

test\_linear\_search.cpp

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4
5 #include "linear_search.h"
6
7 using namespace std;
8
9 #define N      100
10 #define MAX    1000
11
12 int main(void) {
13     int i, j;
14     vector<int> v;
15     vector<int>::iterator itr;
16
17     srand(time(0));
18     for (i = 0; i < N; i++) {
19         v.push_back(rand() % MAX);
20     }
21
22     /* Now let us print the vector through iterator */
23     for(itr = v.begin(); itr != v.end(); itr++) {
24         cout << *itr << " ";
25     }
26     cout << endl;
27
28     j = rand() % MAX;
29
30     /* Now search using linear search. */
31     itr = linearSearch(v.begin(), v.end(), j);
32     if (itr != v.end()) {
33         cout << "Found element: " << *itr << endl;
34     } else {
35         cout << "Element not found: " << j << endl;
36     }
37
38     return 0;
39 }
```

# BIG-O, ADT, AND LISTS

# AGENDA

---

## Introduction to program analysis

Growth of functions

Notions of algorithm complexity

Benchmarking and efficiency

Data structures, Abstract Data Types (ADT), and Pre/Post Conditions

Linked Lists and Operations

Doubly Linked Lists and Operations

# INTRODUCTION TO PROGRAM ANALYSIS

---

*What is the **best** algorithm to solve a certain problem?*

**Analysis** is key for understanding algorithms well to apply them to solve practical problems.

However, we cannot do extensive experimentation and deep mathematical analysis of all programs we run.

We need a **basic framework** for performance analysis to **analyze and compare** algorithms so we can apply them to solve our problems.

**How can we solve this?**

# INTRODUCTION TO PROGRAM ANALYSIS

---

Let us first try **empirical analysis** (implement the algorithms and measure!):

- It takes time to implement all relevant algorithms
- What is the **right** input data? (*real, good, random, ideal, perverse*)
- What is the **right** platform? (*compiler, CPU clock, memory, operating system*)
- What if the problem is **hard** and all algorithms take too much time to run?

There are many **pitfalls** in selecting algorithms:

- **Ignoring** performance characteristics. Faster algorithms are often **more complicated**.
- Paying **too much** attention to performance. "*Premature optimization is the root of all evil.*" (*Don Knuth*)

**Resources are finite** (your time), spend wisely on improving the algorithms that **matter**.

# INTRODUCTION TO PROGRAM ANALYSIS

---

Just to illustrate the effect of **the compiler**, compile and run the program below with different **optimization settings (-O)** using the time command:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    int N;
    long count = 0;
    printf("input N: ");
    scanf("%d", &N);
    int msec = 0, trigger = 10; /* 10ms */
    clock_t before = clock();
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                count++;
            }
        }
    }
    clock_t taken = clock() - before;
    msec = taken * 1000/CLOCKS_PER_SEC;
    printf("count = %ld, msec taken = %d", count, msec);
}
```

count.c

```
~/countc$ clang-7 -pthread -o main main.c
~/countc$ ./main
input N: 1000
count = 1000000000, msec taken = 2260
~/countc$ clang-7 -pthread -O1 -o main main.c
~/countc$ ./main
input N: 1000
count = 1000000000, msec taken = 0
~/countc$ 
```

Without optimization

With optimization

# INTRODUCTION TO PROGRAM ANALYSIS

---

There are many reasons to analyze algorithms:

- To **compare** different algorithms for the same task
- To **predict** performance in a new environment
- To **adjust parameters** for a certain algorithm
- To **know how much resources** are needed to solve the task

Performance characteristics of many algorithms is known, but that is not **always** the case.

Discovering that in many cases leads to **new mathematical insights**.

→ **Analysis of Algorithms** is the CS field that studies **complexity analysis** (in *time* and *space*):

- The *analyst* discovers **as much as** possible about an algorithm
- The *programmer* **applies** such information to **select** an algorithm

# INTRODUCTION TO PROGRAM ANALYSIS

---

We now can **separate** *analysis* from *implementation*:

- What is the **abstract** operation executed by the algorithm? (*time*)
- What is the **unit of memory** required to run the algorithm? (*space*)

Hence we can analyze algorithms no matter **how much the abstract operation runs** in nanoseconds or **how many bits an integer has** in a certain machine.

We are implicitly defining a *model of computation*:

- Simple operations take **1 unit of time**
- Simple variables take **1 unit of storage**

→ We can then **approximately count** these quantities for **ideal** input (*best case*), **random** input (*average case*), **bad** input (*worst case*).

# REFLECTION

---

Consider the following problem:

$$a + b + c = n$$

Find all sets of nonnegative integers  
 $(a, b, c)$  that sum to integer  $n$  ( $n \geq 0$ )

And the following two solutions, made by Alice and Bob (the pseudo code is presented, but both Bob and Alice use only pen and paper to solve the problem for a given  $n$ ):

## Bob's Solution

1. Try all combinations of  $(a, b, c)$
2. If  $a + b + c = n$ , print( $a, b, c$ )

## Alice's Solution

1. For all  $(a, b)$  set  $c = n - (a + b)$
2. If  $c \geq 0$ , print( $a, b, c$ )

Discuss:

1. Which one is the most efficient of the two, and why?
2. Did you consider the fact that Bob (respectively Alice) might be faster with pen and paper than Alice (resp. Bob)? If so, why? If not, why?

# AGENDA

---

Introduction to program analysis

## **Growth of functions**

Notions of algorithm complexity

Benchmarking and efficiency

Data structures and Abstract Data Types (ADT)

Linked Lists and Operations

Doubly Linked Lists and Operations

# GROWTH OF FUNCTIONS

---

Most of algorithms have a **primary parameter  $N$**  that affects the runtime substantially.

*What is this **primary parameter?***

Depending on the *problem*, the **input size** can be the *degree of a polynomial, the size of a file, the length of a string, the dimensions in an image, etc.*

If there are **multiple parameters** (like the dimensions on an image), we express one parameter as a function of the other, so we have a single parameter  $N$ .

→ **Objective:** is to express **all the resources** as a function  $T$  of just  $N$ , typically in the **worst case**.

# REFLECTION

---

What is the primary parameter ( $N$ )  
in Bob's solution to the problem on the right?

$$a + b + c = n$$

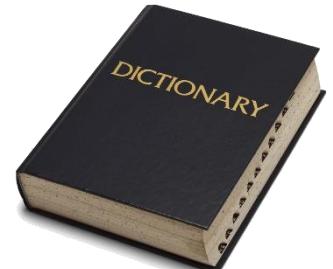
Find all sets of nonnegative integers  
( $a, b, c$ ) that sum to integer  $n$  ( $n \geq 0$ )

Bob's Solution

1. Try all combinations of ( $a, b, c$ )
2. If  $a + b + c = n$ , print( $a, b, c$ )

Imagine that you need to find a word in a dictionary like the one on the right.

- What does your algorithm look like?
- What is the primary parameter ( $N$ )?



<https://www.collinsdictionary.com/dictionary/english/dictionary>

# GROWTH OF FUNCTIONS

---

If we can write the number of steps that algorithms take as **functions**, then we can see how they grow as the primary parameter increases!

**Note:** this restricts the domain of the functions to the **natural numbers!**

For example, see the values  $T(N)$  for **different** ( $N, T$ ):

$\lg N$	$\sqrt{N}$	$N$	$N \lg N$	$N(\lg N)^2$	$N^{3/2}$	$N^2$
3	3	10	33	110	32	100
7	10	100	664	4414	1000	10000
10	32	1000	9966	99317	31623	1000000
13	100	10000	132877	1765633	1000000	1000000000
17	316	100000	1660964	27588016	31622777	100000000000
20	1000	1000000	19931569	397267426	10000000000	100000000000000

# GROWTH OF FUNCTIONS

---

What if these are **real programs** running in **real machines**?

operations per second	problem size 1 million			problem size 1 billion		
	$N$	$N \lg N$	$N^2$	$N$	$N \lg N$	$N^2$
$10^6$	seconds	seconds	weeks	hours	hours	never
$10^9$	instant	instant	hours	seconds	seconds	decades
$10^{12}$	instant	instant	seconds	instant	instant	weeks

→ **Important:** it is likely that the runtime of a program will involve constants and minor terms. We **ignore** those and consider only the *dominant term*.

# GROWTH OF FUNCTIONS

---

Time will **almost always** proportional to *one of these functions*:

- *Constant (1)*: basic operations are executed at most a few times.
- *Logarithmic or  $\log(N)$* : solves problem by breaking in pieces and solving a smaller piece. Notice that base of the log **does not matter**.
- *Linear ( $N$ )*: a small amount of work is done per input element.
- *Linearithmic or  $N\log(N)$* : usually arises in **recursive algorithms**
- *Quadratic ( $N^2$ )*: solution involves **all pairs** of input elements.
- *Cubic ( $N^3$ )*: solution involves **all triples** of input elements.
- *Exponential ( $2^N$ )*: typically brute-force solution by testing **all solutions**.

**Important:** same idea applies to *space!* In many times, **time and space are closely related**.

# GROWTH OF FUNCTIONS

---

There will be other **functions and constants** as well.

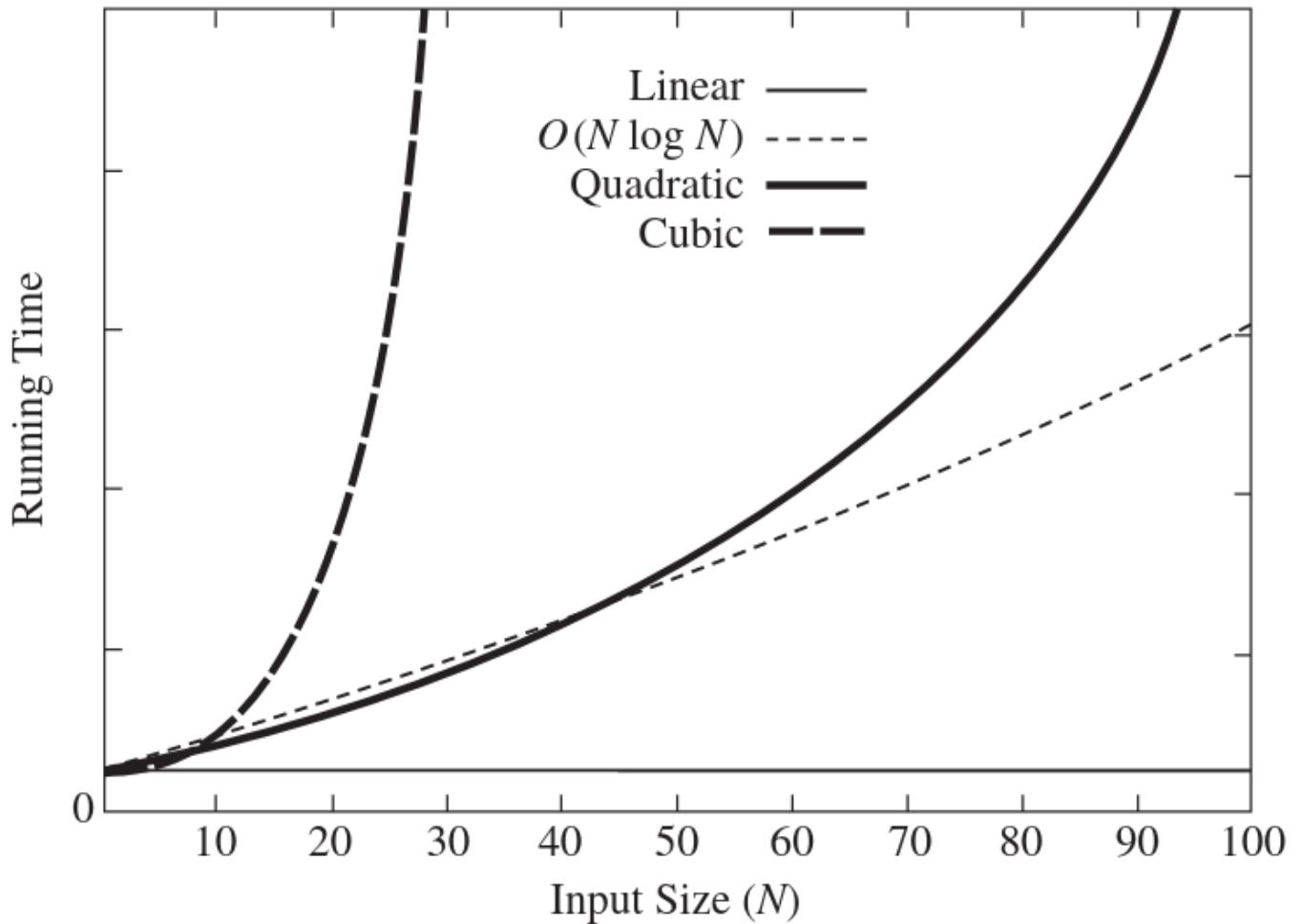
<i>function</i>	<i>name</i>	<i>typical value</i>	<i>approximation</i>
$[x]$	floor function	$[3.14] = 3$	$x$
$\lceil x \rceil$	ceiling function	$\lceil 3.14 \rceil = 4$	$x$
$\lg N$	binary logarithm	$\lg 1024 = 10$	$1.44 \ln N$
$F_N$	Fibonacci numbers	$F_{10} = 55$	$\phi^N / \sqrt{5}$
$H_N$	harmonic numbers	$H_{10} \approx 2.9$	$\ln N + \gamma$
$N!$	factorial function	$10! = 3628800$	$(N/e)^N$
$\lg(N!)$		$\lg(100!) \approx 520$	$N \lg N - 1.44N$
		$e = 2.71828\dots$	
		$\gamma = 0.57721\dots$	
		$\phi = (1 + \sqrt{5})/2 = 1.61803\dots$	
		$\ln 2 = 0.693147\dots$	
		$\lg e = 1/\ln 2 = 1.44269\dots$	

# GROWTH OF FUNCTIONS

---

Let us plot some of these functions and see how they grow with  $N$ .

Notice that some of them **intersect!**



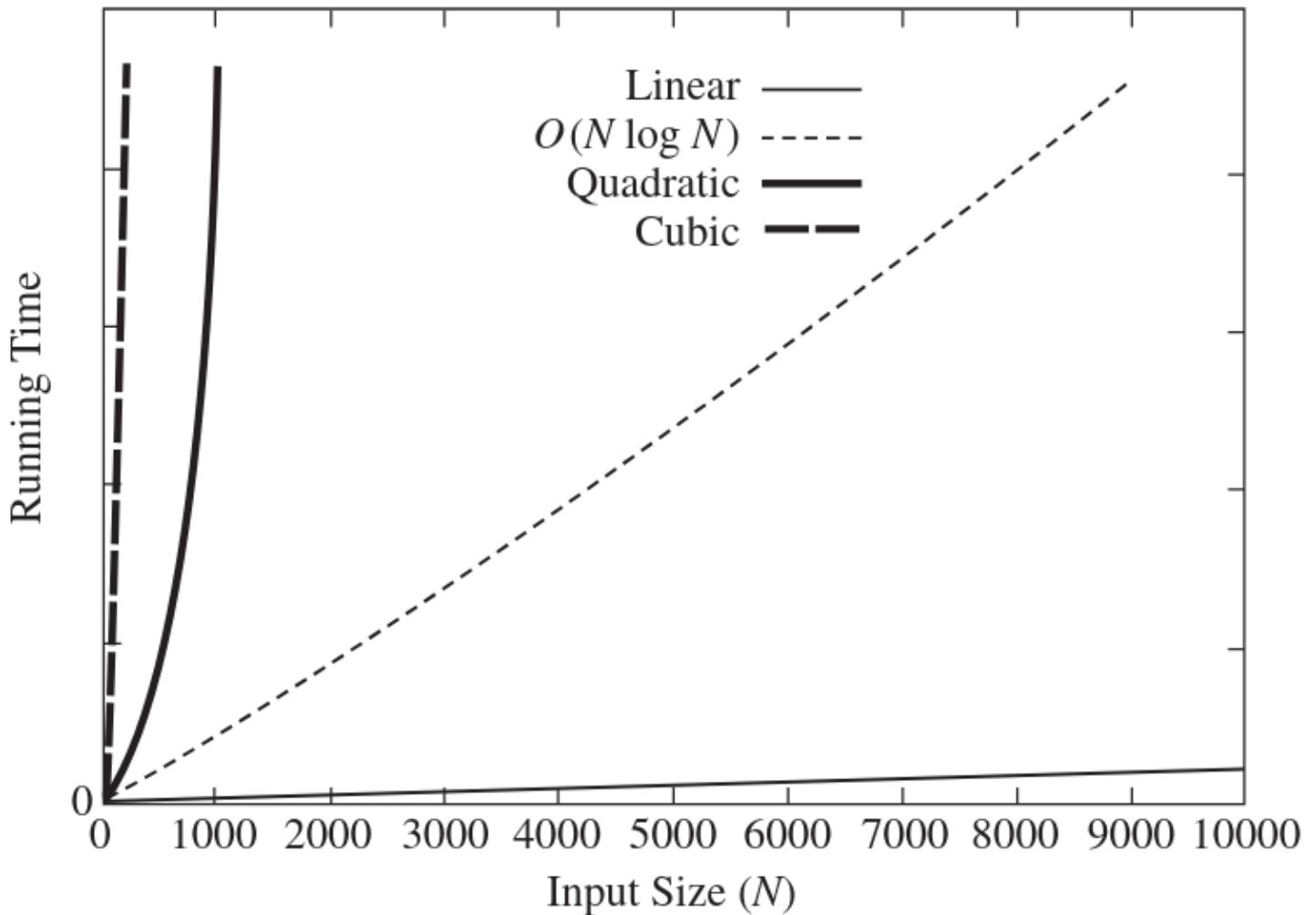
# GROWTH OF FUNCTIONS

---

We are *particularly* interested in how they behave for **large values of N!**

*But why?*

This is also called the *asymptotic behavior*, i.e., **when N goes to infinity.**



# REFLECTION

---

Recall the following problem:

$$a + b + c = n$$

Find all sets of nonnegative integers  
 $(a, b, c)$  that sum to integer  $n$  ( $n \geq 0$ )

And the following two solutions, made by Alice and Bob (the pseudo code is presented, but both Bob and Alice use only pen and paper to solve the problem for a given  $n$ ):

## Bob's Solution

1. Try all combinations of  $(a, b, c)$
2. If  $a + b + c = n$ , print( $a, b, c$ )

## Alice's Solution

1. For all  $(a, b)$  set  $c = n - (a + b)$
2. If  $c \geq 0$ , print( $a, b, c$ )

Discuss:

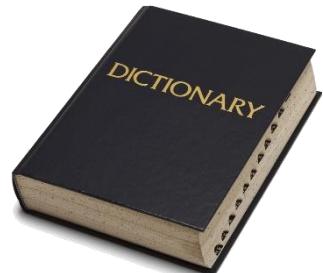
1. How many steps, given a value for  $n$ , will Bob take? What about Alice?

# REFLECTION

---

Imagine that you have to find a word in a dictionary like the one on the right.

- How many steps does your algorithm take to find a given word, as a function of the number of pages?



<https://www.collinsdictionary.com/dictionary/english/dictionary>

# AGENDA

---

Introduction to program analysis

Growth of functions

**Notions of algorithm complexity**

Benchmarking and efficiency

Data structures, Abstract Data Types (ADT), and Pre/Post Conditions

Linked Lists and Operations

Doubly Linked Lists and Operations

# NOTIONS OF ALGORITHM COMPLEXITY

---

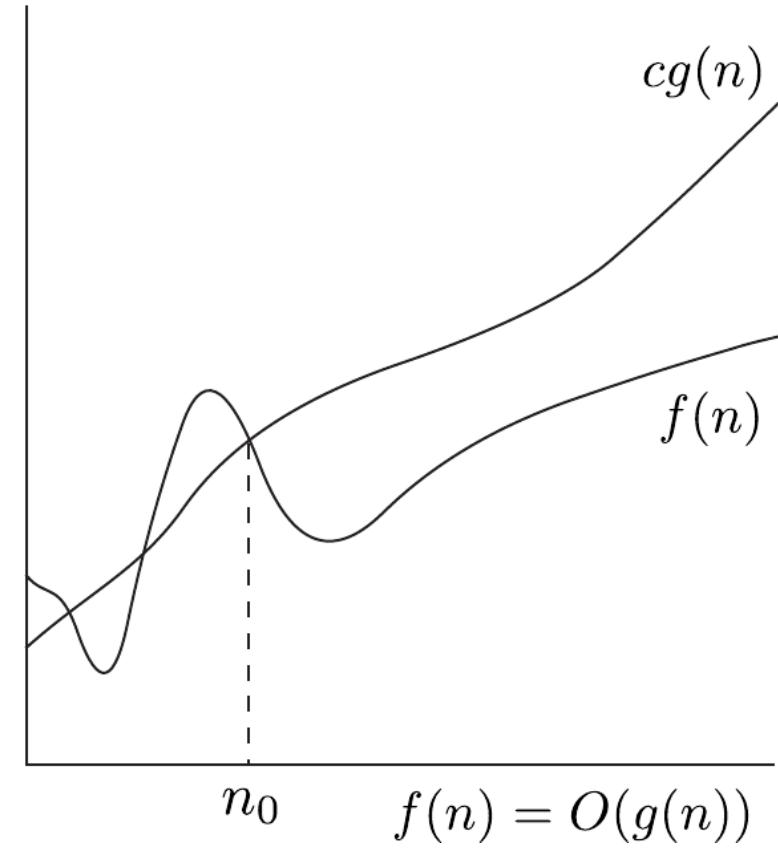
Our main tool will be  **$O$ -notation** (*read "Big-Oh"*) which formalizes the notion of **being proportional**, so we can capture the **overall performance** of an algorithm.

## Definition:

A function  $f(N)$  is said to be  $O(g(N))$  if there exist constants  $c$  and  $n_0$  such that

$$0 \leq f(N) \leq c * g(N)$$

for  $c > 0$  and all  $N \geq n_0$ .



# NOTIONS OF ALGORITHM COMPLEXITY

---

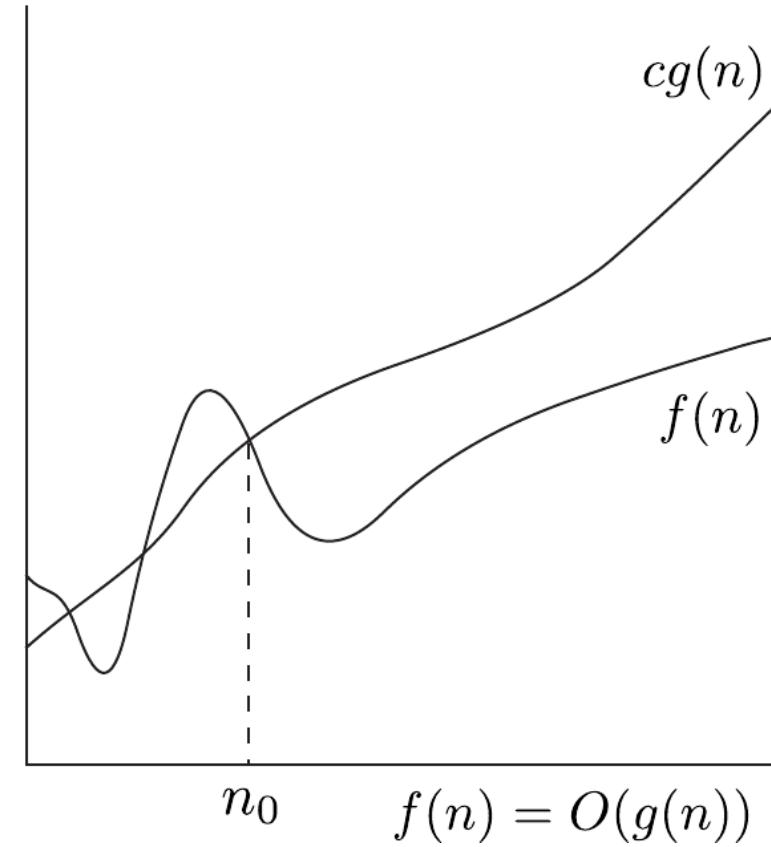
Another way to see the **same definition** if you are more inclined to **Calculus** and **continuous functions** can be found below.

## Definition:

A function  $f(N)$  is  $O(g(N))$  if

$$\lim_{N \rightarrow \infty} (f(N)/g(N)) \in [0, \infty),$$

which limits the growth rate of  $f$  up to a constant times the growth rate of  $g$ .



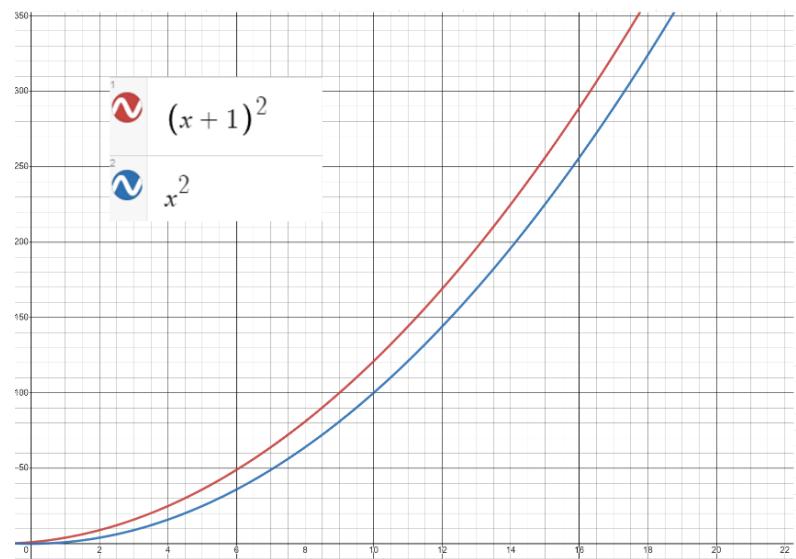
# NOTIONS OF ALGORITHM COMPLEXITY

---

Example: Consider an algorithm that takes  $(n + 1)^2$  steps.

Why can we just claim that it is  $O(n^2)$ , and why not  $O((n + 1)^2)$ ? In other words, why do we care about simpler approximations and large  $n$ ?

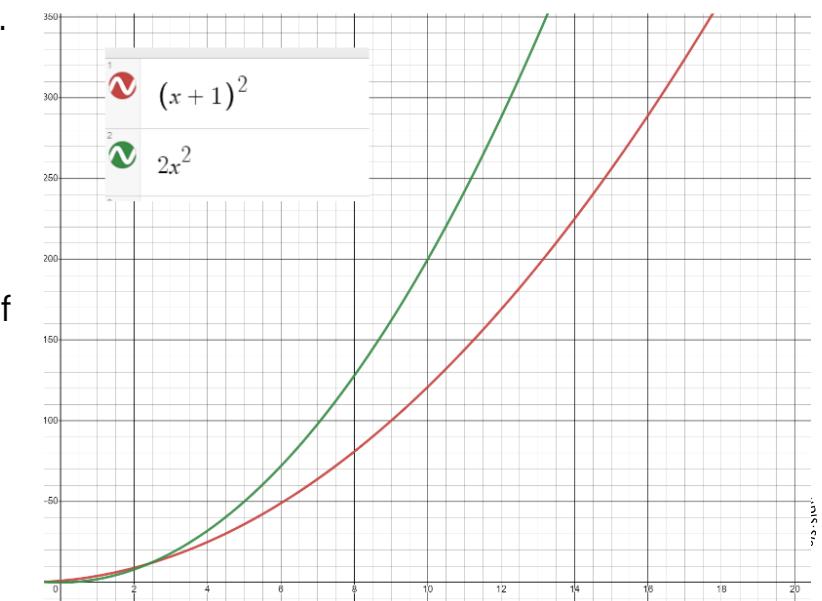
Consider the plot below and note that the blue function (our estimate for the algorithm's cost) is always below the red one. So it's not really a good estimate, right...?



... but since we abstract away from the cost of each individual step taken by the algorithm, we make that assumption that it has some constant cost  $c$ .

So imagine that now each instruction takes  $c = 2$  ns to run. Now the real cost of the algorithm is  $2 * (n + 1)^2$

Now we see in the plot below that the estimate is no longer that bad at all...



This is why it is often enough, in the worst case analysis, to consider the terms with the highest order in the expression describing the number of steps.

# REFLECTION

---

Apply the definition of  $O(..)$  to the right and prove that  $f(n) = (n + 1)^2$  is  $O(n^2)$ .

Hint: use the plots in the previous slide to find  $c$  and  $n_0$ .

## Definition:

A function  $f(N)$  is said to be  $O(g(N))$  if there exist constants  $c$  and  $n_0$  such that

$$0 \leq f(N) \leq c * g(N)$$

for  $c > 0$  and all  $N \geq n_0$ .

# NOTIONS OF ALGORITHM COMPLEXITY

---

We use the *O-notation* to simplify the analysis in a **technically precise sense**:

- To bound the error that we make when we **ignore constants and small terms**.
- To allow us to classify algorithms based on an **upper bound** of their running times.

We can now speak of an algorithm being **asymptotically** faster or slower than another.

# REFLECTION

---

Show that  $N \log(N) = O(N^{3/2})$ .

**Reminder:** we are doing *worst-case analysis!*

# NOTIONS OF ALGORITHM COMPLEXITY

---

Consider the following code fragment to compute  $\sum i^3$  for  $i = 1 .. N$ .

```
1 int sum(int n) {  
2     int i, partialSum = 0;  
3  
4     for (i = 1; i <= n; i++) {  
5         partialSum = partialSum + (i * i * i);  
6     }  
7     return partialSum;  
8 }
```

partialsum.c

- **Line 2** costs 1 initialization.
- **Line 4** costs 1 initialization,  $(N+1)$  comparisons and  $N$  increments.
- **Line 5** costs 2 multiplications, 1 addition and 1 assignment per iteration.
- **Line 7** costs 1 assignment.
- **Total:**  $T(N) = 6N+4$  operations.
- **Complexity:**  $T(N) = O(N)$

# NOTIONS OF ALGORITHM COMPLEXITY

---

Consider the following code fragment to compute  $\sum i^3$  for  $i = 1 .. N$ .

```
1 int sum(int n) {  
2     int i, partialSum = 0;  
3  
4     for (i = 1; i <= n; i++) {  
5         partialSum = partialSum + (i * i * i);  
6     }  
7     return partialSum;  
8 }
```

partialsum.c

- **Line 2** costs  $O(1)$ .
- **Line 4** costs  $O(N)$  in total.
- **Line 5** costs  $O(1)$  per iteration.
- **Line 7** costs  $O(1)$ .
- **Total:**  $O(N) + 3O(1) = O(N)$ .
- **Complexity:**  $T(N) = O(N)$

# NOTIONS OF ALGORITHM COMPLEXITY

---

Some simple rules for **quick analysis**:

- *Loops*: multiply cost of the body by the number of iterations.
- *Nested loops*: the same, but be **careful** about numbers of iterations.
- *Conditionals*: cost of the test plus the larger of the two branches.

*What about recursive functions?*

*We come back to these later.*

# REFLECTION (1/2)

---

Analyze the time complexity of **insertion sort**.

```
1  /*
2   * Insertion sort
3   */
4
5  #include <assert.h>          /* assert */
6
7  void swap(int a, int b, int *x, int n) {
8      /* pre-condition */
9      /* a and b are within bounds of x array */
10     assert(0 <= a && a < n && 0 <= b && b < n);
11
12     /* post-condition */
13     int t; /* temporary variable for swapping */
14     t = x[a];
15     x[a] = x[b];
16     x[b] = t;
17 }
```

insertion\_sort.c

# REFLECTION (2/2)

---

Analyze the time complexity of **insertion sort**.

```
19 void insertion_sort(int n, int *x) {
20     int i; /* counter variable */
21     int j; /* counter variable */
22
23     /* pre-condition */
24     assert(0 <= n);
25
26     /* post-condition: x[0..n] is sorted */
27     for (i = 1; i < n; i = i + 1) {
28         /* invariant: x[0..i-1] is sorted */
29         j = i;
30         /* insert next element j into
31          * ordered position
32          */
33         for (; j > 0; j = j - 1) {
34             if (x[j] > x[j - 1])
35                 break;           /* jth element is in correct position */
36             else
37                 swap(j, j - 1, x, n);
38         }
39     }
40 }
```

insertion\_sort.c

# AGENDA

---

Introduction to program analysis

Growth of functions

Notions of algorithm complexity

**Benchmarking and efficiency**

Data structures, Abstract Data Types (ADT), and Pre/Post Conditions

Linked Lists and Operations

Doubly Linked Lists and Operations

# BENCHMARKING AND EFFICIENCY

---

Running experiments in modern computers is **non-trivial**:

1. **Differences** in processor, memory, compiler, operating system
2. Processors **increase** performance depending on the **environment**
3. It may be hard to decide what a **realistic** workload is
4. It is **easy** (or even *tempting*) to benchmark on **optimistic conditions**

When running experiments, try to standardize **conditions** as much as possible. You can generate random inputs through functions `srandom(time(0))` and `random()`.

**Important:** useful for simple experiments, but **insecure** for everything else!

# AGENDA

---

Introduction to program analysis

Growth of functions

Notions of algorithm complexity

Benchmarking and efficiency

**Data structures, Abstract Data Types (ADT), and Pre/Post Conditions**

Linked Lists and Operations

Doubly Linked Lists and Operations

# DATA STRUCTURES

---

**Storing** and **accessing** data in **containers** is a basic step for the solution of problems. You know the **advantages** of an array:

- We can efficiently **store**, **access** and **change** data in an array
- What is the **time complexity** of these operations?

However, arrays have their **limitations** (*which?*), and we need more **flexibility** when handling and **structuring** data.

**Remark.** In many problems, choosing the right **data structure** is critical for an elegant and efficient solution.

# DATA STRUCTURES

---

Arrays have several **limitations**:

- Their capacity is in principle **fixed at declaration time**
- Elements cannot be easily **removed**, so it is hard to support dynamic data

What if the number of elements to store (or even the maximum) is **not known in advance**?

Notice that this situation happens frequently **in practice**, for example when you are reading from a file or keyboard input.

# DATA STRUCTURES

---

You already know C-style implementations of data structures using **structs and pointers**. We will study a C++ implementation using higher-level abstraction through **classes and objects**.

We will define the abstract objects we want to manipulate and the operations we perform, initially without worrying about how to **represent** the data and to **implement** those operations.



*Abstraction is the most fundamental tool to build more general and robust computer programs.*

# ABSTRACT DATA TYPES

---

## Definition:

An *abstract data type (ADT)* is a value or set of values that can be accessed only through an *interface*.

You can also see ADTs as a **mathematical abstraction**, with public functions their **interface**.

The following slides show two alternative ways of defining abstract data types in C++

Regardless of the approach, the goal is the same: separate the interface from the implementation, and therefore hide (i.e., decouple) the implementation details from the client.

# ABSTRACT DATA TYPES

---

## Method 1: Use inheritance

```
C point.h
1 #pragma once
2 class Point
3 {
4     /*Represents a point in a
5      | two dimensional plane*/
6 public:
7     /*calculates the distance
8      | from this point to a */
9     virtual float distance(Point& a) = 0;
10    virtual float getX() = 0;
11    virtual float getY() = 0;
12};
```

```
C++ test_point.cpp
vector<PointImpl> array = vector<PointImpl>();

for (int i=0; i<N; i++){
    array.push_back(PointImpl());
}

for (i = 0; i < N; i++) {
    for (j = i + 1; j < N; j++) {
        if (array[i].distance(array[j]) < d) {
            cnt++;
        }
    }
}
```

Pass by ref

```
C pointimpl.h
1 #pragma once
2 #include "point.h"
3 #include <math.h>
4 class PointImpl:
5     public Point
6 {
7 private:
8     float x, y;
9 public:
10    PointImpl(float xp, float yp) {
11        x = xp;
12        y = yp;
13    }
14    PointImpl() {
15        x = 1.0 * rand() / RAND_MAX;
16        y = 1.0 * rand() / RAND_MAX;
17    }
18    ~PointImpl() {}
19    inline float getX() { return x; }
20    inline float getY() { return y; }
21    float distance(Point& a) {
22        float dx = x - a.getX();
23        float dy = y - a.getY();
24        return sqrt(dx * dx + dy * dy);
25    }
26};
```

The distance only needs another Point, and not another PointImpl.

# ABSTRACT DATA TYPES

---

Method 2: Use abstract classes (method preferred by the book).

```
1 #ifndef _POINT_H_
2 #define _POINT_H_
3
4 class Point {
5     private:
6         float x, y;
7
8     public:
9         Point();
10        ~Point();
11        float distance(Point a);
12    };
13
14 #endif
```

point.h

```
point > point.cpp > ...
point > point > ...
1 #include <math.h>
2 #include "point.h"
3 #include <stdlib.h>
4
5 /* Constructor of a point. */
6 Point::Point() {
7     x = 1.0 * rand() / RAND_MAX;
8     y = 1.0 * rand() / RAND_MAX;
9 }
10
11 /* Destructor of a point. */
12 Point::~Point() {}
13
14 /* Compute the distance to another point. */
15 float Point::distance(Point a) {
16     float dx = x - a.x;
17     float dy = y - a.y;
18     return sqrt(dx * dx + dy * dy);
19 }
```

point.cpp

# ABSTRACT DATA TYPES

---

- The constructor generates a **random** point
- Destructor is **trivial** (memory allocation is automatic)
- The `distance()` method can access the **internal data**.
- We can also implement methods to **get** each field for a **cleaner Interface**.

```
1 #include <math.h>
2 #include "point.h"
3
4 /* Constructor of a point. */
5 Point::Point() {
6     x = 1.0*random()/RAND_MAX;
7     y = 1.0*random()/RAND_MAX;
8 }
9
10 /* Destructor of a point. */
11 Point::~Point() {}
12
13 /* Compute the distance to another point. */
14 float Point::distance(Point a) {
15     float dx = x - a.x;
16     float dy = y - a.y;
17     return sqrt(dx*dx + dy*dy);
18 }
```

point.cpp

# ABSTRACT DATA TYPES

---

There are several new things in this testing code.

A *namespace* defines a *scope* for the symbols. We are *using* the default one.

We can mix with C library functions without a problem.

Dynamic memory allocation.

Calling a function or method.

```
1 using namespace std;
2
3 #include <iostream>
4 #include <stdlib.h>
5 #include <time.h>
6 #include <assert.h>
7
8 #include "point.h"
9
10 int main(int argc, char *argv[]) {
11     int i, j, cnt = 0;
12     srand(time(0));
13
14     if (argc != 3) {
15         cout << "Usage: test_point n d" << endl;
16         return -1;
17     }
18
19     int N = atoi(argv[1]);
20     float d = atof(argv[2]);
21
22     Point *array = new Point[N];
23
24     for (i = 0; i < N; i++) {
25         for (j = i + 1; j < N; j++) {
26             if (array[i].distance(array[j]) < d) {
27                 cnt++;
28             }
29         }
30     }
31     cout << cnt << " pairs within distance " << d << endl;
32 }
33 }
```

C++ input/output

test\_point.cpp

# ABSTRACT DATA TYPES

---

Now let's improve the **interface** and **functionality** of the ADT:

- We changed `distance()` to be a *static class function*, no instance is needed to call it.
- We added *friend* operators for equality and printing.
- Notice the implementation of the new functions.

point\_plus.h

```
1 #ifndef _POINT_H_
2 #define _POINT_H_
3
4 class Point {
5     private:
6         float x, y;
7
8     public:
9         Point();
10        ~Point();
11        const float X() { return x; }
12        const float Y() { return y; }
13        static float distance(Point a, Point b);
14        friend int operator==(Point a, Point b);
15        friend ostream& operator<<(ostream& t, Point p);
16    };
17
18 #endif
```

```
16 /* Compute the distance between points. */
17 float Point::distance(Point a, Point b) {
18     float dx = a.X() - b.X();
19     float dy = a.Y() - b.Y();
20     return sqrt(dx*dx + dy*dy);
21 }
```

```
22
23 #define EPSILON 0.001f
24
25 int operator==(Point a, Point b) {
26     return Point::distance(a, b) < EPSILON;
27 }
```

```
28
29 ostream& operator<<(ostream& t, Point p) {
30     cout << "(" << p.x << " , " << p.y << ")";
31     return t;
32 }
```

point\_plus.cpp

# ABSTRACT DATA TYPES

---

The updated test function shows how the **operators** are used:

- The **printing operator** can now be used as if Point was a basic type (line 25).
- The same applies to the **equality operator** (line 27).

We now can handle these objects without knowing how the class was **implemented**.

test\_point\_plus.cpp

```
1 using namespace std;
2
3 #include <iostream>
4 #include <stdlib.h>
5 #include <time.h>
6 #include <assert.h>
7
8 #include "point_plus.h"
9
10 int main(int argc, char *argv[]) {
11     int i, j, cnt = 0, equ = 0;
12
13     if (argc != 3) {
14         cout << "Usage: test_point_plus n d" << endl;
15         return -1;
16     }
17     srand(time(0));
18
19     int N = atoi(argv[1]);
20     float d = atof(argv[2]);
21
22     Point *array = new Point[N];
23
24     for (i = 0; i < N; i++) {
25         cout << array[i] << endl;
26         for (j = i + 1; j < N; j++) {
27             if (array[i] == array[j]) {
28                 equ++;
29             }
30             if (Point::distance(array[i], array[j]) < d) {
31                 cnt++;
32             }
33         }
34     }
35     cout << cnt << " pairs within distance " << d << endl;
36     cout << equ << " pairs of equal points " << endl;
37     delete [] array;
38 }
39
```

# REFLECTION

---

What are the pros and cons of each method (inheritance vs abstract classes)?

# EXAMPLE: INHERITANCE – APPROACH

---

- Specify **interface** using an **abstract base class**.
- Only **public pure virtual method** declarations.
- Still specify interface in separate header file.
- Separates interface and implementation, and allows multiple implementations of same interface **by inheritance**.

MaxHeap.h

```
C MaxHeap.h > ...
1 #pragma once
2 #include <vector>
3
4 using namespace std;
5
6 class MaxHeap {
7
8 public:
9     // is the heap empty?
10    virtual bool isEmpty() const = 0;
11
12    // number of elements in the heap
13    virtual int size() = 0;
14
15    // add an element to the heap
16    virtual void insert(const int x) = 0;
17
18    // find the maximum element in the heap
19    virtual const int findMax() const = 0;
20
21    // delete and return the maximum element of the heap
22    virtual int deleteMax() = 0;
23};
24
```

# EXAMPLE: INHERITANCE – APPROACH

- Implementation of specialized MaxHeap is now done using inheritance from the base class.
- Multiple implementations can co-exists – e.g. MaxHeapVector or MaxHeapList.
- Abstract base class cannot be instantiated but can be used as pointer and reference type.

```
MaxHeap > C maxheapvector.h > ...
1 #pragma once
2 using namespace std;
3 #include <vector>
4 #include "MaxHeap.h"
5
6 class MaxHeapVector : public MaxHeap
7 {
8 private:
9     vector<int> v;
10
11 public:
12     MaxHeapVector() {}
13     virtual ~MaxHeapVector() {}
14
15     virtual bool isEmpty() const {
16         // TBD
17     }
18
19     virtual int size() {
20         // TBD
21     }
22
23     virtual void insert(const int x) {
24         // TBD
25     }
26 }
```

MaxHeapVector.h

```
5 #include "MaxHeap.h"
6 #include "MaxHeapVector.h"
7 #include <iostream>
8
9 int main(void){
10     MaxHeap *m = new MaxHeapVector();
11     m->insert(2);
12     m->insert(3);
13     m->insert(-1);
```

Main.cpp

# EXAMPLE: ABSTRACT CLASS – APPROACH

- One approach is to specify **interface** using standard **public** and **private** parts in a class.
- **Private** implementation.
- **Public** method declarations.
- Specify interface in separate header file.
- Separates interface and implementation, but does not allow multiple implementations of same interface.

MaxHeap.h

```
C MaxHeap.h > ...
1 #pragma once
2 #include <vector>
3
4 using namespace std;
5
6 class MaxHeap {
7
8 private:
9     vector<int>* elements;
10
11 public:
12     MaxHeap();
13
14     // is the heap empty?
15     bool isEmpty();
16
17     // number of elements in the heap
18     int size();
19
20     // add an element to the heap
21     void insert(const int x);
22
23     // find the maximum element in the heap
24     const int findMax();
25
26     // delete and return the maximum element of the heap
27     int deleteMax();
28 }
```

# EXAMPLE: ABSTRACT CLASS – APPROACH

- Implementation using the private type (`vector<int>` in this case).
- Usage in the standard way – instantiating the defined class.
- Notice, no inheritance/ specialization – only single implementation.

```
MaxHeap.cpp > ...
1 #include "MaxHeap.h"
2
3 MaxHeap::MaxHeap() {
4     elements = new vector<int>();
5 }
6
7 bool MaxHeap::isEmpty() {
8     // TBD
9 }
10
11 int MaxHeap::size() {
12     // TBD
13 }
14
15 void MaxHeap::insert(const int x) {
16     // TBD
17 }
18
```

```
Main.cpp > ...
1 #include <iostream>
2 #include <stdlib.h>
3 #include "MaxHeap.h"
4 #include <cassert>
5
6 using namespace std;
7
8 int main(int argc, char* argv[]) {
9
10     MaxHeap heap = MaxHeap();
11
12     assert(heap.isEmpty());
13
14     heap.insert(0);
15     heap.insert(2);
```

# PRE/POST CONDITIONS

---

Idea: use assertions to enforce contracts between a function/object and its clients.

- Discuss:
  - is this a good idea or a bad one?
  - What does a contract mean?
  - What are the alternatives?

Example:

- <https://riptutorial.com/c/example/1810/precondition-and-postcondition>

Pros:

- Actively enforce contract.
- Obeys fail-first principle.
- Serve as documentation: details what a function/object does without showing how.

Cons:

- More code = more maintenance.

# AGENDA

---

Introduction to program analysis  
Growth of functions  
Notions of algorithm complexity  
Benchmarking and efficiency  
Data structures, Abstract Data Types (ADT), and Pre/Post Conditions

**Linked Lists and Operations**

Doubly Linked Lists and Operations

# LIST ADT

---

A *general list* is a sequential arrangement of the form  $A_0, A_1, A_2, \dots, A_{N-1}$ . We define the size of the list is  $N$ . The *empty list* has size 0.

For any list, we say that  $A_i$  **succeeds**  $A_{i-1}$  ( $i < N$ ), or that it **precedes** in the converse ( $i > 0$ ). The position of an element  $A_i$  in the list is  $i$ .

Given this **abstraction**, we need some operations to insert, remove or look for elements in the list.

The following slides show the `LinkedList` data structure, that implements the List ADT.

```
C simple_list.h
1 #pragma once
2
3 template <typename Object>
4 class List {
5
6 public:
7     virtual int size() = 0;
8     virtual bool empty() = 0;
9
10    virtual void clear() = 0;
11    virtual void push_front(const Object x) = 0;
12    virtual Object pop_front() = 0;
13    virtual Object find_kth(int pos) = 0;
14};
```

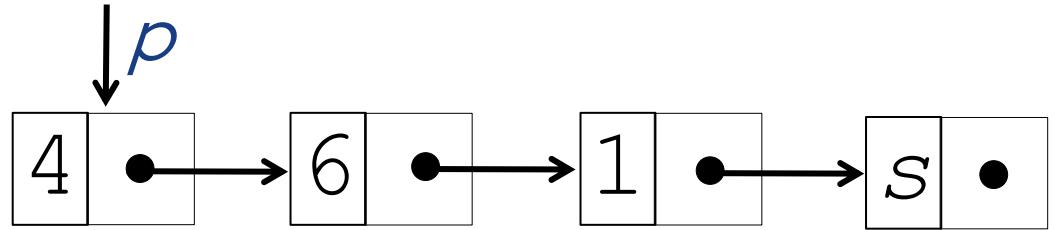
# LINKED LISTS

---

## Definition:

A *linked list* is a set of items where each item is part of a *node* that also contains a *link* to another node.

In C, pointers were **explicitly manipulated** to perform operations in the list. This **breaks the abstraction**.



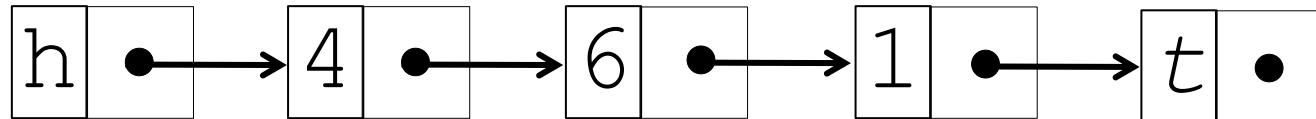
It is **typical** to implement linked lists using a *sentinel node* to represent the end of the list, but there are other **conventions** (NULL pointer or circular list).

# LINKED LISTS

---

We can get **better abstraction** by:

- **Hiding** the pointers (*interface*)
- Using a **private** Node type
- Using **head/tail** to *mark* the beginning and end of the list



Notice how the interface does not  
*leak implementation details.*

```
1 #ifndef _LIST_H_
2 #define _LIST_H_
3
4 class List {
5     private:
6         // A basic simple linked list node.
7         struct Node {
8             int value;
9             Node *next;
10        };
11
12     int theSize;
13     Node *head;
14     Node *tail; // previous sentinel
15
16 public:
17     List();
18     ~List();
19     int size();
20     bool empty();
21     void clear();
22     void push_front(int x);
23     void insert(int x, int pos);
24     int find_kth(int pos);
25     int pop_front();
26     int remove(int pos);
27 };
28
29 #endif
```

simple\_int\_list.h

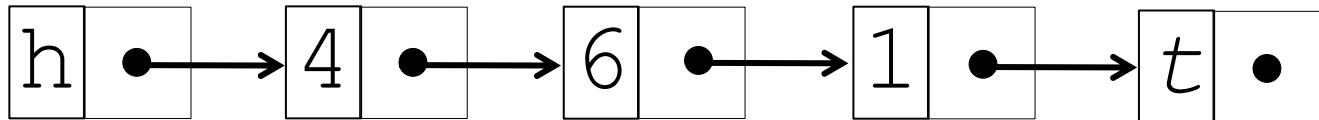
# LINKED LISTS

---

The constructor now builds the **two** marker nodes and initializes them.

The destructor deletes the **two nodes**.

Let us take a brief look at `clear()`.



```
1 #include <cassert>
2 #include "simple_int_list.h"
3
4 List::List() {
5     theSize = 0;
6     head = new Node;
7     tail = new Node;
8     head->next = tail;
9     tail->next = nullptr;
10 }
11
12 List::~List() {
13     clear();
14     delete head;
15     delete tail;
16 }
17
18 int List::size() {
19     return theSize;
20 }
21
22 bool List::empty() {
23     return (size() == 0);
24 }
25
26 void List::clear() {
27     Node *p = head->next;
28     while (p != tail) {
29         Node *t = p->next;
30         delete p;
31         p = t;
32         head->next = t;
33     }
34 }
```

simple\_int\_list.cpp

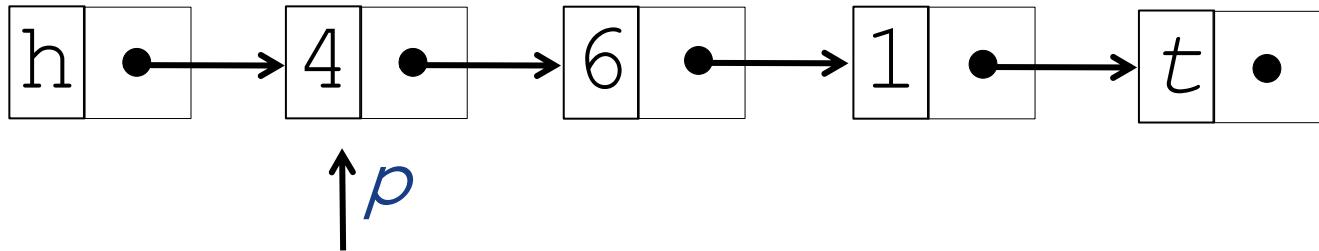
# LINKED LISTS

---

The constructor now builds the **two** marker nodes and initializes them.

The destructor deletes the **two nodes**.

Let us take a brief look at `clear()`.



```
1 #include <cassert>
2 #include "simple_int_list.h"
3
4 List::List() {
5     theSize = 0;
6     head = new Node;
7     tail = new Node;
8     head->next = tail;
9     tail->next = nullptr;
10 }
11
12 List::~List() {
13     clear();
14     delete head;
15     delete tail;
16 }
17
18 int List::size() {
19     return theSize;
20 }
21
22 bool List::empty() {
23     return (size() == 0);
24 }
25
26 void List::clear() {
27     Node *p = head->next;
28     while (p != tail) {
29         Node *t = p->next;
30         delete p;
31         p = t;
32         head->next = t;
33     }
34 }
```

simple\_int\_list.cpp

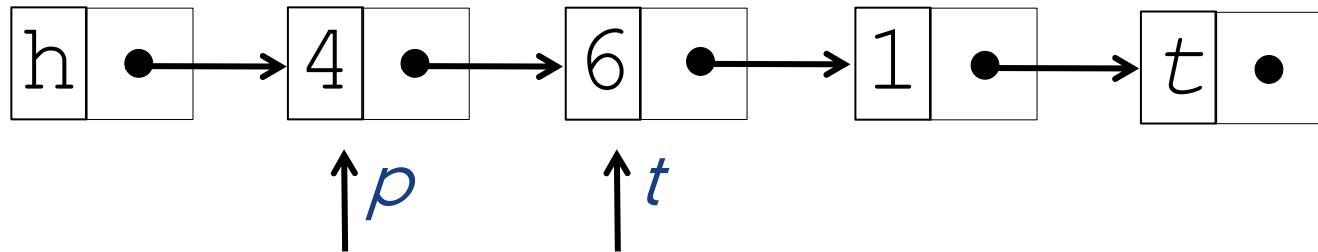
# LINKED LISTS

---

The constructor now builds the **two** marker nodes and initializes them.

The destructor deletes the **two nodes**.

Let us take a brief look at `clear()`.



```
1 #include <cassert>
2 #include "simple_int_list.h"
3
4 List::List() {
5     theSize = 0;
6     head = new Node;
7     tail = new Node;
8     head->next = tail;
9     tail->next = nullptr;
10 }
11
12 List::~List() {
13     clear();
14     delete head;
15     delete tail;
16 }
17
18 int List::size() {
19     return theSize;
20 }
21
22 bool List::empty() {
23     return (size() == 0);
24 }
25
26 void List::clear() {
27     Node *p = head->next;
28     while (p != tail) {
29         Node *t = p->next;
30         delete p;
31         p = t;
32         head->next = t;
33     }
34 }
```

simple\_int\_list.cpp

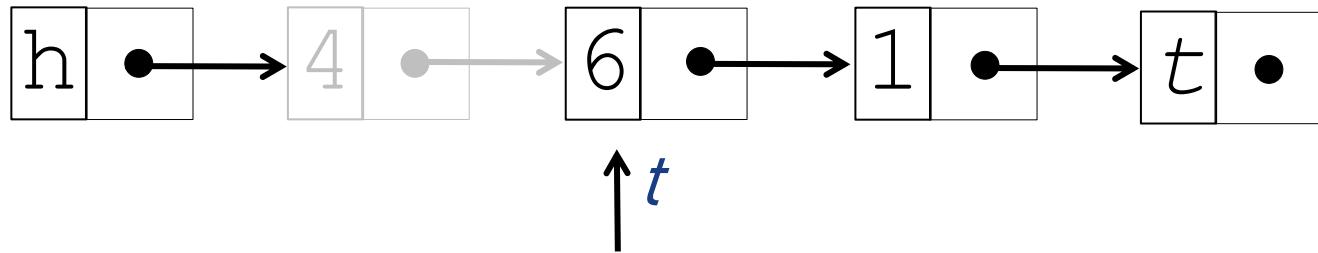
# LINKED LISTS

---

The constructor now builds the **two** marker nodes and initializes them.

The destructor deletes the **two nodes**.

Let us take a brief look at `clear()`.



```
1 #include <cassert>
2 #include "simple_int_list.h"
3
4 List::List() {
5     theSize = 0;
6     head = new Node;
7     tail = new Node;
8     head->next = tail;
9     tail->next = nullptr;
10 }
11
12 List::~List() {
13     clear();
14     delete head;
15     delete tail;
16 }
17
18 int List::size() {
19     return theSize;
20 }
21
22 bool List::empty() {
23     return (size() == 0);
24 }
25
26 void List::clear() {
27     Node *p = head->next;
28     while (p != tail) {
29         Node *t = p->next;
30         delete p;
31         p = t;
32         head->next = t;
33     }
34 }
```

simple\_int\_list.cpp

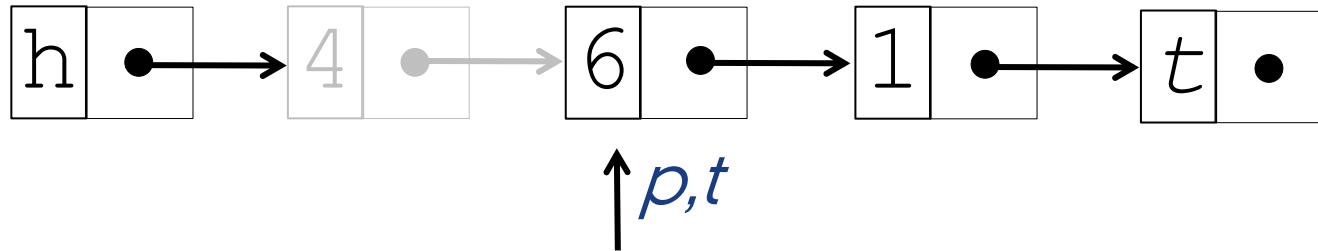
# LINKED LISTS

---

The constructor now builds the **two** marker nodes and initializes them.

The destructor deletes the **two nodes**.

Let us take a brief look at `clear()`.



```
1 #include <cassert>
2 #include "simple_int_list.h"
3
4 List::List() {
5     theSize = 0;
6     head = new Node;
7     tail = new Node;
8     head->next = tail;
9     tail->next = nullptr;
10 }
11
12 List::~List() {
13     clear();
14     delete head;
15     delete tail;
16 }
17
18 int List::size() {
19     return theSize;
20 }
21
22 bool List::empty() {
23     return (size() == 0);
24 }
25
26 void List::clear() {
27     Node *p = head->next;
28     while (p != tail) {
29         Node *t = p->next;
30         delete p;
31         p = t;
32         head->next = t;
33     }
34 }
```

simple\_int\_list.cpp

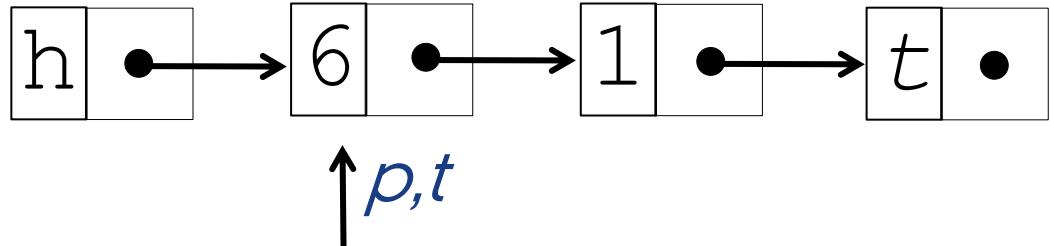
# LINKED LISTS

---

The constructor now builds the **two** marker nodes and initializes them.

The destructor deletes the **two nodes**.

Let us take a brief look at `clear()`.



```
1 #include <cassert>
2 #include "simple_int_list.h"
3
4 List::List() {
5     theSize = 0;
6     head = new Node;
7     tail = new Node;
8     head->next = tail;
9     tail->next = nullptr;
10 }
11
12 List::~List() {
13     clear();
14     delete head;
15     delete tail;
16 }
17
18 int List::size() {
19     return theSize;
20 }
21
22 bool List::empty() {
23     return (size() == 0);
24 }
25
26 void List::clear() {
27     Node *p = head->next;
28     while (p != tail) {
29         Node *t = p->next;
30         delete p;
31         p = t;
32         head->next = t;
33     }
34 }
```

simple\_int\_list.cpp

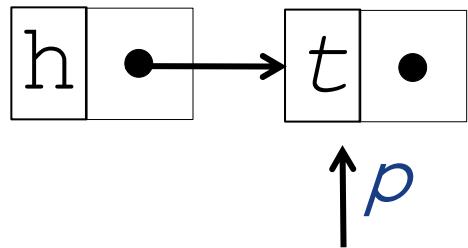
# LINKED LISTS

---

The constructor now builds the **two** marker nodes and initializes them.

The destructor deletes the **two nodes**.

Let us take a brief look at `clear()`.



In the stop condition,  
the list is **empty**.

```
1 #include <cassert>
2 #include "simple_int_list.h"
3
4 List::List() {
5     theSize = 0;
6     head = new Node;
7     tail = new Node;
8     head->next = tail;
9     tail->next = nullptr;
10 }
11
12 List::~List() {
13     clear();
14     delete head;
15     delete tail;
16 }
17
18 int List::size() {
19     return theSize;
20 }
21
22 bool List::empty() {
23     return (size() == 0);
24 }
25
26 void List::clear() {
27     Node *p = head->next;
28     while (p != tail) {
29         Node *t = p->next;
30         delete p;
31         p = t;
32         head->next = t;
33     }
34 }
```

simple\_int\_list.cpp

# REFLECTION

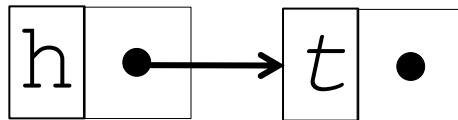
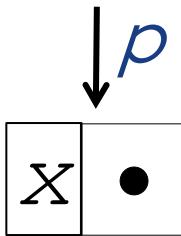
---

Is there a bug in the clear operation shown in the previous slide? If so, what is the bug and how can we fix it?

# LINKED LISTS

---

Adding to the **beginning** og the list  
Is easy and takes a **few** operations.



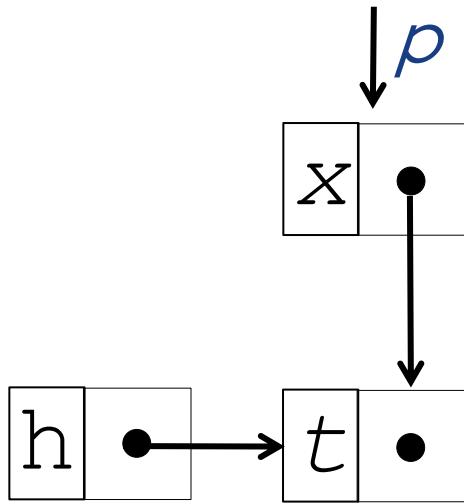
```
35 void List::push_front(int x) {  
36     Node *p = new Node;  
37     p->value = x;  
38     p->next = head->next;  
39     head->next = p;  
40     theSize++;  
41 }  
42  
43 void List::insert(int x, int pos) {  
44     assert(pos >= 0 && pos < theSize);  
45     Node *p = head->next;  
46     while (pos > 1) {  
47         p = p->next;  
48         pos--;  
49     }  
50     Node *q = new Node;  
51     q->value = x;  
52     q->next = p->next;  
53     p->next = q;  
54     theSize++;  
55 }  
56  
57 int List::find_kth(int pos) {  
58     assert(pos >= 0 && pos < theSize);  
59     Node *p = head->next;  
60     while (pos > 0) {  
61         p = p->next;  
62         pos--;  
63     }  
64     return p->value;  
65 }
```

simple\_int\_list.cpp

# LINKED LISTS

---

Adding to the **beginning** og the list  
Is easy and takes a **few** operations.



```
35 void List::push_front(int x) {  
36     Node *p = new Node;  
37     p->value = x;  
38     p->next = head->next;  
39     head->next = p;  
40     theSize++;  
41 }  
42  
43 void List::insert(int x, int pos) {  
44     assert(pos >= 0 && pos < theSize);  
45     Node *p = head->next;  
46     while (pos > 1) {  
47         p = p->next;  
48         pos--;  
49     }  
50     Node *q = new Node;  
51     q->value = x;  
52     q->next = p->next;  
53     p->next = q;  
54     theSize++;  
55 }  
56  
57 int List::find_kth(int pos) {  
58     assert(pos >= 0 && pos < theSize);  
59     Node *p = head->next;  
60     while (pos > 0) {  
61         p = p->next;  
62         pos--;  
63     }  
64     return p->value;  
65 }
```

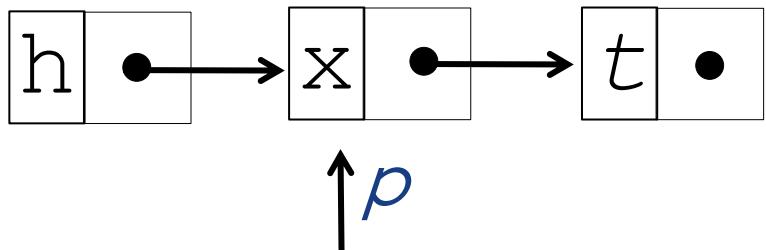
simple\_int\_list.cpp

# LINKED LISTS

---

Adding to the **beginning** og the list  
Is easy and takes a **few** operations.

What is the **time complexity**?



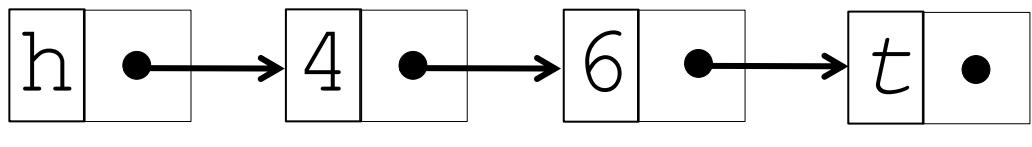
```
35 void List::push_front(int x) {  
36     Node *p = new Node;  
37     p->value = x;  
38     p->next = head->next;  
39     head->next = p;  
40     theSize++;  
41 }  
42  
43 void List::insert(int x, int pos) {  
44     assert(pos >= 0 && pos < theSize);  
45     Node *p = head->next;  
46     while (pos > 1) {  
47         p = p->next;  
48         pos--;  
49     }  
50     Node *q = new Node;  
51     q->value = x;  
52     q->next = p->next;  
53     p->next = q;  
54     theSize++;  
55 }  
56  
57 int List::find_kth(int pos) {  
58     assert(pos >= 0 && pos < theSize);  
59     Node *p = head->next;  
60     while (pos > 0) {  
61         p = p->next;  
62         pos--;  
63     }  
64     return p->value;  
65 }
```

simple\_int\_list.cpp

# LINKED LISTS

---

Inserting at a **certain position** requires to find the **predecessor** first. Let us take a look at `insert(x, 2)`.



$\uparrow p, pos = 2$

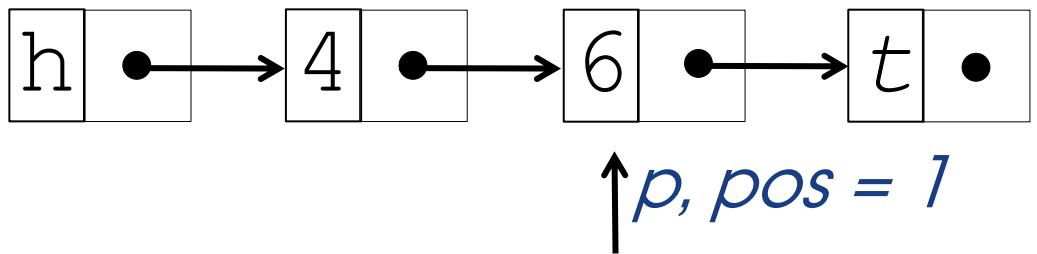
```
35 void List::push_front(int x) {  
36     Node *p = new Node;  
37     p->value = x;  
38     p->next = head->next;  
39     head->next = p;  
40     theSize++;  
41 }  
42  
43 void List::insert(int x, int pos) {  
44     assert(pos >= 0 && pos < theSize);  
45     Node *p = head->next;  
46     while (pos > 1) {  
47         p = p->next;  
48         pos--;  
49     }  
50     Node *q = new Node;  
51     q->value = x;  
52     q->next = p->next;  
53     p->next = q;  
54     theSize++;  
55 }  
56  
57 int List::find_kth(int pos) {  
58     assert(pos >= 0 && pos < theSize);  
59     Node *p = head->next;  
60     while (pos > 0) {  
61         p = p->next;  
62         pos--;  
63     }  
64     return p->value;  
65 }
```

simple\_int\_list.cpp

# LINKED LISTS

---

Inserting at a **certain position** requires to find the **predecessor** first. Let us take a look at `insert(x, 2)`.



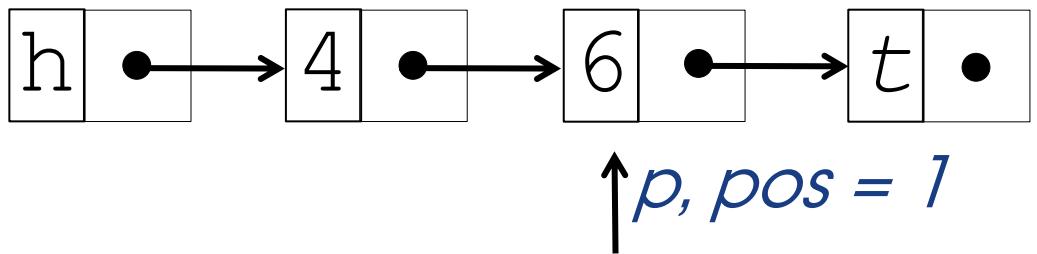
```
35 void List::push_front(int x) {  
36     Node *p = new Node;  
37     p->value = x;  
38     p->next = head->next;  
39     head->next = p;  
40     theSize++;  
41 }  
42  
43 void List::insert(int x, int pos) {  
44     assert(pos >= 0 && pos < theSize);  
45     Node *p = head->next;  
46     while (pos > 1) {  
47         p = p->next;  
48         pos--;  
49     }  
50     Node *q = new Node;  
51     q->value = x;  
52     q->next = p->next;  
53     p->next = q;  
54     theSize++;  
55 }  
56  
57 int List::find_kth(int pos) {  
58     assert(pos >= 0 && pos < theSize);  
59     Node *p = head->next;  
60     while (pos > 0) {  
61         p = p->next;  
62         pos--;  
63     }  
64     return p->value;  
65 }
```

simple\_int\_list.cpp

# LINKED LISTS

---

Inserting at a **certain position** requires to find the **predecessor** first. Let us take a look at `insert(x, 2)`.



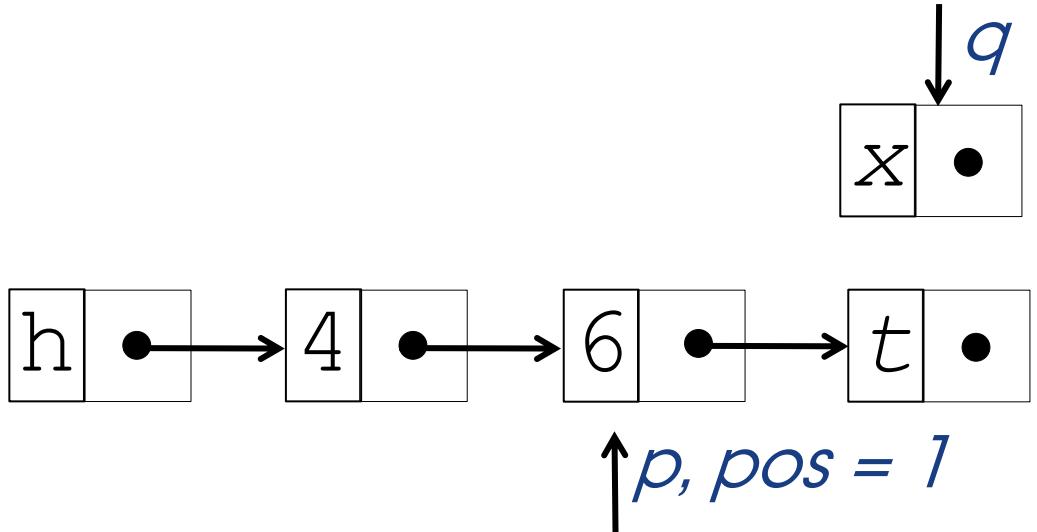
```
35 void List::push_front(int x) {  
36     Node *p = new Node;  
37     p->value = x;  
38     p->next = head->next;  
39     head->next = p;  
40     theSize++;  
41 }  
42  
43 void List::insert(int x, int pos) {  
44     assert(pos >= 0 && pos < theSize);  
45     Node *p = head->next;  
46     while (pos > 1) {  
47         p = p->next;  
48         pos--;  
49     }  
50     Node *q = new Node;  
51     q->value = x;  
52     q->next = p->next;  
53     p->next = q;  
54     theSize++;  
55 }  
56  
57 int List::find_kth(int pos) {  
58     assert(pos >= 0 && pos < theSize);  
59     Node *p = head->next;  
60     while (pos > 0) {  
61         p = p->next;  
62         pos--;  
63     }  
64     return p->value;  
65 }
```

simple\_int\_list.cpp

# LINKED LISTS

---

Inserting at a **certain position** requires to find the **predecessor** first. Let us take a look at `insert(x, 2)`.



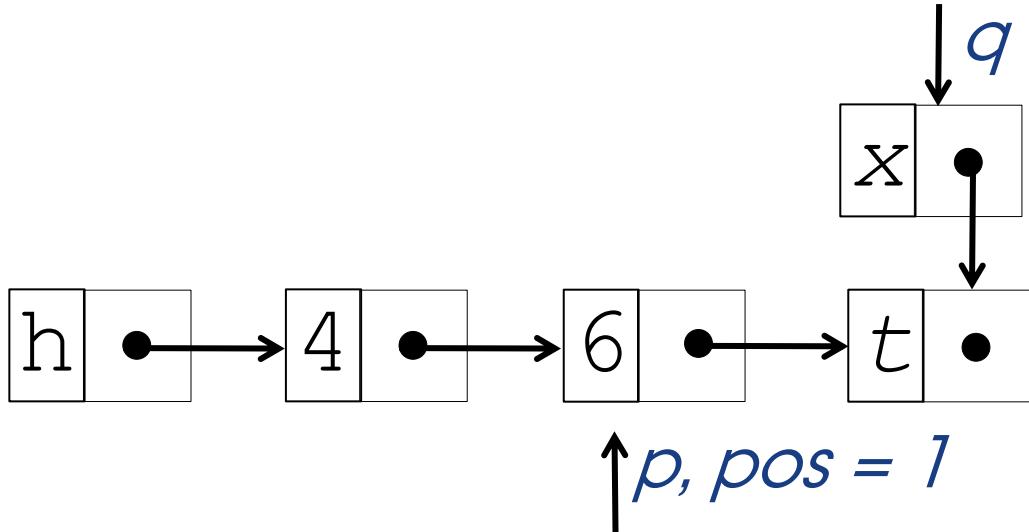
```
35 void List::push_front(int x) {  
36     Node *p = new Node;  
37     p->value = x;  
38     p->next = head->next;  
39     head->next = p;  
40     theSize++;  
41 }  
42  
43 void List::insert(int x, int pos) {  
44     assert(pos >= 0 && pos < theSize);  
45     Node *p = head->next;  
46     while (pos > 1) {  
47         p = p->next;  
48         pos--;  
49     }  
50     Node *q = new Node;  
51     q->value = x;  
52     q->next = p->next;  
53     p->next = q;  
54     theSize++;  
55 }  
56  
57 int List::find_kth(int pos) {  
58     assert(pos >= 0 && pos < theSize);  
59     Node *p = head->next;  
60     while (pos > 0) {  
61         p = p->next;  
62         pos--;  
63     }  
64     return p->value;  
65 }
```

simple\_int\_list.cpp

# LINKED LISTS

---

Inserting at a **certain position** requires to find the **predecessor** first. Let us take a look at `insert(x, 2)`.



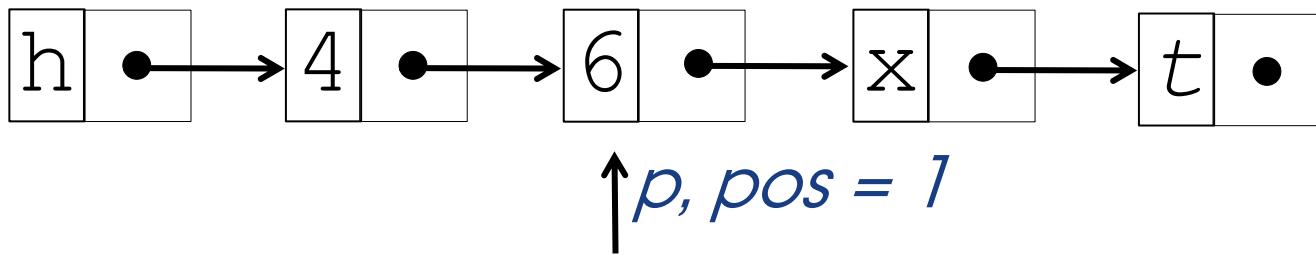
```
35 void List::push_front(int x) {  
36     Node *p = new Node;  
37     p->value = x;  
38     p->next = head->next;  
39     head->next = p;  
40     theSize++;  
41 }  
42  
43 void List::insert(int x, int pos) {  
44     assert(pos >= 0 && pos < theSize);  
45     Node *p = head->next;  
46     while (pos > 1) {  
47         p = p->next;  
48         pos--;  
49     }  
50     Node *q = new Node;  
51     q->value = x;  
52     q->next = p->next;  
53     p->next = q;  
54     theSize++;  
55 }  
56  
57 int List::find_kth(int pos) {  
58     assert(pos >= 0 && pos < theSize);  
59     Node *p = head->next;  
60     while (pos > 0) {  
61         p = p->next;  
62         pos--;  
63     }  
64     return p->value;  
65 }
```

simple\_int\_list.cpp

# LINKED LISTS

---

Inserting at a **certain position** requires to find the **predecessor** first. Let us take a look at `insert(x, 2)`.



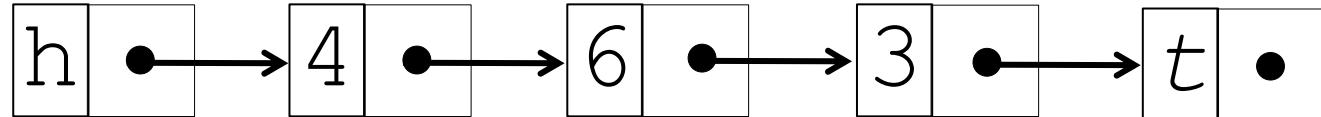
```
35 void List::push_front(int x) {  
36     Node *p = new Node;  
37     p->value = x;  
38     p->next = head->next;  
39     head->next = p;  
40     theSize++;  
41 }  
42  
43 void List::insert(int x, int pos) {  
44     assert(pos >= 0 && pos < theSize);  
45     Node *p = head->next;  
46     while (pos > 1) {  
47         p = p->next;  
48         pos--;  
49     }  
50     Node *q = new Node;  
51     q->value = x;  
52     q->next = p->next;  
53     p->next = q; // Line 53 is highlighted in blue  
54     theSize++;  
55 }  
56  
57 int List::find_kth(int pos) {  
58     assert(pos >= 0 && pos < theSize);  
59     Node *p = head->next;  
60     while (pos > 0) {  
61         p = p->next;  
62         pos--;  
63     }  
64     return p->value;  
65 }
```

simple\_int\_list.cpp

# REFLECTION

---

Run with pen and paper  
find\_kth(2) on the following list:



Finding the  $k$ th element is similar to insert(2), but notice the different **condition** (why?)

```
35 void List::push_front(int x) {  
36     Node *p = new Node;  
37     p->value = x;  
38     p->next = head->next;  
39     head->next = p;  
40     theSize++;  
41 }  
42  
43 void List::insert(int x, int pos) {  
44     assert(pos >= 0 && pos < theSize);  
45     Node *p = head->next;  
46     while (pos > 1) {  
47         p = p->next;  
48         pos--;  
49     }  
50     Node *q = new Node;  
51     q->value = x;  
52     q->next = p->next;  
53     p->next = q;  
54     theSize++;  
55 }  
56  
57 int List::find_kth(int pos) {  
58     assert(pos >= 0 && pos < theSize);  
59     Node *p = head->next;  
60     while (pos > 0) {  
61         p = p->next;  
62         pos--;  
63     }  
64     return p->value;  
65 }
```

simple\_int\_list.cpp

# REFLECTION

---

What is the worst case complexity of  
find\_kth?

```
35 void List::push_front(int x) {  
36     Node *p = new Node;  
37     p->value = x;  
38     p->next = head->next;  
39     head->next = p;  
40     theSize++;  
41 }  
42  
43 void List::insert(int x, int pos) {  
44     assert(pos >= 0 && pos < theSize);  
45     Node *p = head->next;  
46     while (pos > 1) {  
47         p = p->next;  
48         pos--;  
49     }  
50     Node *q = new Node;  
51     q->value = x;  
52     q->next = p->next;  
53     p->next = q;  
54     theSize++;  
55 }  
56  
57 int List::find_kth(int pos) {  
58     assert(pos >= 0 && pos < theSize);  
59     Node *p = head->next;  
60     while (pos > 0) {  
61         p = p->next;  
62         pos--;  
63     }  
64     return p->value;  
65 }
```

simple\_int\_list.cpp

# LINKED LISTS

---

Removing from the **front** also takes just a **few** operations, so it is  $O(1)$ .

Removing at a certain position **again** requires to find the predecessor.

```
67 int List::pop_front() {  
68     Node *p = head->next;  
69     int x = p->value;  
70     head->next = p->next;  
71     theSize--;  
72     delete p;  
73     return x;  
74 }  
75  
76 int List::remove(int pos) {  
77     assert(pos >= 0 && pos < theSize);  
78     Node *p = head->next;  
79     while (pos > 1) {  
80         p = p->next;  
81         pos--;  
82     }  
83     Node *q = p->next;  
84     int x = q->value;  
85     p->next = q->next;  
86     theSize--;  
87     delete q;  
88     return x;  
89 }
```

# REFLECTION

---

Notice that we are **updating** the size as we **insert/remove** elements.

What is the **time complexity** of `size()` (shown below)?

```
17  
18 int List::size() {  
19     return theSize;  
20 }
```

Could we implement the linked list without using `theSize`? If so, what would be the time complexity of `size()`?

```
67 int List::pop_front() {  
68     Node *p = head->next;  
69     int x = p->value;  
70     head->next = p->next;  
71     theSize--;  
72     delete p;  
73     return x;  
74 }  
75  
76 int List::remove(int pos) {  
77     assert(pos >= 0 && pos < theSize);  
78     Node *p = head->next;  
79     while (pos > 1) {  
80         p = p->next;  
81         pos--;  
82     }  
83     Node *q = p->next;  
84     int x = q->value;  
85     p->next = q->next;  
86     theSize--;  
87     delete q;  
88     return x;  
89 }
```

simple\_int\_list.cpp

# REFLECTION

---

The testing code **instantiates** a list **dynamically**.

Again, notice how **abstraction** removes explicit **pointers**, **types** and **implementation details**.

What is the **output** of the program?

```
1 using namespace std;
2
3 #include <iostream>
4
5 #include "simple_int_list.h"
6
7 int main(int argc, char *argv[]) {
8     List *list = new List();
9
10    list->push_front(10);
11    list->push_front(5);
12    list->push_front(3);
13    list->push_front(7);
14
15    cout << "List size      : " << list->size() << endl;
16    cout << "First element  : " << list->find_kth(0) << endl;
17    cout << "Second element : " << list->find_kth(1) << endl;
18    cout << "Removed element: " << list->remove(1) << endl;
19    cout << "Second element : " << list->find_kth(1) << endl;
20    list->insert(20, 1);
21    cout << "Added element  : " << 20 << endl;
22    cout << "Second element : " << list->find_kth(1) << endl;
23
24    while (list->empty() == false) {
25        cout << "Next element:   " << list->pop_front() << endl;
26    }
27    cout << "List is empty?  " << list->empty() << endl;
28
29    delete list;
30 }
```

test\_simple\_int\_list.cpp

# LINKED LISTS

---

Our list still has a big **flexibility problem**. It can only handle integers. We would have to implement a **different list** for each object.

We need a higher-level abstraction to build a **container for any kind of object** we want to store.

C++ provides a **meta-programming** facility called *templates* that fits the job. Let us adapt our linked list to use this tool.

# LINKED LISTS

---

There are several **new things** here!

A *template* keyword defines a generic type.

Templates do not have an **implementation file**, so we include simple functions in the header.

For **clarity**, we will still move **complex implementations** to another file.

We can now specify the interface **in terms of a generic type**. How does it impact the **implementation**?

```
1 #ifndef _LIST_H_
2 #define _LIST_H_
3
4 template <typename Object>
5 class List {
6     private:
7         struct Node {
8             Object data;
9             Node *next;
10        };
11     int theSize;
12     Node *head;
13     Node *tail;
14
15 public:
16     List() {
17         theSize = 0;
18         head = new Node; tail = new Node;
19         head->next = tail;
20         tail->next = nullptr;
21     }
22
23     ~List() { clear(); delete head; delete tail; }
24
25     int size() { return theSize; }
26     bool empty() { return (size() == 0); }
27
28     void clear(),
29     void push_front(const Object x);
30     Object pop_front();
31     Object find_kth(int pos);
32 };
33
34 #include "simple_list.tpp"
35
36 #endif
```

simple\_list.h

# LINKED LISTS

The impact is quite **small**.  
The **compiler** deals with the abstraction.

Notice the *template* keyword in all functions.

```
1 #include <cassert>
2
3 template <typename Object>
4 void List<Object>::clear() {
5     Node *p = head->next;
6     while (p != tail) {
7         Node *t = p->next;
8         delete p;
9         p = t;
10        head->next = t;
11    }
12 }
13
14 template <typename Object>
15 void List<Object>::push_front(const Object x) {
16     Node *p = new Node;
17     p->data = x;
18     p->next = head->next;
19     head->next = p;
20     theSize++;
21 }
23 template <typename Object>
24 Object List<Object>::pop_front() {
25     Node *p = head->next;
26     Object x = p->data;
27     head->next = p->next;
28     theSize--;
29     delete p;
30     return x;
31 }
32
33 template <typename Object>
34 Object List<Object>::find_kth(int pos) {
35     assert(pos >= 0 && pos < theSize);
36     Node *p = head->next;
37     while (pos > 0) {
38         p = p->next;
39         pos--;
40     }
41     return p->data;
42 }
```

simple\_list.tpp

# LINKED LISTS

---

Finally, look at the testing code:

- We can specify lists of different **types** (integers and floats)
- We can call the **same operations**, no matter the **underlying type**.
- Even the **printing operation** is equivalent!

That is the **power of abstraction**.

```
1 using namespace std;
2 #include <cmath>
3 #include <iostream>
4 #include "simple_list.h"
5
6 int main(int argc, char *argv[]) {
7     List<int> *list = new List<int>();
8     list->push_front(10);
9     list->push_front(5);
10    list->push_front(3);
11    list->push_front(7);
12
13    cout << "List size:      " << list->size() << endl;
14    cout << "First element: " << list->find_kth(0) << endl;
15    cout << "Second element: " << list->find_kth(1) << endl;
16    while (list->empty() == false) {
17        cout << "Next element: " << list->pop_front() << endl;
18    }
19    cout << "List is empty? " << list->empty() << endl;
20
21    delete list;
22
23    List<float> *listf = new List<float>();
24    listf->push_front(1.1);
25    listf->push_front(-0.5);
26    listf->push_front(M_PI);
27    listf->push_front(7.0);
28
29    cout << "List size:      " << listf->size() << endl;
30    cout << "First element: " << listf->find_kth(0) << endl;
31    cout << "Second element: " << listf->find_kth(1) << endl;
32    while (listf->empty() == false) {
33        cout << "Next element: " << listf->pop_front() << endl;
34    }
35    cout << "List is empty? " << listf->empty() << endl;
36
37    delete listf;
38 }
```

test\_simple\_list.cpp

# LINKED LISTS

Now we look at the inheritance method for defining the list ADT and its implementation:

C simple\_list.h

```
1 #pragma once
2
3 template <typename Object>
4 class List {
5
6     public:
7         virtual int size() = 0;
8         virtual bool empty() = 0;
9         virtual void clear() = 0;
10        virtual void push_front(const Object x) = 0;
11        virtual void push_back(const Object x) = 0;
12        virtual Object pop_front() = 0;
13        virtual Object pop_back() = 0;
14        virtual Object find_kth(int pos) = 0;
15    };
16
```

These two new operations have been introduced for the sake of motivating the next data structure.

C simple\_linked\_list.h

```
1 #pragma once
2
3 #include <cassert>
4 #include "simple_list.h"
5
6 template <typename Object>
7 class LinkedList:
8     public List<Object>
9 {
10     private:
11         struct Node {
12             Object data;
13             Node *next;
14         };
15         int theSize;
16         Node *head;
17         Node *tail;
18
19     public:
20
21         LinkedList() {
22             theSize = 0;
23             head = new Node; tail = new Node;
24             head->next = tail;
25             tail->next = nullptr;
26         }
27
28         ~LinkedList() {
29             clear();
30             delete head;
31             delete tail;
32         }
33 }
```

C simple\_linked\_list.h

```
41     }
42
43     int size() { return theSize; }
44     bool empty() { return (size() == 0); }
45
46     void clear() {
47         Node *p = head->next;
48         while (p != tail) {
49             Node *t = p->next;
50             delete p;
51             p = t;
52             head->next = t;
53         }
54     }
55
56     void push_front(const Object x) {
57         Node *p = new Node;
58         p->data = x;
59         p->next = head->next;
60         head->next = p;
61         theSize++;
62     }
63
64     void push_back(const Object x) {
65         // Locate last node:
66         Node *last = head;
67         while (last->next != tail) {
68             last = last->next;
69         }
70         // Add new node and set its next to tail.
71         Node *p = new Node;
72         p->data = x;
73         p->next = tail;
74         // Set last node to new node.
75         last->next = p;
76         theSize++;
77     }
78 }
```

C simple\_linked\_list.h

```
79     Object pop_front(){
80         Node *p = head->next;
81         Object x = p->data;
82         head->next = p->next;
83         theSize--;
84         delete p;
85         return x;
86     }
87
88     Object pop_back(){
89         assert(theSize > 0);
90         if (theSize == 1){
91             return pop_front();
92         }
93         assert (theSize >= 2);
94         Node *second_to_last = head;
95         while (second_to_last->next->next != tail) {
96             second_to_last = second_to_last->next;
97         }
98         Object x = second_to_last->next->data;
99         second_to_last->next = tail;
100        theSize--;
101        delete second_to_last->next;
102        return x;
103    }
104
105    Object find_kth(int pos){
106        assert(pos >= 0 && pos < theSize);
107        Node *p = head->next;
108        while (pos > 0) {
109            p = p->next;
110            pos--;
111        }
112        assert(pos >= 0 && p != NULL);
113        return p->data;
114    }
115 }
```



# REFLECTION

---

- What if we stored the second to last node in a private field, just like we do with tail? Would this improve the efficiency of the *pop\_back* operation?

# LINKED LISTS

---

What about **time complexity** of our list implementation?

- Constructor and destructor run in  $O(1)$ .
- Measuring the size runs in  $O(1)$ .
- Inserting and removing **at the front** take  $O(1)$ .
- Random access **in a fixed position** takes  $O(N)$ .
- Inserting and removing **in a fixed position** take  $O(N)$ .

# REFLECTION

---

- What is the time complexity of removing/inserting at the back of the list?

# AGENDA

---

Introduction to program analysis  
Growth of functions  
Notions of algorithm complexity  
Benchmarking and efficiency  
Data structures, Abstract Data Types (ADT), and Pre/Post Conditions  
Linked Lists and Operations  
**Doubly Linked Lists and Operations**

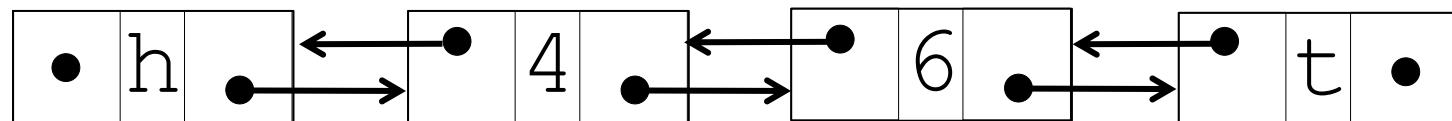
# DOUBLY LINKED LISTS

---

## Definition:

A *doubly linked list* is a linked list where each node has references to the previous and the next node.

We can easily **modify** our **template** linked list implementation to add a pointer to the **predecessor** node.



Most operations will **essentially** remain the same, but we will take a look at the updated interface and implementation.

# DOUBLY LINKED LISTS

---

A few **modifications** are needed:

- An additional **pointer**.
- Additional **functions** to add and remove at the end.

How do these modifications **affect** the **implementation**?

Basically we need to **update more pointers** inside the operations.

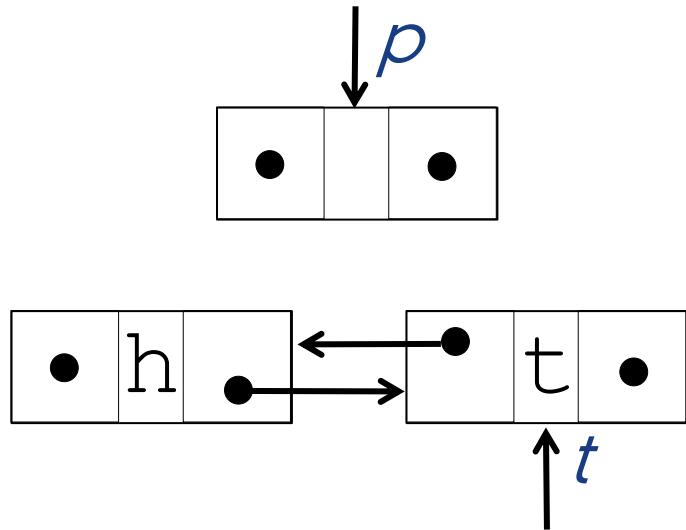
```
4 template <typename Object>
5 class List {
6     private:
7         struct Node {
8             Object data;
9             Node *next;
10            Node *prev;
11        };
12        int theSize;
13        Node *head;
14        Node *tail;
15
16    public:
17        List() {
18            theSize = 0;
19            head = new Node; tail = new Node;
20            head->next = tail;
21            tail->prev = head;
22            head->prev = tail->next = nullptr;
23        }
```

```
30    void clear();
31    void push_front(const Object x);
32    void push_back(const Object x);
33    Object pop_front();
34    Object pop_back();
35    Object find_kth(int pos);
36 }
```

# DOUBLY LINKED LISTS

---

Let us look how **insertion at the front** works now for element  $x$ , starting from an **empty list**.



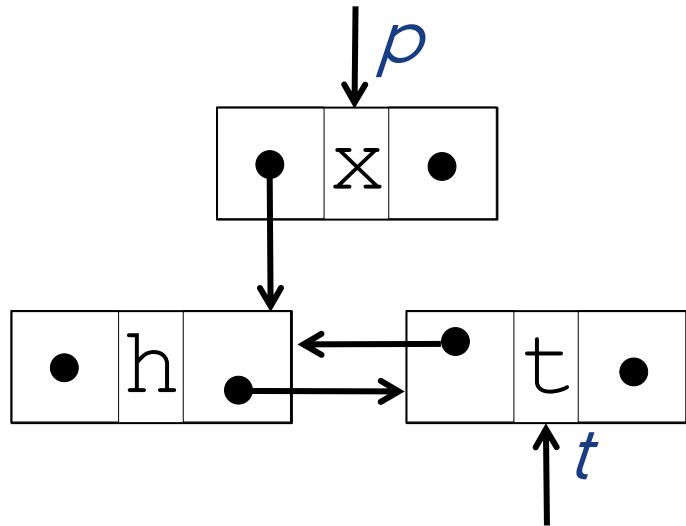
```
15 template <typename Object>
16 void List<Object>::push_front(const Object x) {
17     Node *p = new Node;
18     Node *t = head->next;
19     p->data = x;
20     p->prev = head;
21     p->next = t;
22     t->prev = p;
23     head->next = p;
24     theSize++;
25 }
26
27 template <typename Object>
28 void List<Object>::push_back(const Object x) {
29     Node *p = new Node;
30     Node *t = tail->prev;
31     p->data = x;
32     p->prev = t;
33     p->next = tail;
34     t->next = p;
35     tail->prev = p;
36     theSize++;
37 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

---

Let us look how **insertion at the front** works now for element  $x$ , starting from an **empty list**.



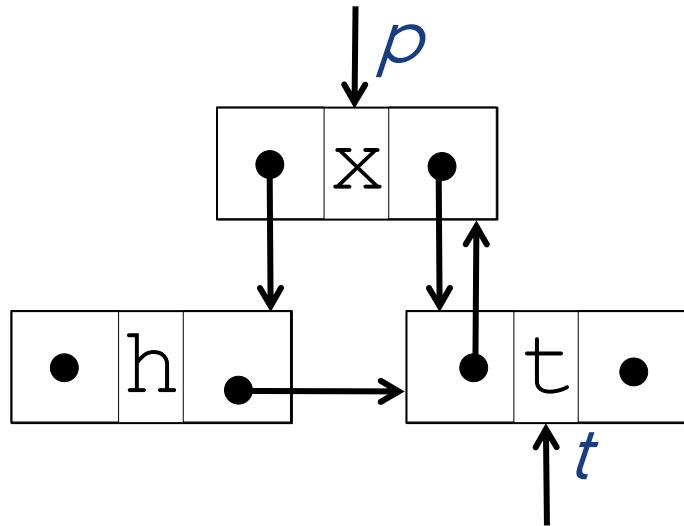
```
15 template <typename Object>
16 void List<Object>::push_front(const Object x) {
17     Node *p = new Node;
18     Node *t = head->next;
19     p->data = x;
20     p->prev = head;
21     p->next = t;
22     t->prev = p;
23     head->next = p;
24     theSize++;
25 }
26
27 template <typename Object>
28 void List<Object>::push_back(const Object x) {
29     Node *p = new Node;
30     Node *t = tail->prev;
31     p->data = x;
32     p->prev = t;
33     p->next = tail;
34     t->next = p;
35     tail->prev = p;
36     theSize++;
37 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

---

Let us look how **insertion at the front** works now for element  $x$ , starting from an **empty list**.



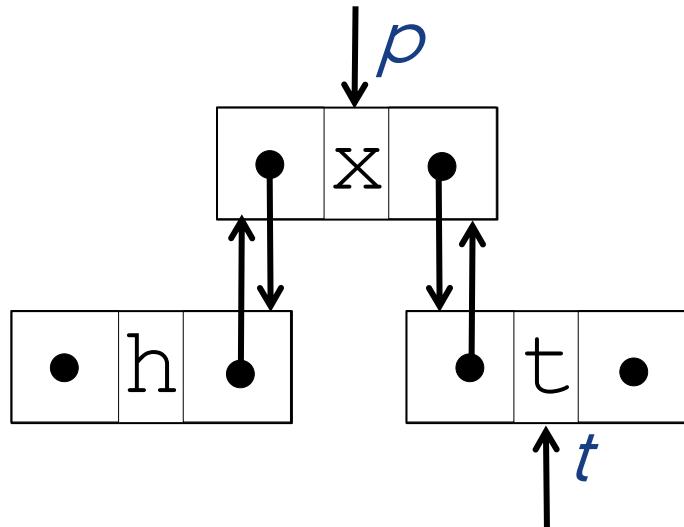
```
15 template <typename Object>
16 void List<Object>::push_front(const Object x) {
17     Node *p = new Node;
18     Node *t = head->next;
19     p->data = x;
20     p->prev = head;
21     p->next = t;
22     t->prev = p;
23     head->next = p;
24     theSize++;
25 }
26
27 template <typename Object>
28 void List<Object>::push_back(const Object x) {
29     Node *p = new Node;
30     Node *t = tail->prev;
31     p->data = x;
32     p->prev = t;
33     p->next = tail;
34     t->next = p;
35     tail->prev = p;
36     theSize++;
37 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

---

Let us look how **insertion at the front** works now for element  $x$ , starting from an **empty list**.



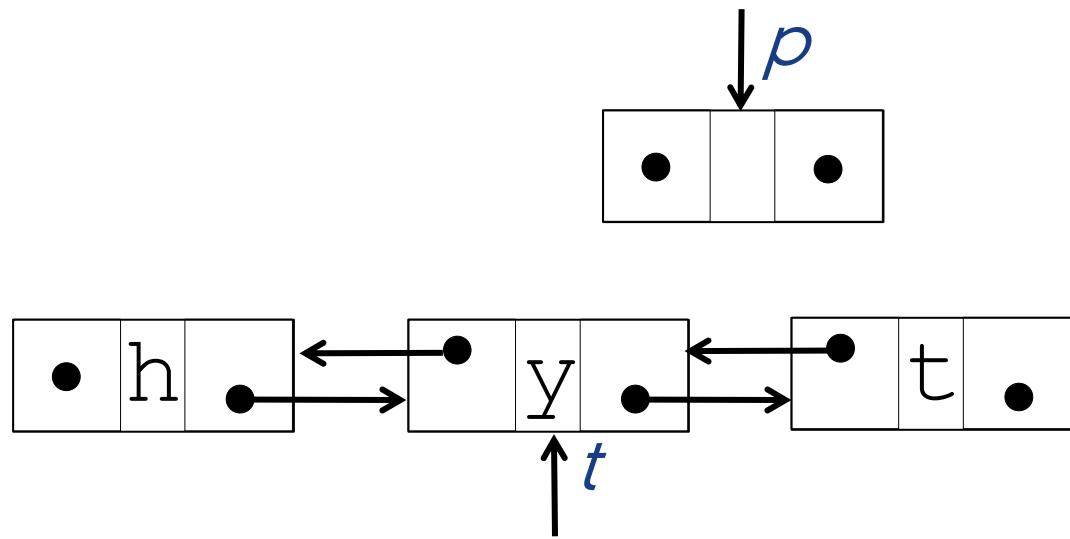
```
15 template <typename Object>
16 void List<Object>::push_front(const Object x) {
17     Node *p = new Node;
18     Node *t = head->next;
19     p->data = x;
20     p->prev = head;
21     p->next = t;
22     t->prev = p;
23     head->next = p;
24     theSize++;
25 }
26
27 template <typename Object>
28 void List<Object>::push_back(const Object x) {
29     Node *p = new Node;
30     Node *t = tail->prev;
31     p->data = x;
32     p->prev = t;
33     p->next = tail;
34     t->next = p;
35     tail->prev = p;
36     theSize++;
37 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

---

Let us look how **insertion at the end** works now for element  $x$ , assuming that the list has size 1.



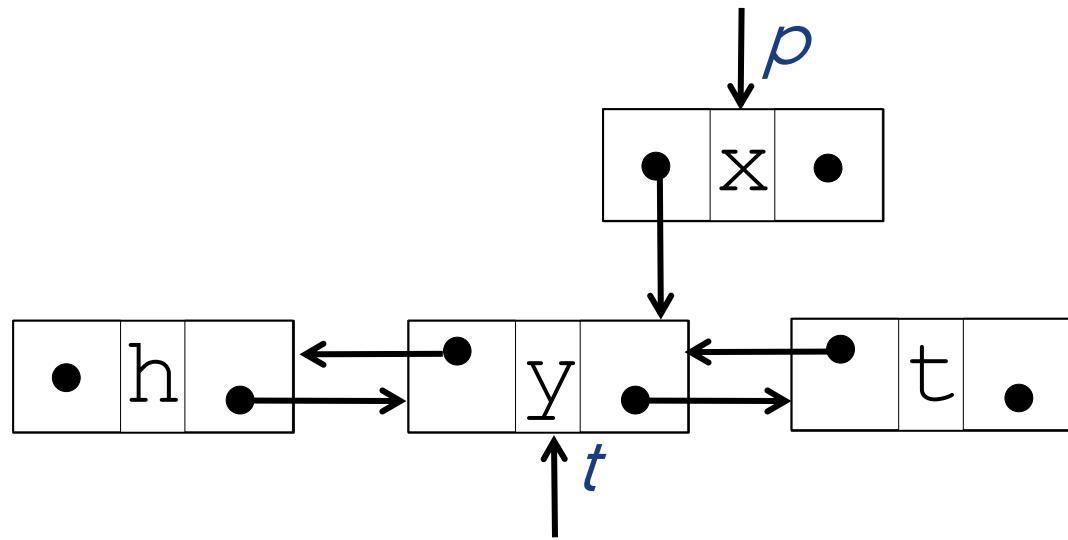
```
15 template <typename Object>
16 void List<Object>::push_front(const Object x) {
17     Node *p = new Node;
18     Node *t = head->next;
19     p->data = x;
20     p->prev = head;
21     p->next = t;
22     t->prev = p;
23     head->next = p;
24     theSize++;
25 }
26
27 template <typename Object>
28 void List<Object>::push_back(const Object x) {
29     Node *p = new Node;
30     Node *t = tail->prev;
31     p->data = x;
32     p->prev = t;
33     p->next = tail;
34     t->next = p;
35     tail->prev = p;
36     theSize++;
37 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

---

Let us look how **insertion at the end** works now for element  $x$ , assuming that the list has size 1.



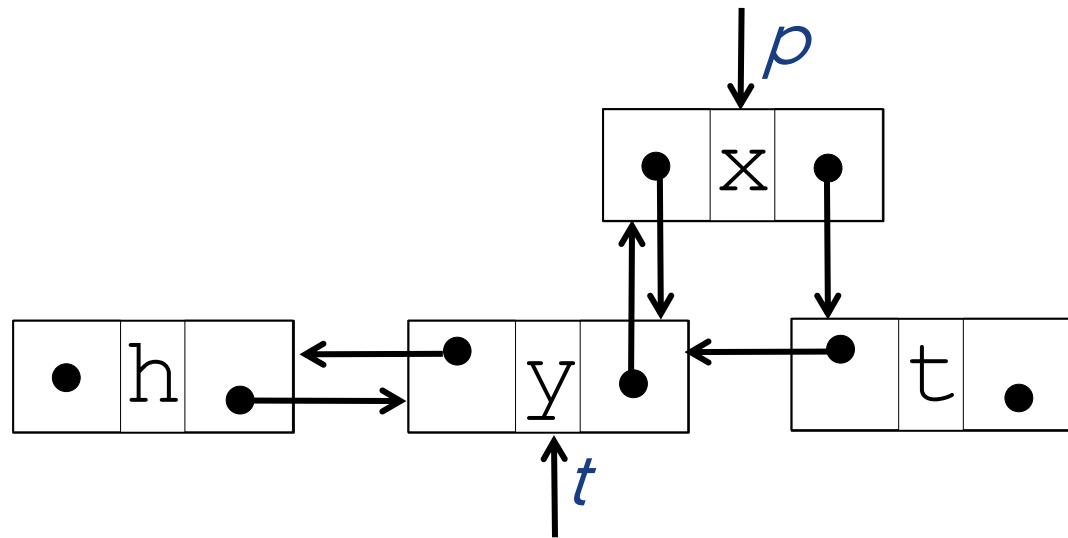
```
15 template <typename Object>
16 void List<Object>::push_front(const Object x) {
17     Node *p = new Node;
18     Node *t = head->next;
19     p->data = x;
20     p->prev = head;
21     p->next = t;
22     t->prev = p;
23     head->next = p;
24     theSize++;
25 }
26
27 template <typename Object>
28 void List<Object>::push_back(const Object x) {
29     Node *p = new Node;
30     Node *t = tail->prev;
31     p->data = x;
32     p->prev = t;
33     p->next = tail;
34     t->next = p;
35     tail->prev = p;
36     theSize++;
37 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

---

Let us look how **insertion at the end** works now for element  $x$ , assuming that the list has size 1.



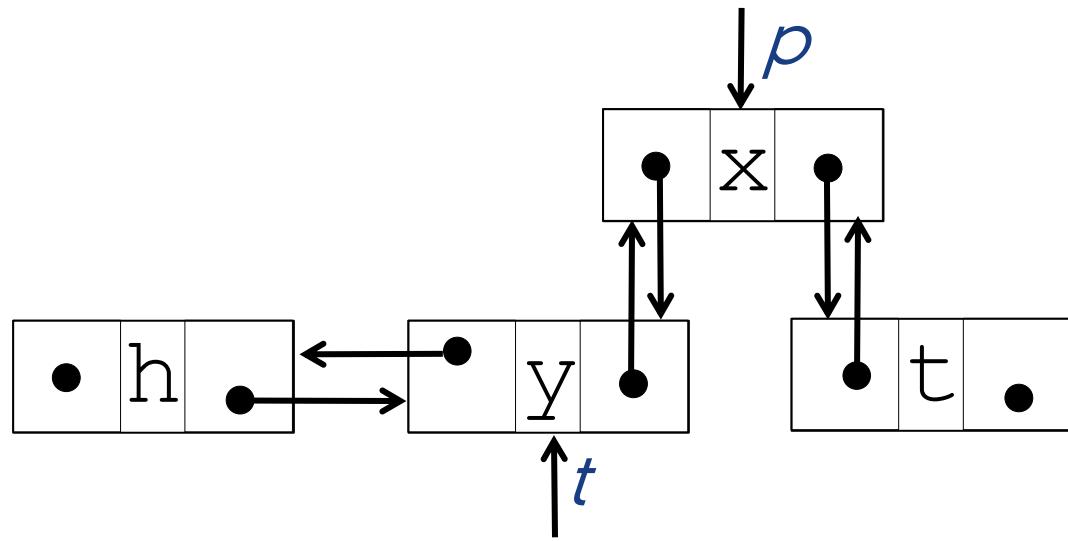
```
15 template <typename Object>
16 void List<Object>::push_front(const Object x) {
17     Node *p = new Node;
18     Node *t = head->next;
19     p->data = x;
20     p->prev = head;
21     p->next = t;
22     t->prev = p;
23     head->next = p;
24     theSize++;
25 }
26
27 template <typename Object>
28 void List<Object>::push_back(const Object x) {
29     Node *p = new Node;
30     Node *t = tail->prev;
31     p->data = x;
32     p->prev = t;
33     p->next = tail;
34     t->next = p;
35     tail->prev = p;
36     theSize++;
37 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

---

Let us look how **insertion at the end** works now for element  $x$ , assuming that the list has size 1.



```
15 template <typename Object>
16 void List<Object>::push_front(const Object x) {
17     Node *p = new Node;
18     Node *t = head->next;
19     p->data = x;
20     p->prev = head;
21     p->next = t;
22     t->prev = p;
23     head->next = p;
24     theSize++;
25 }
26
27 template <typename Object>
28 void List<Object>::push_back(const Object x) {
29     Node *p = new Node;
30     Node *t = tail->prev;
31     p->data = x;
32     p->prev = t;
33     p->next = tail;
34     t->next = p;
35     tail->prev = p;
36     theSize++;
37 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

---

Let us look how **insertion at the end** works now for element y.

What is the time complexity of **inserting at the end**?

Let us now take a quick look at the **modified code for removing** elements.

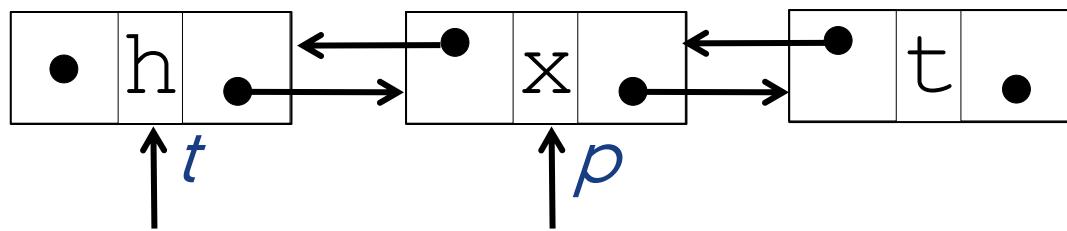
```
15 template <typename Object>
16 void List<Object>::push_front(const Object x) {
17     Node *p = new Node;
18     Node *t = head->next;
19     p->data = x;
20     p->prev = head;
21     p->next = t;
22     t->prev = p;
23     head->next = p;
24     theSize++;
25 }
26
27 template <typename Object>
28 void List<Object>::push_back(const Object x) {
29     Node *p = new Node;
30     Node *t = tail->prev;
31     p->data = x;
32     p->prev = t;
33     p->next = tail;
34     t->next = p;
35     tail->prev = p;
36     theSize++;
37 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

---

Now we can remove elements at the end/front in  $O(1)$  by **just adjusting the pointers!**



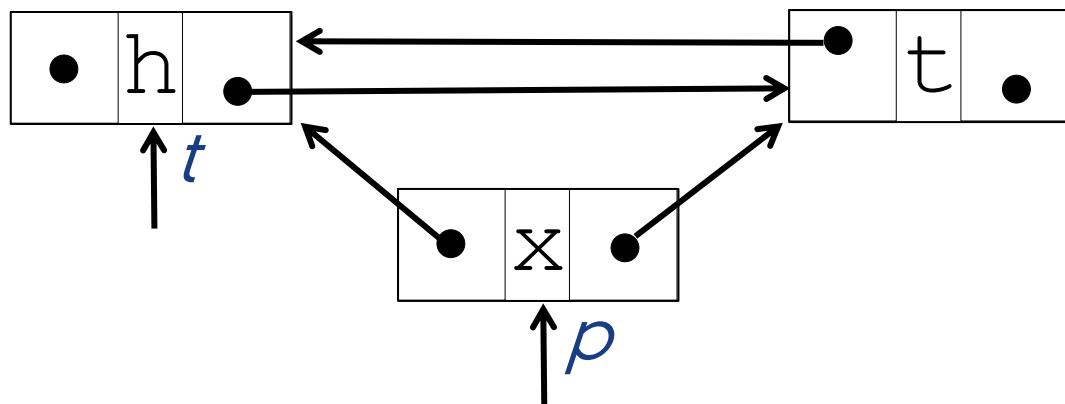
```
39 template <typename Object>
40 Object List<Object>::pop_front() {
41     Node *p = head->next;
42     Node *t = p->next;
43     Object x = p->data;
44     t->prev = head;
45     head->next = t;
46     theSize--;
47     delete p;
48     return x;
49 }
50
51 template <typename Object>
52 Object List<Object>::pop_back() {
53     Node *p = tail->prev;
54     Node *t = p->prev;
55     Object x = p->data;
56     t->next = tail;
57     tail->prev = t;
58     theSize--;
59     delete p;
60     return x;
61 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

---

Now we can remove elements at the end/front in  $O(1)$  by **just adjusting the pointers!**



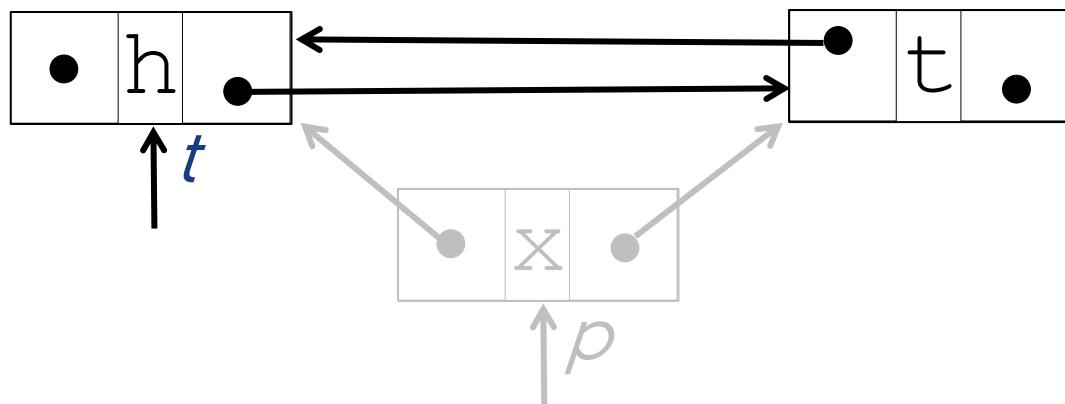
```
39 template <typename Object>
40 Object List<Object>::pop_front() {
41     Node *p = head->next;
42     Node *t = p->next;
43     Object x = p->data;
44     t->prev = head;
45     head->next = t;
46     theSize--;
47     delete p;
48     return x;
49 }
50
51 template <typename Object>
52 Object List<Object>::pop_back() {
53     Node *p = tail->prev;
54     Node *t = p->prev;
55     Object x = p->data;
56     t->next = tail;
57     tail->prev = t;
58     theSize--;
59     delete p;
60     return x;
61 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

---

Now we can remove elements at the end/front in  $O(1)$  by **just adjusting the pointers!**



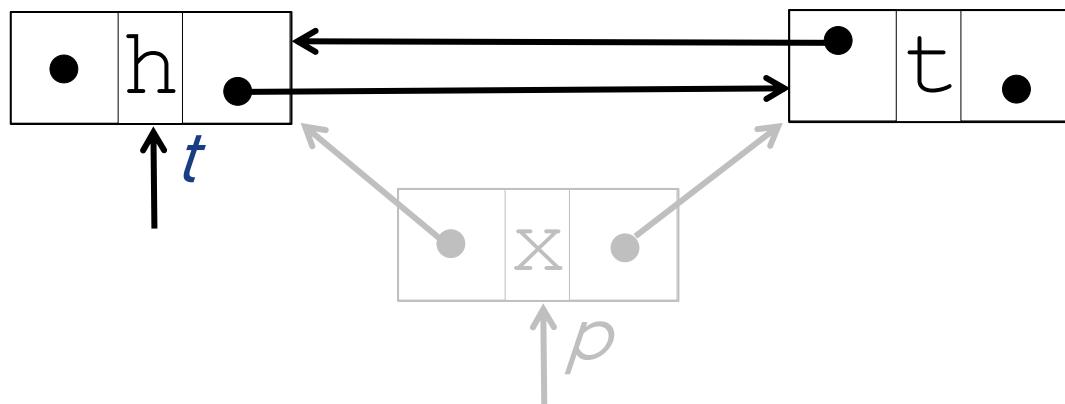
```
39 template <typename Object>
40 Object List<Object>::pop_front() {
41     Node *p = head->next;
42     Node *t = p->next;
43     Object x = p->data;
44     t->prev = head;
45     head->next = t;
46     theSize--;
47     delete p;
48     return x;
49 }
50
51 template <typename Object>
52 Object List<Object>::pop_back() {
53     Node *p = tail->prev;
54     Node *t = p->prev;
55     Object x = p->data;
56     t->next = tail;
57     tail->prev = t;
58     theSize--;
59     delete p;
60     return x;
61 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

---

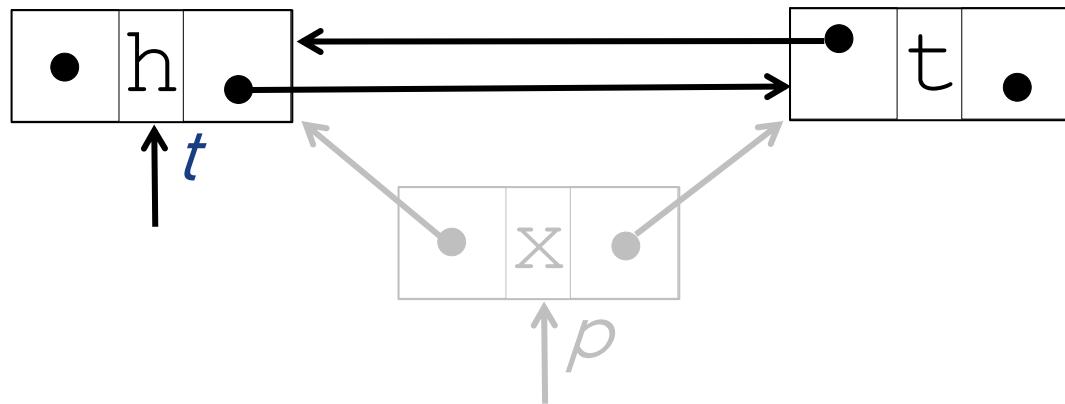
Now we can remove elements at the end/front in  $O(1)$  by **just adjusting the pointers!**



```
39 template <typename Object>
40 Object List<Object>::pop_front() {
41     Node *p = head->next;
42     Node *t = p->next;
43     Object x = p->data;
44     t->prev = head;
45     head->next = t;
46     theSize--;
47     delete p;
48     return x;
49 }
50
51 template <typename Object>
52 Object List<Object>::pop_back() {
53     Node *p = tail->prev;
54     Node *t = p->prev;
55     Object x = p->data;
56     t->next = tail;
57     tail->prev = t;
58     theSize--;
59     delete p;
60     return x;
61 }
```

double\_list.tpp

# DOUBLY LINKED LISTS



It also means we can remove any element in  $O(1)$  given a **pointer**!

Unfortunately, this would **break the abstraction**, so we will need **iterators** to make it elegant **again**.

```
39 template <typename Object>
40 Object List<Object>::pop_front() {
41     Node *p = head->next;
42     Node *t = p->next;
43     Object x = p->data;
44     t->prev = head;
45     head->next = t;
46     theSize--;
47     delete p;
48     return x;
49 }
```

```
50
51 template <typename Object>
52 Object List<Object>::pop_back() {
53     Node *p = tail->prev;
54     Node *t = p->prev;
55     Object x = p->data;
56     t->next = tail;
57     tail->prev = t;
58     theSize--;
59     delete p;
60     return x;
61 }
```

double\_list.tpp

# DOUBLY LINKED LISTS

Now we look at the Double Linked List implementation of the List ADT using the inheritance method:

C simple\_list.h

```
1 #pragma once
2
3 template <typename Object>
4 class List {
5     public:
6         virtual int size() = 0;
7         virtual bool empty() = 0;
8         virtual void clear() = 0;
9         virtual void push_front(const Object x) = 0;
10        virtual void push_back(const Object x) = 0;
11        virtual Object pop_front() = 0;
12        virtual Object pop_back() = 0;
13        virtual Object find_kth(int pos) = 0;
14    };
15
16
```

This is the same interface as the one implemented for Linked Lists.

C double\_linked\_list.h

```
1 #pragma once
2
3 #include <assert>
4 #include "simple_list.h"
5
6 template <typename Object>
7 class DoubleLinkedList: public List<Object>
8 {
9     private:
10        struct Node {
11            Object data;
12            Node *next;
13            Node *prev;
14        };
15        int theSize;
16        Node *head;
17        Node *tail;
18
19    public:
20
21        DoubleLinkedList() {
22            theSize = 0;
23            head = new Node; tail = new Node;
24            head->next = tail;
25            tail->prev = head;
26            head->prev = tail->next = nullptr;
27        }
28
29        ~DoubleLinkedList() {
30            clear();
31            delete head;
32            delete tail;
33        }
34
35        int size() { return theSize; }
36        bool empty() { return (size() == 0); }
37
38}
```

C double\_linked\_list.h

```
40 void clear() {
41     Node *p = head->next;
42     while (p != tail) {
43         Node *t = p->next;
44         delete p;
45         p = t;
46     }
47     head->next = tail;
48     tail->prev = head;
49 }
50
51 void push_front(const Object x) {
52     Node *p = new Node;
53     Node *t = head->next;
54     p->data = x;
55     p->prev = head;
56     p->next = t;
57     t->prev = p;
58     head->next = p;
59     theSize++;
60 }
61
62 void push_back(const Object x) {
63     Node *p = new Node;
64     Node *t = tail->prev;
65     p->data = x;
66     p->prev = t;
67     p->next = tail;
68     t->next = p;
69     tail->prev = p;
70     theSize++;
71 }
72
73 Object pop_front(){
74     Node *p = head->next;
75     Node *t = p->next;
76     Object x = p->data;
77     t->prev = head;
78     head->next = t;
79     theSize--;
80     delete p;
81     return x;
82 }
```

What is the complexity of these operations?

C double\_linked\_list.h

```
84 Object pop_back(){
85     Node *p = tail->prev;
86     Node *t = p->prev;
87     Object x = p->data;
88     t->next = tail;
89     tail->prev = t;
90     theSize--;
91     delete p;
92     return x;
93 }
94
95 Object find_kth(int pos){
96     assert(pos >= 0 && pos < theSize);
97     Node *p = head->next;
98     while (pos > 0) {
99         p = p->next;
100        pos--;
101    }
102    return p->data;
103 }
104 }
```



AARHUS  
UNIVERSITY

# STACKS, QUEUES, AND MATRICES

# STACKS

---

## Definition:

A *stack* is an ADT with two operations to *push* (insert) and *pop* (remove) an item.

Only the *top* element is **accessible**.



A stack implements a *LIFO (last-in-first-out) access pattern*.

# STACKS

---

We can solve the **maximum size limitation** of array-based stacks using a doubly linked list. The time complexity for the stack operations will remain the same:

- **Push or pop** take  $O(1)$ .
- Looking at the **top** element also take  $O(1)$ .

We can build a *template* stack directly over our doubly linked list.

# STACKS

---

The stack implementation is almost **trivial**:

- Note how we **instantiate** the linked list **with the same type**
- We can reuse the linked list operations **directly**
- Implementation of the linked list is abstracted away

That is **again** the **power of abstraction**.

```
1 #ifndef _STACK_H_
2 #define _STACK_H_
3
4 #include "../list/double_list.h"
5
6 template <typename Object>
7 class Stack {
8     private:
9         List<Object> *list;
10
11     public:
12     Stack() {
13         list = new List<Object>();
14     }
15
16     ~Stack() { delete list; }
17
18     bool empty() { return (list->size() == 0); }
19     Object top() { return list->find_kth(0); }
20     Object pop() { return list->pop_front(); }
21
22     void push(const Object x) {
23         list->push_front(x);
24     }
25 }
26
27 #endif
```

stack\_class.h

# STACKS

---

The stack implementation is almost **trivial**:

- Note how we **instantiate** the linked list **with the same type**
- We can reuse the linked list operations **directly**
- Implementation of the linked list is **abstracted away**

That is **again** the power of abstraction.

```
1 using namespace std;
2 #include <iostream>
3 #include "stack_class.h"
4
5 int main(int argc, char *argv[]) {
6     Stack<int> *stack = new Stack<int>();
7     stack->push(10);
8     stack->push(5);
9     stack->push(3);
10    stack->push(7);
11
12    cout << "Top element: " << stack->top() << endl;
13    while (stack->empty() == false) {
14        cout << "Next element: " << stack->pop() << endl;
15    }
16    cout << "Stack is empty? " << stack->empty() << endl;
17
18    delete stack;
19 }
```

test\_stack\_class.cpp

# QUEUES

---

## Definition:

A *queue* is an ADT with two operations to *put* (insert) and *get* (remove) an item.



Only the *front* element is **accessible**.

A queue implements a *FIFO* (*first-in-first-out*) **access pattern**.

**Remark.** The doubly head-tail linked list we implemented is also called a *double-ended queue* (deque).

# QUEUES

---

First let us try to implement a queue using an **array** as the **underlying data structure**.

Note how **size** and **capacity** must be handled.

```
1 #ifndef _QUEUE_H_
2 #define _QUEUE_H_
3
4 #include <cassert>
5
6 template <typename Object>
7 class Queue {
8     private:
9         int N;
10        int head, tail, size;
11        Object *data;
12
13    public:
14        Queue(int capacity) {
15            data = new Object[capacity];
16            N = capacity;
17            head = N;
18            size = tail = 0;
19        }
20
21        ~Queue() { delete[] data; }
22        bool empty() { return (size == 0); }
23
24        Object front() {
25            assert(empty() == false);
26            return data[head % N];
27        }
28
29        Object get() {
30            assert(empty() == false);
31            head = head % N;
32            Object x = data[head++];
33            size--;
34            return x;
35        }
36
37        void put(const Object x) {
38            assert(size < N);
39            data[tail] = x;
40            tail = (tail + 1) % N;
41            size++;
42        }
43    };
44
45 #endif
```

queue\_class\_array.h

How to make implementation more **general**?

# REFLECTION

---

Using the implementation of Queue of the previous slide, use pen and paper to execute the following operations and keep track of all variables inside the queue:

1. `Queue<int> *queue = new Queue<int>(4);`
2. `queue->put(10);`
3. `queue->put(5);`
4. `queue->put(3);`
5. `queue->put(7);`
6. `cout << queue->get() << endl;` // what is printed?
7. `cout << queue->get() << endl;` // what is printed?
8. `queue->put(2);`
9. `queue->put(4);`

# QUEUES

---

We can have flexibility by allocating a new array whenever **queue is full**. What is the **time complexity** for this variant?

We can solve the **maximum size limitation** of array-based queues using a doubly linked list **again**. The time complexity for the queue operations should remain as expected:

- **Put or get** take  $O(1)$ .
- Looking at the **back** element also takes  $O(1)$ .

# QUEUES

---

Now the queue implementation is **trivial**:

- Note how we **instantiate** the linked list **with the same type**
- We can reuse the linked list operations **directly**
- Implementation of the linked list is **abstracted away**

```
1 #ifndef _QUEUE_H_
2 #define _QUEUE_H_
3
4 #include "../list/double_list.h"
5
6 template <typename Object>
7 class Queue {
8     private:
9         List<Object> *list;
10
11     public:
12         Queue() {
13             list = new List<Object>();
14         }
15
16         ~Queue() { delete list; }
17
18         bool empty() { return (list->size() == 0); }
19         Object front() { return list->find_kth(0); }
20         Object get() { return list->pop_front(); }
21
22         void put(const Object x) {
23             list->push_back(x);
24         }
25     };
26
27 #endif
```

queue\_class.h

# MATRICES

# MATRICES IN C++

---

C++ is a **superset** of the C programming language, so the same basic types are also **supported**.

Hence C++ supports multi-dimensional arrays **over all types**. This includes **basic** types (int, float, etc), **structs** and **strings**, object **instances** and naturally all **user-defined types**.

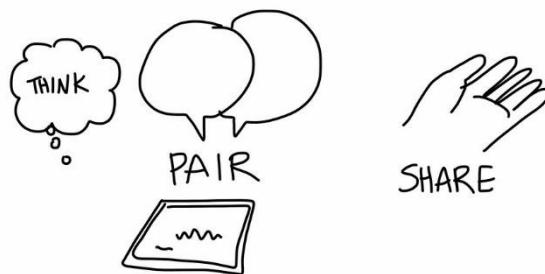
*What is missing in terms of functionality?*

# MATRICES IN C++

---

Just like plain arrays, multi-dimensional arrays in C have capacity fixed at **declaration time** and cannot afford **dynamic data sets**.

*How can we solve this using the data structures we know?*



# MATRICES IN C++

---

Just like plain arrays, multi-dimensional arrays in C have capacity fixed at **declaration time** and cannot afford **dynamic data sets**.

*How can we solve this using the data structures we know?*

Notice that the STL does **not** have matrix, so C++ users can **continue** using C multi-dimensional arrays. There are **libraries** which implement **faster** matrix arithmetic.

→ **Note:** for basic types, `std::valarray` (array of **values**) is **faster** than `std::vector`.

# LINEAR ALGEBRA PROGRAMS

---

- We are going to discuss the use of matrices and vectors in programs
- Our interest is less in the mathematics but more in the **representation of matrices and vectors** in a program
- The mathematics is discussed in detail in the course *Numerical Linear Algebra* in the 2<sup>nd</sup> semester

# VECTORS AND MATRICES REFRESHER

---

- A **matrix** is a rectangular table of numbers (we use `int`)

$$\text{rows} \begin{pmatrix} a_{(0,0)} & a_{(0,1)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & a_{(1,1)} & \cdots & a_{(1,n-1)} \\ \vdots & \vdots & \vdots & \vdots \\ a_{(m-1,0)} & a_{(m-1,1)} & \cdots & a_{(m-1,n-1)} \end{pmatrix} \text{columns}$$

- The **dimension** of a matrix is described by its number of rows and columns
- The matrix above has dimension  $m \times n$

# VECTORS AND MATRICES REFRESHER

---

- A **row vector** is a matrix of dimension  $1 \times n$

$$\begin{pmatrix} a_{(0,0)} & a_{(0,1)} & \cdots & a_{(0,n-1)} \end{pmatrix}$$

consisting of one row

- A **column vector** is a matrix of dimension  $m \times 1$

$$\begin{pmatrix} a_{(0,0)} \\ a_{(1,0)} \\ \vdots \\ a_{(m-1,0)} \end{pmatrix}$$

consisting of one column

# MATRIX MULTIPLICATION

---

- Matrices are multiplied component wise

$$\begin{pmatrix} a_{(0,0)} & a_{(0,1)} & \cdots & a_{(0,m-1)} \\ a_{(1,0)} & a_{(1,1)} & \cdots & a_{(1,m-1)} \\ \vdots & \vdots & \vdots & \vdots \\ a_{(k-1,0)} & a_{(k-1,1)} & \cdots & a_{(k-1,m-1)} \end{pmatrix} * \begin{pmatrix} b_{(0,0)} & b_{(0,1)} & \cdots & b_{(0,n-1)} \\ b_{(1,0)} & b_{(1,1)} & \cdots & b_{(1,n-1)} \\ \vdots & \vdots & \vdots & \vdots \\ b_{(m-1,0)} & b_{(m-1,1)} & \cdots & b_{(m-1,n-1)} \end{pmatrix} = \begin{pmatrix} c_{(0,0)} & c_{(0,1)} & \cdots & c_{(0,n-1)} \\ c_{(1,0)} & c_{(1,1)} & \cdots & c_{(1,n-1)} \\ \vdots & \vdots & \vdots & \vdots \\ c_{(k-1,0)} & c_{(k-1,1)} & \cdots & c_{(k-1,n-1)} \end{pmatrix}$$

where

$$c_{(i,j)} = a_{(i,0)} * b_{(0,j)} + a_{(i,1)} * b_{(1,j)} + \dots + a_{(i,m-1)} * b_{(m-1,j)} .$$

- The dimensions of the matrices must “match”: the first matrix has the dimension  $k \times m$ , the second  $m \times n$  and the result the dimension  $k \times n$
- memorization rule:  $k \times m - m \times n \rightarrow k \times n$

# REFLECTION

---

Write down the resulting matrix C obtained by multiplying the following two matrices:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} * \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix} = C$$

How many multiplications and summations did you perform per element in C?

# REPRESENTING MATRICES BY ARRAYS

---

- Arrays appear to be the ideal data structure to represent matrices
- Given a matrix

$$\begin{pmatrix} a_{(0,0)} & a_{(0,1)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & a_{(1,1)} & \cdots & a_{(1,n-1)} \\ \vdots & \vdots & \vdots & \vdots \\ a_{(m-1,0)} & a_{(m-1,1)} & \cdots & a_{(m-1,n-1)} \end{pmatrix}$$

- we can rearrange the rows as follows

$$( a_{(0,0)} \ a_{(0,1)} \ \cdots \ a_{(0,n-1)} \ a_{(1,0)} \ a_{(1,1)} \ \cdots \ a_{(1,n-1)} \ \cdots \ a_{(m-1,0)} \ a_{(m-1,1)} \ \cdots \ a_{(m-1,n-1)} )$$

- Of course, the numbering of the coordinates changes and we have to relate it to the one indicated

# REPRESENTING MATRICES BY ARRAYS

---

- This can be done as follows

$$\begin{matrix} x[0] & x[1] & \cdots & x[n-1] & x[n] & x[n+1] & \cdots & x[2*n-1] & \cdots & x[(m-1)*n] & x[(m-1)*n+1] & \cdots & x[(m-1)*(n-1)] \\ \uparrow & \uparrow & & \uparrow & \uparrow & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow \\ (a_{(0,0)} & a_{(0,1)} & \cdots & a_{(0,n-1)} & a_{(1,0)} & a_{(1,1)} & \cdots & a_{(1,n-1)} & \cdots & a_{(m-1,0)} & a_{(m-1,1)} & \cdots & a_{(m-1,n-1)} & ) \end{matrix}$$

- Using this approach we can access coordinate  $a_{(i,j)}$  in the array  $x$  as  $x[i*n+j]$
- This is how the computer represents **two-dimensional arrays** when we compile a program
- We rather use two-dimensional arrays directly
  - They correspond closely to the data we want to represent
  - They help us to avoid programming errors

# REFLECTION

---

Write the single array representation of the following matrix.

$$A = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

What is the element returned by the following accesses?

$$A[0] = ?$$

$$A[2] = ?$$

$$A[3] = ?$$

$$A[4] = ?$$

# TWO-DIMENSIONAL ARRAYS

---

- To represent the matrix

$$\begin{pmatrix} a_{(0,0)} & a_{(0,1)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & a_{(1,1)} & \cdots & a_{(1,n-1)} \\ \vdots & \vdots & \vdots & \vdots \\ a_{(m-1,0)} & a_{(m-1,1)} & \cdots & a_{(m-1,n-1)} \end{pmatrix}$$

we can use the two dimensional array declared by  
`int a[m][n];`

- Coordinate  $a_{(i,j)}$  is accessed writing `a[i][j]`
- A value `v` is assigned to  $a_{(i,j)}$  writing  
`a[i][j] = v;`

# TWO-DIMENSIONAL ARRAYS

---

- A matrix can be initialised with a constant

```
int a[2][4] =  
    { { 1, 2, 3, 4 }, { 3, 2, 1, 4 } };
```

- Coordinate  $a_{(i,j)}$  is accessed writing  $a[i][j]$
- A value  $v$  is assigned to  $a_{(i,j)}$  writing  
 $a[i][j] = v;$

# MATRIX MULTIPLICATION PROGRAM

---

- The program on the right-hand side multiplies the matrices  $a$  and  $b$  to yield  $c$

$$b \rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$
$$a \rightarrow \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix} \begin{pmatrix} 3 & 5 & 7 \\ 7 & 5 & 3 \end{pmatrix} \leftarrow c$$

```
/*
 * Multiply two matrices.
 */

#include <stdio.h> /* scanf, printf */

/* program body */
int main(void)
{
    int a[2][4] = { { 1, 2, 3, 4}, { 4, 3, 2, 1} };
    int b[4][3] = { { 1, 0, 0 }, { 1, 1, 0 }, { 0, 1, 1 }, { 0, 0, 1 } };
    int c[2][3]; /* result of the multiplication */
    int i; /* counter variable for the rows of c */
    int j; /* counter variable for the columns of c */
    int k; /* counter variable for the sum */

    /* precondition */
    /* true */

    /* postcondition */
    /* compute product of matrices a and b
     * according to the formula on the slides
     */
    for (i = 0; i < 2; i = i + 1)
        for (j = 0; j < 3; j = j + 1)
    {
        c[i][j] = 0;
        for (k = 0; k < 4; k = k + 1)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }

    /* print output */
    for (i = 0; i < 2; i = i + 1)
    {
        for (j = 0; j < 3; j = j + 1)
            printf("%d ", c[i][j]);
        printf("\n");
    }

    /* return from body and exit program */
    return 0;
}
```

From programming for computer engineering, week 4

# REFLECTION

---

What is the worst-case complexity of adding two matrices A and B, with sizes  $k \times m$  and  $k \times m$ , respectively?

What is the worst-case complexity of multiplying matrices A and B, with sizes  $k \times m$  and  $m \times n$ , respectively?

# MATRICES IN C++

As before, let us start from the interface of a Matrix ADT, and introduce several **language features** for productivity.

A **dynamic** matrix is a vector of vectors, as expected.

This loop is particularly compact!  
A **natural** way of performing the same would be using **iterators**.

Now take a deeper look over **initialization lists** and **range-based** for loops.

```
1 #ifndef MATRIX_H
2 #define MATRIX_H
3
4 using namespace std;
5
6 #include <iostream>
7 #include <iomanip>
8 #include <vector>
9
10 template <typename Object>
11 class Matrix {
12     private:
13         vector<vector<Object>> array;
14
15     public:
16         Matrix() {};
17
18         Matrix(int rows, int cols) : array(rows) {
19             for(auto& r : array) {
20                 r.resize(cols);
21             }
22         }
23
24         Matrix(vector<vector<Object>> v) : array{v} {}
25
26         int numrows() const { return array.size(); }
27
28         int numcols() const {
29             if (numrows() > 0) {
30                 return array[0].size();
31             }
32             return 0;
33         }
34 }
```

A new **header** for input/output formatting.

Constructors initialize members!

matrix\_class.h



# MORE LANGUAGE FEATURES

---

An **initialization list** is used to initialize the data members directly.  
The **syntax** for constructors is below:

```
class-name (parameters) : member_initializers
```

Each **member initializer** has the following syntax in C++11:

```
identifier expression-list  
identifier brace-init-list
```

# MORE LANGUAGE FEATURES

---

Some **examples** can be seen in the following:

- Notice the **syntax and usage**
- The order is **not** important
- One initializer does **not** interfere in another one.
- The constructor is empty because initialization happens **automatically**.

init\_list.cpp

```
1 #include <iostream>
2 #include <assert.h>
3 using namespace std;
4
5 class X {
6 public:
7     int a, b, i, j;
8     const int& r;
9     int *p;
10
11     X(int i)
12         : r(a) // initializes X::r to refer to X::a. Kind of like an "alias".
13             // Every access to X::r is implicitly
14                 // de-referencing the pointer to get to the value of a.
15             , p(&a) // initializes X::p to refer to X::p. Almost equivalent to X::r
16                 // except now we have to de-reference p explicitly, and it is not const.
17             , b{i} // initializes X::b to the value of the parameter i, using the default constructor
18                 // Read more in https://www.modernescpp.com/index.phpinitialization
19             , i(i) // initializes X::i to the value of the parameter i
20             , j(this->i) // initializes X::j to the value of X::i
21             , a{i} // initializes X::a to the value of i
22     { }
23 }
```

# MORE LANGUAGE FEATURES

---

init\_list.cpp

```
1 #include <iostream>
2 #include <assert.h>
3 using namespace std;
4
5 class X {
6 public:
7     int a, b, i, j;
8     const int& r;
9     int *p;
10
11    X(int i)
12        : r(a) // initializes X::r to refer to X::a. Kind of like an "alias".
13          // Every access to X::r is implicitly
14          // de-referencing the pointer to get to the value of a.
15        , p(&a) // initializes X::p to refer to X::p. Almost equivalent to X::r
16          // except now we have to de-reference p explicitly, and it is not const.
17        , b{i} // initializes X::b to the value of the parameter i, using the default constructor.
18          // Read more in https://www.modernescpp.com/index.phpinitialization
19        , i(i) // initializes X::i to the value of the parameter i
20        , j(this->i) // initializes X::j to the value of X::i
21        , a{i} // initializes X::a to the value of i
22    {}
23 }
```

init\_list.cpp

<https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/>

<https://ncona.com/2019/01/variable-initialization-in-cpp/>

```
25    class Y {
26        public:
27            int a, b, i, j;
28            const int& r = a; // equivalent to r(a) initializer in X
29            int *p;
30
31            Y(int i)
32            { // The following are equivalent to the respective initializer in X
33                p = &a;
34                b = i;
35                this->i = i;
36                j = this->i;
37                a = i;
38            }
39        };
40
41    int main(int argc, char *argv[])
42    {
43        X someX(10);
44        X *anotherX = new X(5);
45
46        assert(someX.a == 10 && anotherX->a == 5);
47        assert(someX.b == 10 && anotherX->b == 5);
48        assert(someX.i == 10 && anotherX->i == 5);
49        assert(someX.j == 10 && anotherX->j == 5);
50        assert(someX.r == 10 && anotherX->r == 5);
51        assert(*someX.p == 10 && (*anotherX->p) == 5);
52
53        Y someY(20);
54        assert(someY.a == 20);
55        assert(someY.b == 20);
56        assert(someY.i == 20);
57        assert(someY.j == 20);
58        assert(someY.r == 20);
59        assert(*someY.p == 20);
60
61        delete anotherX;
```

# MORE LANGUAGE FEATURES

---

A **range-based for loop** was added in C++11. It is used as a more readable **equivalent** to the traditional for loop operating over a **range of values**, such as all elements in a container:

```
vector<int> vec;  
vec.push_back(10);  
vec.push_back(20);  
  
for (auto i : vec) {  
    cout << i;  
}
```

Notice how the iterator is **implicit**. We can also use the `auto` keyword for **automatic type inference**.

# MATRICES IN C++

Let us go back the Matrix **operators**:

- The **subscript operator** can be overloaded to return a (constant) **reference**.
- Notice the **exceptions**, although **not** mandatory as in `at()`.
- The matrix **addition** behaves like the matrix instance is **mutable**.
- Printing fixes **precision** and **reserves space** per field (`setw`).

```
35     const vector<Object>& operator[](int row) const {
36         if (row < 0 || row >= array.size())
37             throw out_of_range("Invalid row.");
38         return array[row];
39     }
40
41     vector<Object>& operator[](int row) {
42         if (row < 0 || row >= array.size())
43             throw out_of_range("Invalid row.");
44         return array[row];
45     }
46
47     friend ostream& operator<<(ostream& t, Matrix<Object> mat) {
48         cout << fixed;
49         cout.precision(2);
50         for(int i = 0; i < mat.numrows(); ++i) {
51             cout << "|";
52             for(int j = 0; j < mat.numcols(); ++j) {
53                 cout << " " << setw(6) << mat[i][j] << " ";
54             }
55             cout << "|" << endl;
56         }
57         return t;
58     }
59
60     void add(Matrix& mat);
61 };
62
63 #include "matrix_class.tpp"
64
65 #endif
```

Does not modify  
instance!

matrix\_class.h

# AGENDA

---

- ✓ Matrices in C++ and more language features
- Matrix arithmetic

# MATRIX ARITHMETIC

---

The matrix **addition** operation as a member function:

- Error handling when matrix dimensions are **not** compatible

```
1 template <typename Object>
2 void Matrix<Object>::add(Matrix& mat) {
3     int rows = numrows();
4     int cols = numcols();
5
6     if (rows != mat.numrows() || cols != mat.numcols()) {
7         throw invalid_argument{"Matrix has incompatible dimensions"};
8     }
9
10    for(int i = 0; i < rows; ++i) {
11        for(int j = 0; j < cols; ++j) {
12            array[i][j] = array[i][j] + mat[i][j];
13        }
14    }
15 }
```

matrix\_class.tpp

→ *Beware of side-effects!*

# REFLECTION

---

In the previous slide, what is the time complexity of the add operation?

1. for square matrices of size  $N$  ?
2. for matrices with size  $R \times C$  ?

# MATRIX ARITHMETIC

---

Matrix multiplication is implemented **externally** (*why?*)

- Same exception when matrix dimensions are **not** compatible

```
17 template <typename Object>
18 Matrix<Object>& multiply(Matrix<Object>& a, Matrix<Object>& b) {
19     if (a.numcols() != b.numrows()) {
20         throw invalid_argument{"Matrix has incompatible dimensions"};
21     }
22
23     Matrix<Object> *mult = new Matrix<Object>(a.numrows(), b.numcols());
24
25     for(int i = 0; i < a.numrows(); ++i) {
26         for(int j = 0; j < b.numcols(); ++j) {
27             (*mult)[i][j] = 0;
28         }
29     }
30
31     for(int i = 0; i < a.numrows(); ++i) {
32         for(int j = 0; j < b.numcols(); ++j) {
33             for (int k = 0; k < b.numrows(); k++) {
34                 (*mult)[i][j] += a[i][k] * b[k][j];
35             }
36         }
37     }
38
39     return *mult;
40 }
```

matrix\_class.tpp

# REFLECTION

---

In the previous slide, what is the time complexity of the multiply operation?

1. for square matrices of size  $N$  ?
2. for matrices with size  $R \times C$  ?

Is there any situation where the multiply operation can perform less steps than the complexity above? I.e., is there a lower bound?

# REFLECTION

---

Now, the testing code. One matrix is **randomized**, the other is the **identity matrix**.

Notice the attempt to write to an **invalid** position (line 24). What **exactly** happens there?

Finally, notice the difference between the add/mult operations.

```
1 #include <iostream>
2 #include <stdexcept>
3 #include "matrix_class.h"
4 #include "stdlib.h"
5 using namespace std;
6
7 #define ROWS      10
8 #define COLS      5
9 #define MAX       100
10
11 int main(void) {
12     Matrix<double> mat(ROWS, COLS), id(COLS, COLS);
13     srand(time(0));
14
15     for (int i = 0; i < COLS; i++) id[i][i] = 1; // initialize id
16
17     for (int i = 0; i < ROWS; i++) {           // randomize mat
18         for (int j = 0; j < COLS; j++) {
19             mat[i][j] = ((double) rand() / (RAND_MAX)) * MAX;
20         }
21     }
22
23     try {
24         mat[ROWS][COLS] = 0.0f;                  // invalid access
25     } catch (exception& e) {
26         cout << e.what() << endl;
27         mat[ROWS - 1][COLS - 1] = 0.0f;
28     }
29
30     cout << "mat=" << endl << mat << endl;
31     mat.add(mat);
32     cout << "add=" << endl << mat << endl;
33
34     cout << "id =" << endl << id << endl;
35     Matrix<double>& mult = multiply(mat, id);
36     cout << "mult=" << endl << mult << endl;
37
38     delete &mult;
39 }
```

test\_matrix.cpp

# EXAMPLE PROBLEMS SOLVED USING MATRICES

---

Matrices are commonly used in memoization.

We revisit this topic later in this course, but if you are curious, you can find more information here: <https://en.wikipedia.org/wiki/Memoization>

Example problem that can be efficiently solved using memoization:

- [https://en.wikipedia.org/wiki/Dynamic\\_programming#Matrix\\_chain\\_multiplication](https://en.wikipedia.org/wiki/Dynamic_programming#Matrix_chain_multiplication)

# SETS, MAPS AND HASHING

# AGENDA

---

- Sets, Maps and Hash Tables
  - Hash functions
  - Hash Table with Chaining
  - Hash Table with Probing

# SETS AND MAPS

---

Today we will see the **hash table ADT**, which allows a subset of operations in  $O(1)$  in average:

- Efficient **insertion** and **deletion** of elements
- Efficient **search** (*lookup*) of elements

There are **many** applications of hash tables, and they are useful whenever a *map* or *key-value store* is useful. A typical application is implementing a compiler **symbol table**.

Notice that some **other** operations may not be efficient anymore.

# MAPS OR DICTIONARIES

---

## Definition:

A *map* or *dictionary* or *key-value store* is an **unordered collection** of items of any type indexed by *keys* (of a simple type (integer or string)). The items can have additional data members (*values*). The keys must be **unique**, so duplicates are not allowed.

A map has **three operations**: **insert**, **contains** and **remove** which respectively **add**, **search** and **delete** a *(key, value)* pair.

As previously, we will **abstract** the values and focus on the keys.

Can you illustrate the difference between sets and maps?



## Definition:

A *set* is an **unordered collection** of items of any type where no duplicates are allowed. It is a computer implementation of the mathematical concept of a finite set.

A set has these **typical operations**: **insert (e)** , **contains (e)** and **remove (e)** which respectively **add**, **search** and **delete** an element.

**Set vs Map:** Later represents values indexed by their keys. Compare the signature of the contains function, for example.

# HASH TABLES

Hash tables are the preferred data structure to implement sets and maps.

We will implement the hash table as an **array of fixed size  $M$** . Each key is mapped into some number in  $[0, M-1]$  and stored in the corresponding position.

On the right, a hash table with  $M = 11$  positions store items indexed by the names (strings).

What is hashing, and what is it used for?

In slide 6, explain "On the right, a hash table with  $M = 11$  positions store items indexed by the names(strings)."

How can it be indexed by name or number when the figure shows otherwise

0	
1	
2	
3	John (25000)
4	Phil (31250)
5	
6	Dave (27500)
7	Mary (28200)
8	
9	
10	

# HASH TABLES

---

The *mapping* is called a **hash function**, which should be simple to compute and ensure that keys are **distributed evenly** among the different positions.

The remaining problems are **choosing** a hash function, and deciding what to do when keys hash to the **same value** (*collision*).

0	
1	
2	
3	John (25000)
4	Phil (31250)
5	
6	Dave (27500)
7	Mary (28200)
8	
9	
10	

# REFLECTION

---

Which one of the following is the right signature of the hash function?

- *Key hash(Value)*
- *Value hash(Key)*
- *int hash(Key)*
- *int hash(Value)*

# AGENDA

---

- ✓ Sets, Maps and Hash Tables
- Hash functions
  - Hash Table with Chaining
  - Hash Table with Probing

# HASH FUNCTIONS

First, we need to decide how to map keys to array positions.

If the keys are **integers**, we can simply return integers *modulo M*. It is often a good idea to select **prime M**, such that random keys will be **evenly distributed** in  $[0, M-1]$ .

→ See Jupyter notebook (`explanation_primes.ipynb`) provided with this week's code for why.

If keys are **strings**, the hash function must be careful. For example:

```
unsigned int hash(const string& key, int tableSize) {  
    unsigned int hashVal = 0;  
    for( char ch : key ) hashVal = 37 * hashVal + ch;  
    return hashVal % tableSize;  
}
```

Letter	Frequency
E	16.09 %
R	7.63 %
N	7.32 %
T	7.19 %
D	6.65 %
A	6.13 %
I	5.73 %
S	5.18 %
L	4.90 %

Frequency of Danish letters

<https://www.sttmedia.com/characterfrequency-danish>

# REFLECTION

---

Slide 11

⋮

In the previous slide, is it possible that there exists two **different** strings  $s_1$  and  $s_2$  such that  $\text{hash}(s_1) = \text{hash}(s_2)$ ?

If so, how can we distinguish them?

# HASH FUNCTIONS

We follow the C++11 function object templates:

```
template <typename Key>
class hash {
public:
    size_t operator()(const Key& k) const;
};
```

Default implementations are provided for standard types. We can extend to custom types.

```
1 // Example of an Employee class
2 class Employee {
3     public:
4         const string& getName() const {
5             return name;
6         }
7
8         bool operator==(const Employee & rhs) const {
9             return getName() == rhs.getName();
10        }
11
12        bool operator!=(const Employee & rhs) const {
13            return !(*this == rhs);
14        }
15        // Additional public members not shown private:
16
17     private:
18         string name;
19         double salary;
20         int seniority;
21         // Additional private members not shown
22     };
23
24 namespace std {
25     template <>
26     class hash<Employee> {
27         public:
28             size_t operator()(const Employee& item) {
29                 static hash<string> hf;
30                 return hf(item.getName());
31             }
32     };
33 }
```

Only equality operators!

Custom hashing

employee\_class.h

# HASH FUNCTIONS

---

We reduce hashing **instances** of user-defined classes to hashing members using the **default** implementations. This gives a good distribution of keys in average.

Now we need to solve collisions. We will study **two approaches**:

1. Separate **chaining** of colliding keys in a linked list
2. **Open addressing** using **probing** of keys

# AGENDA

---

- ✓ Sets, Maps and Hash Tables
- ✓ Hash functions
- Hash Table with Chaining
- Hash Table with Probing

# HASH TABLES WITH CHAINING

Load factor??

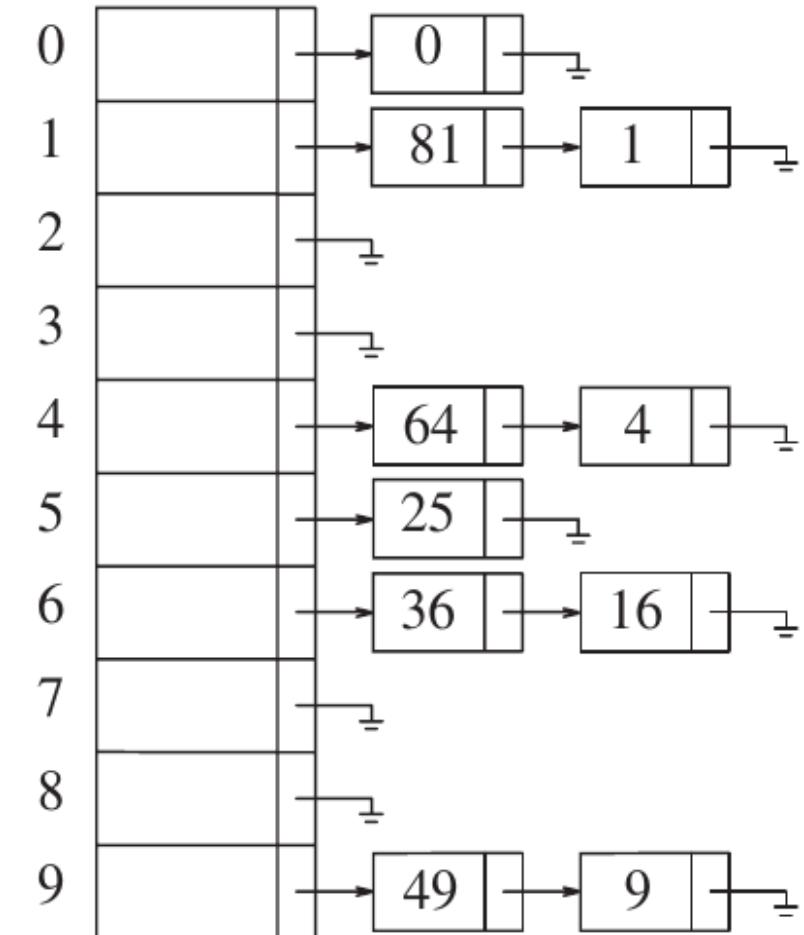


In separate chaining, each position of the array point to a linked list of **colliding** items.

Note that node in the list still contains the original key and implements *operator==*, so colliding elements can be distinguished by comparing them directly.

We can implement insert, search and deletion by first **mapping** to a linked list and then **operating** over the list.

Define the *load factor*  $\lambda$  as the **ratio of elements** in the table to the table **size**.



Hash table with chaining and load factor 1.0

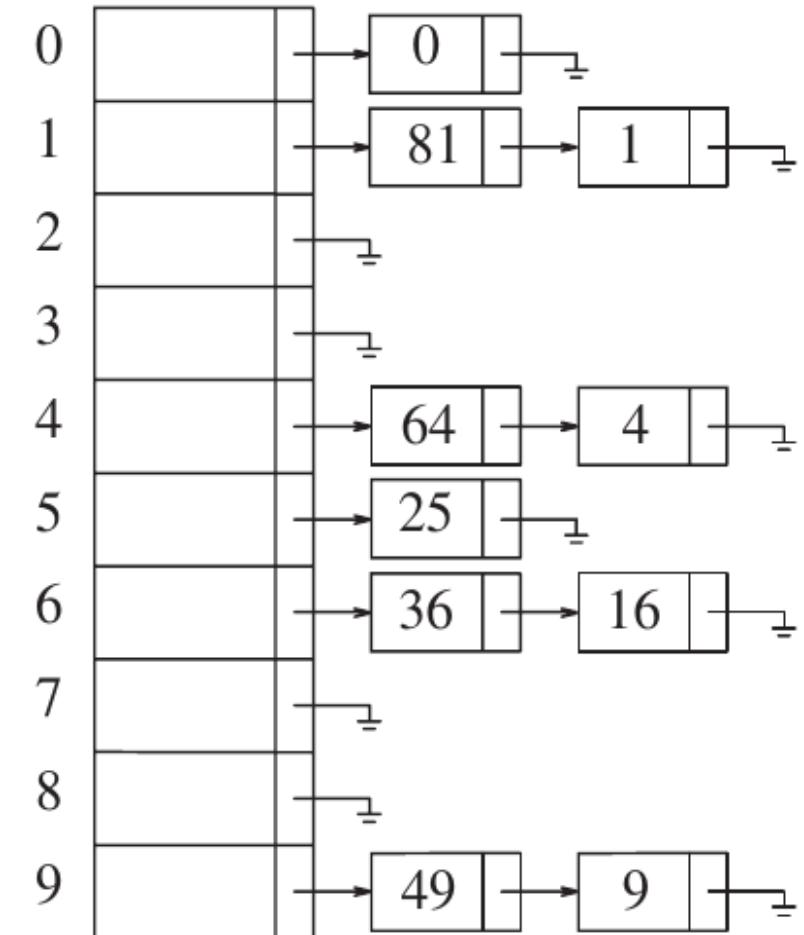
# HASH TABLES WITH CHAINING

---

The disadvantage of this approach is **memory consumption**, due to the many linked lists.

Operations cost  $O(\lambda)$  in **average**, since this is the expected length of each list.

If the load factor is kept below 1.0, operations run in  $O(1)$  in average, assuming that objects will be evenly distributed.



Hash table with chaining and load factor 1.0

# HASH TABLE INTERFACE

---

hash\_table\_chaining.h

**Templated** hash table has:

- Constructor uses **prime** size  $M$
- Internal data structure as a **vector of lists of keys**
- **Custom** function for hashing keys modulo  $M$
- If capacity is reached and load factor **degenerates**, `rehash()` increases table

```
1 #ifndef _SEPARATE_CHAINING_H_
2 #define _SEPARATE_CHAINING_H_
3
4 #include <vector>
5 #include <list>
6 #include <string>
7 #include <algorithm>
8 #include <functional>
9 using namespace std;
10
11 int nextPrime(int n);
12 bool isPrime(int n);
13
14 template<typename HashedObj>
15 class HashTable {
16 public:
17     HashTable(int size = 101):currentSize {0} {
18         theLists.resize(nextPrime(size));
19     }
20
21     void makeEmpty();
22     bool contains(const HashedObj& x) const;
23     bool insert(const HashedObj& x);
24     bool remove(const HashedObj& x);
25
26 private:
27     vector<list<HashedObj>> theLists; // The array of Lists
28     int currentSize;
29
30     size_t myhash(const HashedObj& x) const;
31     void rehash();
32
33 };
34
35 #include "hash_table_chaining.tpp"
36
37 #endif
```

# HASH TABLE IMPL

- Each operation starts by **hashing to a linked list** and returns success or failure
- Insertion and deletion keep track of current size to increase table when **needed**
- Notice the use of automatic typing to **simplify** the code and std::find to remove explicit **iterator loops**

```
1 template<typename HashedObj> hash_table_chaining.tpp
2 bool HashTable<HashedObj>::contains(const HashedObj& x) const {
3     auto& whichList = theLists[myhash(x)];
4     return find(begin(whichList), end(whichList), x) != end(whichList);
5 }
6
7 template<typename HashedObj>
8 void HashTable<HashedObj>::makeEmpty() {
9     for (auto& thisList : theLists)
10         thisList.clear();
11 }
12
13 template<typename HashedObj>
14 bool HashTable<HashedObj>::insert(const HashedObj& x) {
15     auto& whichList = theLists[myhash(x)];
16     if (find(begin(whichList), end(whichList), x) != end(whichList))
17         return false;
18     whichList.push_back(x);
19
20     // Rehash; see Section 5.5
21     if (++currentSize > theLists.size())
22         rehash();
23
24     return true;
25 }
26
27 template<typename HashedObj>
28 bool HashTable<HashedObj>::remove(const HashedObj& x) {
29     auto& whichList = theLists[myhash(x)];
30     auto itr = find(begin(whichList), end(whichList), x);
31
32     if (itr == end(whichList))
33         return false;
34
35     whichList.erase(itr);
36     --currentSize;
37     return true;
38 }
```

# HASH TABLE IMPL

- The hash function `myhash()` relies on a function object `template hash()` for the **template type**
- Increasing the hash table keeps the size prime and **moves** the old table to the new one
- The auxiliary functions at the end handle **primality**

```
hash_table_chaining.tpp
40 template<typename HashedObj>
41 size_t HashTable<HashedObj>::myhash(const HashedObj& x) const {
42     static hash<HashedObj> hf;
43     return hf(x) % theLists.size();
44 }
45
46 template<typename HashedObj>
47 void HashTable<HashedObj>::rehash() {
48     vector <list<HashedObj>> oldLists = theLists;
49
50     // Create new double-sized, empty table
51     theLists.resize(nextPrime(2 * theLists.size()));
52     for (auto & thisList:theLists) {
53         thisList.clear();
54     }
55
56     // Copy table over
57     currentSize = 0;
58     for (auto& thisList:oldLists) {
59         for (auto& x:thisList) {
60             insert(std::move(x));
61         }
62     }
63 }
64
65 int nextPrime(int n) {
66     n += (n % 2 ? 0 : 1);
67     while (!isPrime(n)) n += 2;
68     return n;
69 }
70
71 bool isPrime(int n) {
72     if (n == 2 || n == 3) return true;
73     if (n == 1 || n % 2 == 0) return false;
74     for (int i = 3; i * i <= n; i += 2)
75         if (n % i == 0)
76             return false;
77
78     return true;
79 }
```

# AGENDA

---

Slide 21-25, probing?



- ✓ Sets, Maps and Hash Tables
- ✓ Hash functions
- ✓ Hash Table with Chaining
- Hash Table with Probing

# HASH TABLES WITH PROBING

---

Slide 21-25, probing?



The chaining collision resolution method requires the implementation of **another** data structure and additional memory to store the linked lists.

*What if we use the hash table **idle space**?*

This is called **open-addressing**. When a collision is found, a hash function is **iterated** times until a new free position is found:

$$h_i(x) = \text{hash}(x) + f(i) \bmod M$$

# HASH TABLES WITH PROBING

Slide 23 (After 58, After 69)

Slide 21-25, probing?

There are three popular approaches:

1. **Linear probing:**  $f(i) = i$ , which means that positions are checked in sequence until a free one is found. Suffers from **primary clustering**, as blocks start to accumulate.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

# HASH TABLES WITH PROBING

---

**2. Quadratic probing:**  $f(i) = i^2$ , which means that positions are checked in increasing distance until a free one is found.  
Suffers from **secondary clustering**.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

# HASH TABLES WITH PROBING

slide 25-26

⋮

---

**3. Double hashing:**  $f(i) = i * \text{hash}_2(x)$ , which needs another uncorrelated hash function to solve collisions. It is **marginally statistically** better than quadratic probing but typically slower due to additional hashing. Quadratic probing **suffices** for us.

*How to handle deletion in open addressing?*

→ Notice that for probing the load factor can never **exceed** 1.0

*How should the contains method be implemented in linear probing?*

*Consider the following hash table, and reflect upon the steps that contains (26) should take.*

0	16
1	6
2	
3	
4	
5	
6	56
7	46
8	36
9	26

# REFLECTION

## *How to handle deletion in open addressing?*

Consider the following hash table implemented with linear probing:

0	16
1	6
2	
3	
4	
5	
6	56
7	46
8	36
9	26

After  
remove(36)



0	16
1	6
2	
3	
4	
5	
6	56
7	46
8	
9	26

Do you foresee any problem in the implementation of a “contains” method that you thought about in the previous slide? (e.g., what happens when the call contains (26) is made?)

# COMPLEXITY

---

Proportional to the length of the contiguous block of occupied cells at which the operation starts

a maximal block of  $k$  occupied cells will have probability  $k/N$  containing the start:  $O(k)$

# PROBLEMS OF QUADRATIC PROBING

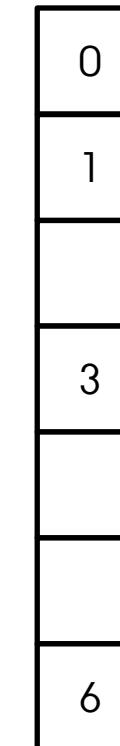
---

It may happen that quadratic probing may never end, for a sufficiently occupied, but not full, hash table.

Example: table with 7 elements,  $\text{hash}(x) = x \bmod 7$ ,

And element with key 6:

Probing step ( $i^2$ )	Position $((x \bmod 7) + i^2) \bmod 7$
0	6
1	0
2	3
3	1
4	1
5	3
6	0
7	6
8	0
9	3
...	



Slide 29

Problems of quadratic probing



Quadratic probing seems to never hit positions 2,4, and 5 of the table, even though they are empty.

# PROBLEMS OF QUADRATIC PROBING

---

To avoid the problem in the previous slides, we have theorem from [Weiss]

## **Theorem 5.1**

If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

So the rule to remember is: always keep  $\lambda < 0.5$ .

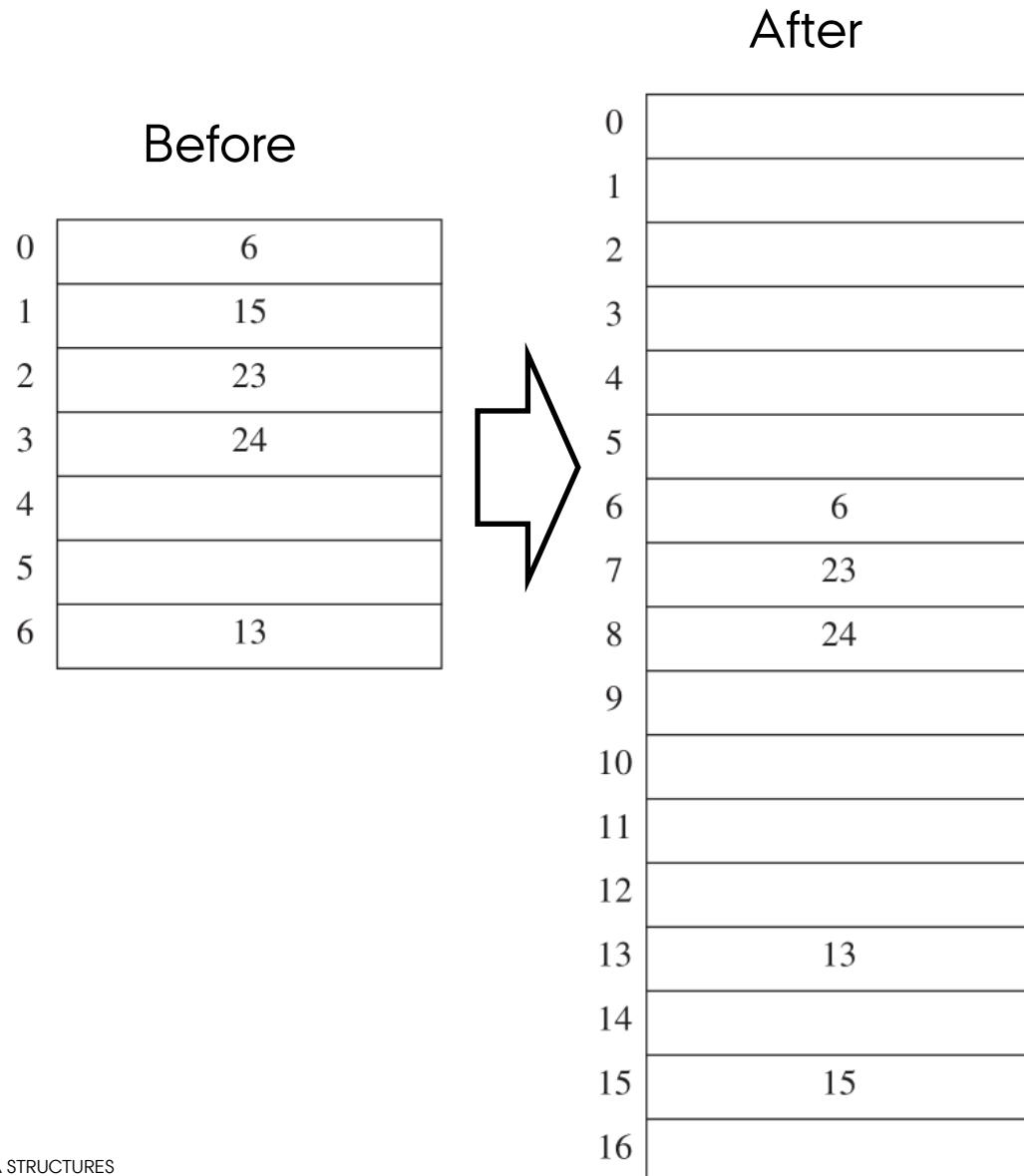
If  $\lambda \geq 0.5$ , then we need to rehash the table. This is mandatory for tables using quadratic probing.

For tables with chaining or linear probing, recommended to rehash when  $\lambda \geq 0.7$ .

# REHASHING

---

1. Allocate new table with  
Capacity = NextPrime(2\*OldCapacity)
2. For each element e in old table:  
insert e into new table, using new hash  
function:  $\text{hash}(e) = e \bmod \text{Capacity}$
3. Delete old table.



# HASH TABLE INTERFACE

QuadraticProbing.h

**Templated** hash table has:

- Constructor uses **prime** size  $M$
- Internal data structure as a **vector of entries**
- Each entry holds a key and the current status
- The status can be **active** (actual element), **empty** (free) or **deleted** (lazy deletion).

```
10  int nextPrime( int n );
11
12  template <typename HashedObj>
13  class HashTable
14  {
15      public:
16      explicit HashTable( int size = 101 ) : array( nextPrime( size ) )
17      { makeEmpty( ); }
18      void makeEmpty()
19      {
20          currentSize = 0;
21          for( auto & entry : array )
22              entry.info = EMPTY;
23      }
24  >
25  >
26  >
27  >
28  >     bool contains( const HashedObj & x ) const ...
29  >     int size( ) ...
30  >     bool insert( const HashedObj & x ) ...
31  >     bool insert( HashedObj && x ) ...
32  >     bool remove( const HashedObj & x ) ...
33  >     enum EntryType { ACTIVE, EMPTY, DELETED };
34
35
36  private:
37      struct HashEntry
38      {
39          HashedObj element;
40          EntryType info;
41
42          HashEntry( const HashedObj & e = HashedObj{ }, EntryType i = EMPTY )
43              : element{ e }, info{ i } { }
44
45          HashEntry( HashedObj && e, EntryType i = EMPTY )
46              : element{ std::move( e ) }, info{ i } { }
47      };
48      vector<HashEntry> array;
49      int currentSize;
50
51      bool isActive( int currentPos ) const
52          { return array[ currentPos ].info == ACTIVE; }
53
54  >
55  >
56  >
57  >
58  >
59  >
60  >
61  >
62  >
63  >
64  >
65  >
66  >
67  >
68  >
69  >
70  >
71  >
72  >
73  >
74  >
75  >
76  >
77  >
78  >
79  >
80  >
81  >
82  >
83  >
84  >
85  >
86  >
87  >
88  >
89  >
90  >
91  >
92  >
93  >
94  >
95  >
96  >
97  >
98  >
99  >
100 >
101 >
102 >
103 >
104 >
105 >
106 >
107 >
108 >
109 >
110 >
111 >
112 >
113 >
114 >
115 >
116 >
117 >
118 >
119 >
120 >
121 >
122 >
123 >
124 >
125 >
126 >
127 >
128 >
129 >
130 >
131 >
132 >
133 >
134 >
135 >
136 >
137 >
138 >
139 >
140 >
141 >
142 >
143 >
144 >
145 >
146 >
147 >
148 >
149 >
150 >
151 >
152 >
153 >
154 >
155 >
156 >
157 >
158 >
159 >
160 >
161 >
```

# HASH TABLE IMPL

- The private `findPos(x)` function is used by the `contains` and `remove`, and tries **quadratic probing** by increasing the position until either  $x$  or `EMPTY` is found

QuadraticProbing.h

```
99  bool isActive( int currentPos ) const
100 | { return array[ currentPos ].info == ACTIVE; }
101 | int findPos( const HashedObj & x ) const
102 {
103     int offset = 1;
104     int currentPos = myhash( x );
105
106     while( array[ currentPos ].info != EMPTY &&
107           |   array[ currentPos ].element != x )
108     {
109         currentPos += offset; // Compute ith probe
110         offset += 2;
111         if( currentPos >= array.size( ) )
112             |   currentPos -= array.size( );
113     }
114
115     return currentPos;
116 }
117 int findPosToInsert( const HashedObj & x ) const
118 {
119     int offset = 1;
120     int currentPos = myhash( x );
121
122     // We wish to stop looping when one of the following conditions are met:
123     // 1. the element is empty
124     // 2. the element is marked as deleted and thus available for reuse, or
125     // 3. the element is active and matches the new element to be inserted
126     // (in which case no new insertion will be made)
127     while( ! (
128         |   (array[ currentPos ].info == EMPTY) // 1
129         || (array[ currentPos ].info == DELETED) // 2
130         || (array[ currentPos ].info == ACTIVE
131             && array[ currentPos ].element == x) // 3
132         ))
133     {
134         currentPos += offset; // Compute ith probe
135         offset += 2;
136         if( currentPos >= array.size( ) )
137             |   currentPos -= array.size( );
138     }
139
140     return currentPos;
141 }
```

# HASH TABLE IMPL

- The private `findPostoInsert(x)` function is similar to `findPos` but is used by the insert. It tries **quadratic probing** by increasing the position until either,  $x$  or empty, or a deleted, cell is found.
- DISCUSS: Inspect the code and compare `findPostoInsert` with `findPos`. What are the differences?

```
99  bool isActive( int currentPos ) const
100 | { return array[ currentPos ].info == ACTIVE; }
101 | int findPos( const HashedObj & x ) const
102 {
103     int offset = 1;
104     int currentPos = myhash( x );
105
106     while( array[ currentPos ].info != EMPTY &&
107           |   array[ currentPos ].element != x )
108     {
109         currentPos += offset; // Compute ith probe
110         offset += 2;
111         if( currentPos >= array.size( ) )
112             |   currentPos -= array.size( );
113     }
114
115     return currentPos;
116 }
117 int findPostoInsert( const HashedObj & x ) const
118 {
119     int offset = 1;
120     int currentPos = myhash( x );
121
122     // We wish to stop looping when one of the following conditions are met:
123     // 1. the element is empty
124     // 2. the element is marked as deleted and thus available for reuse, or
125     // 3. the element is active and matches the new element to be inserted
126     // (in which case no new insertion will be made)
127     while( ! (
128         |   (array[ currentPos ].info == EMPTY) // 1
129         || (array[ currentPos ].info == DELETED) // 2
130         || (array[ currentPos ].info == ACTIVE
131             && array[ currentPos ].element == x) // 3
132     ))
133     {
134         currentPos += offset; // Compute ith probe
135         offset += 2;
136         if( currentPos >= array.size( ) )
137             |   currentPos -= array.size( );
138     }
139
140     return currentPos;
}
```

QuadraticProbing.h

# HASH TABLE IMPL

---

- The `rehash()` function is similar to chaining, and again needs to **insert one by one**.

```

86 >     struct HashEntry...
97
98
99 >     vector<HashEntry> array;
100 >     int currentSize;
101 >     bool isActive( int currentPos ) const
102 |     { return array[ currentPos ].info == ACTIVE; }
103 >     int findPos( const HashedObj & x ) const...
104 >     int findPosToInsert( const HashedObj & x ) const...
105 >     void rehash( )
106 |
107 >     {
108 |         vector<HashEntry> oldArray = array;
109 |
110 |         // Create new double-sized, empty table
111 |         array.resize( nextPrime( 2 * oldArray.size( ) ) );
112 |         for( auto & entry : array )
113 |             entry.info = EMPTY;
114 |
115 |         // Copy table over
116 |         currentSize = 0;
117 |         for( auto & entry : oldArray )
118 |             if( entry.info == ACTIVE )
119 |                 insert( std::move( entry.element ) );
120 |
121 >     size_t myhash( const HashedObj & x ) const
122 |
123 >     {
124 |         static hash<HashedObj> hf;
125 |         return hf( x ) % array.size( );
126 |

```

# HASH TABLE IMPL

- Searching, inserting and removing will first **try** to find the element using quadratic probing
- Insertion needs now needs to handle **deleted** slots
- The hash table is increased if the current **occupation** takes half of the table to **control** the load factor.

QuadraticProbing.h

```
24 > bool contains( const HashedObj & x ) const
25 {
26     return isActive( findPos( x ) );
27 }
28 > int size( ) ...
29 > bool insert( const HashedObj & x )
30 {
31     // Insert x as active
32     int currentPos = findPosToInsert( x );
33     if( isActive( currentPos ) ){
34         assert(array[ currentPos ].element == x);
35         return false;
36     }
37
38     if( array[ currentPos ].info != DELETED )
39         ++currentSize;
40
41     array[ currentPos ] = std::move( x );
42     array[ currentPos ].info = ACTIVE;
43
44     // Rehash; see Section 5.5
45     if( currentSize > array.size( ) / 2 )
46         rehash();
47
48     return true;
49 }
50 > bool insert( HashedObj && x ) ...
51 > bool remove( const HashedObj & x )
52 {
53     int currentPos = findPos( x );
54     if( !isActive( currentPos ) )
55         return false;
56
57     array[ currentPos ].info = DELETED;
58     return true;
59 }
60
61 enum EntryType { ACTIVE, EMPTY, DELETED };
```

# OPTIONAL - NOTES ON THEOREM 5.1

---

## Theorem 5.1

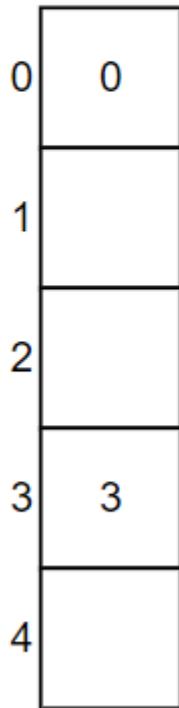
If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

# OPTIONAL - NOTES ON THEOREM 5.1

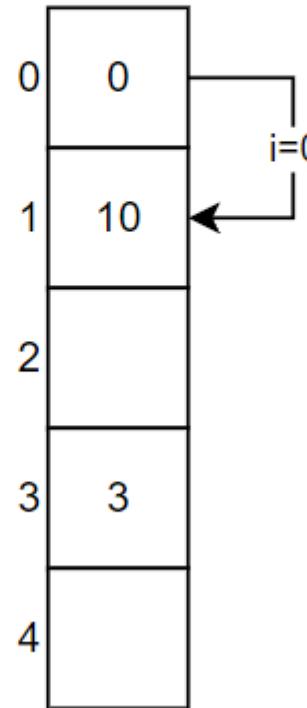
---

First we make some attempts at inserting using quadratic probing, using the conditions of the theorem:

Before:



After inserting  $10 \% 5 = 0$



# OPTIONAL – NOTES ON THEOREM 5.1

---

Maybe we can change the configuration to occupy the first step of the quadratic probing:

Before:

0	0
1	1
2	
3	
4	

After inserting  $10 \% 5 = 0$

0	0	i=0
1	1	i=1
2	10	
3		
4		

Note that if we decided to occupy the cell indexed by 3, we would have to break the conditions of the theorem, as then the table would not be half empty.

# OPTIONAL - NOTES ON THEOREM 5.1

---

## Theorem 5.1

If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

That's the same as showing that, when inserting an element, that there will be at least one jump leading to an empty cell.

If we show that each of the  $N$  initial positions will always lead to a distinct cell, then by necessity there must be  $N$  distinct cells in the table.

If we take  $N=\text{ceil}(\text{TableSize}/2)$ , then we have demonstrated the theorem, because at least one of those distinct cells will be empty!

# OPTIONAL – NOTES ON THEOREM 5.1

---

## Theorem 5.1

If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

### Proof

Let the table size,  $\text{TableSize}$ , be an (odd) prime greater than 3. We show that the first  $\lceil \text{TableSize}/2 \rceil$  alternative locations (including the initial location  $h_0(x)$ ) are all distinct. Two of these locations are  $h(x) + i^2 \pmod{\text{TableSize}}$  and  $h(x) + j^2 \pmod{\text{TableSize}}$ , where  $0 \leq i, j \leq \lfloor \text{TableSize}/2 \rfloor$ . Suppose, for the sake of contradiction, that these locations are the same, but  $i \neq j$ . Then

$$\begin{aligned} h(x) + i^2 &= h(x) + j^2 \pmod{\text{TableSize}} \\ i^2 &= j^2 \pmod{\text{TableSize}} \\ i^2 - j^2 &= 0 \pmod{\text{TableSize}} \\ (i - j)(i + j) &= 0 \pmod{\text{TableSize}} \end{aligned}$$

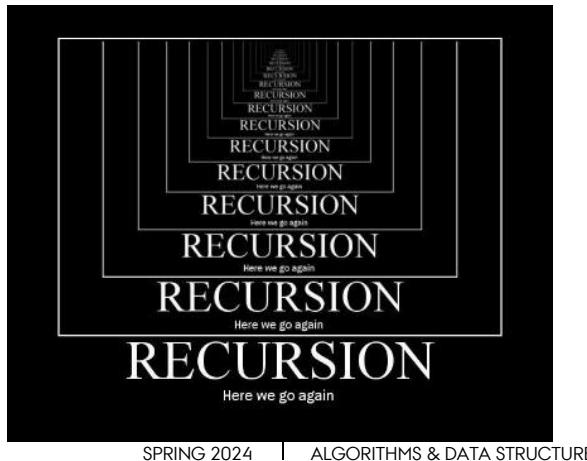
Since  $\text{TableSize}$  is prime, it follows that either  $(i - j)$  or  $(i + j)$  is equal to 0  $(\bmod \text{TableSize})$ . Since  $i$  and  $j$  are distinct, the first option is not possible. Since  $0 \leq i, j \leq \lfloor \text{TableSize}/2 \rfloor$ , the second option is also impossible. Thus, the first  $\lceil \text{TableSize}/2 \rceil$  alternative locations are distinct. If at most  $\lfloor \text{TableSize}/2 \rfloor$  positions are taken, then an empty spot can always be found.

# RECURSION

# RECUSION

---

- **Recursion** is a problem-solving technique that can be used when a problem can be solved by solving one or more **smaller versions of the problem itself**
- “Recursive” means “**self-referencing**”
- A **function is recursive** if it calls itself directly or indirectly
- A **data structure is recursive** if it has fields of its own type (e.g. linked list, tree)



# RECURSION

---

- A simple **recursive function** to sum first  $x$  natural numbers
- **Base case** solved without recursion (without calls to self)
- **Recursive case** solved recursively (calling self on smaller input) with **progress** toward base case

$$\begin{aligned} f(4) &= 4 + f(3) \\ &= 4 + 3 + f(2) \\ &= 4 + 3 + 2 + f(1) \\ &= 4 + 3 + 2 + 1 + f(0) \\ &= 5 + 4 + 3 + 2 + 1 + 0 \\ &= 10 \end{aligned}$$

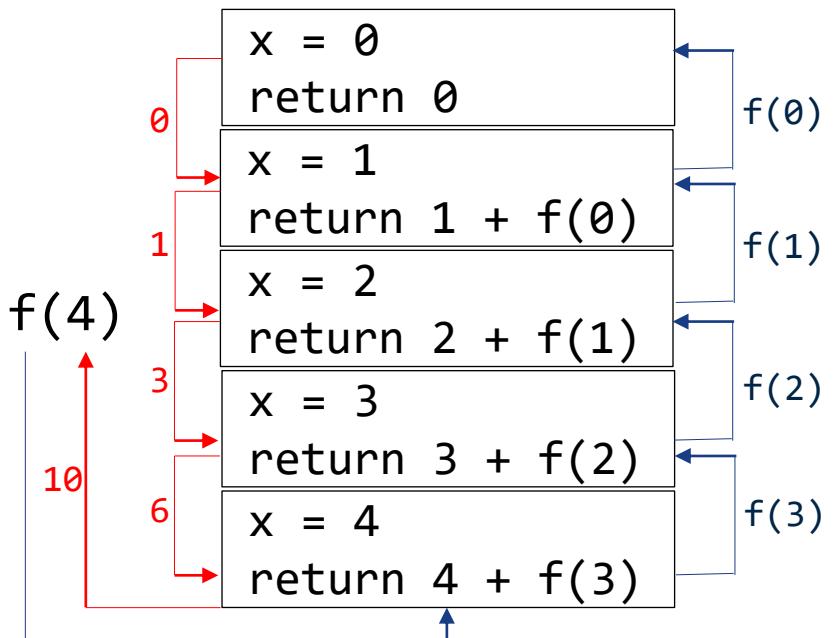
recursive cases      base case

```
1 #include <iostream>
2 #include <stdio.h>
3
4 int f(int x)
5 {
6     if (x == 0)           base case
7         return 0;
8     else                 recursive case
9         return x + f(x - 1);
10 }
11
12 int main()
13 {
14     unsigned int i;
15
16     while (true) {
17         std::cin >> i;
18         printf("f(%d)=%d\n", i, f(i));
19     }
20 }
```

```
hejersbo@D42857:~/test/sum$ ./rec
0
f(0)=0
2
f(2)=3
4
f(4)=10
10
f(10)=55
1000
f(1000)=500500
```

# RECURSION

- A simple **recursive function** to sum first  $x$  natural numbers
- Function calls uses **stack of activation frames (stack frames)**



```
(gdb) bt
#0  f (x=0) at rec.cpp:7
#1  0x0000555555551d2 in f (x=1) at rec.cpp:9
#2  0x0000555555551d2 in f (x=2) at rec.cpp:9
#3  0x0000555555551d2 in f (x=3) at rec.cpp:9
#4  0x0000555555551d2 in f (x=4) at rec.cpp:9
#5  0x0000555555551d2 in f (x=5) at rec.cpp:9
#6  0x0000555555551d2 in f (x=6) at rec.cpp:9
#7  0x0000555555551d2 in f (x=7) at rec.cpp:9
#8  0x0000555555551d2 in f (x=8) at rec.cpp:9
#9  0x0000555555551d2 in f (x=9) at rec.cpp:9
#10 0x0000555555551d2 in f (x=10) at rec.cpp:9
#11 0x000055555555211 in main () at rec.cpp:18
```

← call stack

```
Stack frame at 0x7fffffff840:
rip = 0x555555551be in f (rec.cpp:7); saved rip = 0x555555551d2
called by frame at 0x7fffffff860
source language c++.
Arglist at 0x7fffffff818, args: x=0
Locals at 0x7fffffff818, Previous frame's sp is 0x7fffffff840
Saved registers:
rbp at 0x7fffffff830, rip at 0x7fffffff838
```

```
(gdb) info frame 1
Stack frame at 0x7fffffff860:
rip = 0x555555551d2 in f (rec.cpp:9); saved rip = 0x555555551d2
called by frame at 0x7fffffff880, caller of frame at 0x7fffffff840
source language c++.
Arglist at 0x7fffffff838, args: x=1
Locals at 0x7fffffff838, Previous frame's sp is 0x7fffffff860
Saved registers:
rbp at 0x7fffffff850, rip at 0x7fffffff858
```

stack frame details



# RECURSION

---

- A simple **recursive function** to sum first  $x$  natural numbers
- Stack size is limited, so too many recursively nested calls can result in **stack overflow**
- How can we be sure this segmentation fault is actually a stack overflow ???
- **Use debugger!!!**

```
hejersbo@D42857:~/test/sum$ ./rec
0
f(0)=0
2
f(2)=3
4
f(4)=10
10
f(10)=55
1000
f(1000)=500500
1000000
Segmentation fault
hejersbo@D42857:~/test/sum$
```

Stack overflow ???



# RECURSION

---

- A simple **recursive function** to sum first  $x$  natural numbers
- Stack size is limited, so too many recursively nested calls can result in **stack overflow**
- Stack size used is ~ difference between stack pointers in top and bottom stack frames.
- `ulimit -s` returns stack size limit (8 MB)

```
(gdb) bt 10
#0  0x0000555555551cd in f (x=738037) at rec.cpp:9
#1  0x0000555555551d2 in f (x=738038) at rec.cpp:9
#2  0x0000555555551d2 in f (x=738039) at rec.cpp:9
#3  0x0000555555551d2 in f (x=738040) at rec.cpp:9
#4  0x0000555555551d2 in f (x=738041) at rec.cpp:9
#5  0x0000555555551d2 in f (x=738042) at rec.cpp:9
#6  0x0000555555551d2 in f (x=738043) at rec.cpp:9
#7  0x0000555555551d2 in f (x=738044) at rec.cpp:9
#8  0x0000555555551d2 in f (x=738045) at rec.cpp:9
#9  0x0000555555551d2 in f (x=738046) at rec.cpp:9
(More stack frames follow...)
(gdb) bt -10
#261955 0x0000555555551d2 in f (x=999992) at rec.cpp:9
#261956 0x0000555555551d2 in f (x=999993) at rec.cpp:9
#261957 0x0000555555551d2 in f (x=999994) at rec.cpp:9
#261958 0x0000555555551d2 in f (x=999995) at rec.cpp:9
#261959 0x0000555555551d2 in f (x=999996) at rec.cpp:9
#261960 0x0000555555551d2 in f (x=999997) at rec.cpp:9
#261961 0x0000555555551d2 in f (x=999998) at rec.cpp:9
#261962 0x0000555555551d2 in f (x=999999) at rec.cpp:9
#261963 0x0000555555551d2 in f (x=1000000) at rec.cpp:9
#261964 0x000055555555211 in main () at rec.cpp:18
```

```
(gdb) frame 0
#0  0x0000555555551cd in f (x=738037) at rec.cpp:9
9          return x + f(x - 1);
(gdb) set $topsp = $sp
(gdb) frame 261964
#261964 0x000055555555211 in main () at rec.cpp:18
18          printf("f(%d)=%d\n", i, f(i));
(gdb) print $sp - $topsp
$1 = 8382848
(gdb) shell ulimit -s
8192
```

# RECUSION

---

- A simple **recursive function** to sum first  $x$  natural numbers
- Are there better ways than using recursion for this sum?
- Formula:  $\sum_{i=1}^N i = \frac{N(N+1)}{2}$  (Weiss, p.5)
- Function  $g$  has  $O(1)$  complexity whereas  $f$  is  $O(n)$  where  $n$  is size of input integer.
- And  $f$  still gives stack overflow

```
1 #include <stdio.h>
2 #include <time.h>
3 #include <iostream>
4 #include "StopwatchGeneric.h"
5
6 int f(int x)
7 {
8     if (x == 0)
9         return 0;
10    else
11        return x + f(x - 1);
12 }
13
14 int g(int x)
15 {
16     return (x * (x + 1)) / 2;
17 }
```

```
18 hejersbo@D42857:~/test/sum$ ./rec
10
g(10)=55      8ticks 1us
f(10)=55      2ticks 1us
100
g(100)=5050   3ticks 1us
f(100)=5050   3ticks 2us
1000
g(1000)=500500 2ticks 1us
f(1000)=500500 28ticks 27us
10000
g(10000)=50005000 1ticks 0us
f(10000)=50005000 121ticks 121us
100000
g(100000)=705082704 3ticks 1us
f(100000)=705082704 1357ticks 1357us
500000
g(500000)=446198416 2ticks 0us
Segmentation fault
```

# REFLECTION

---

Download the code for this week in BS, and debug one of the provided sum codes in your favorite IDE.

Then:

Identify the different stack calls.

What causes the call stack to increase?

What causes it to decrease?

# INDUCTION

---

- Recursion goes hand-in-hand with the proof method **induction** (to prove correctness)
- E.g. is formula:  $\sum_{i=1}^N i = \frac{N(N+1)}{2}$  correct (our simple function g)?
- A proof by induction has a **base case**, and an **inductive case**
- **First step** is proving the theorem for the **base case**
- **Second step**, assume the theorem is true for all values up to some limit  $k$ , and then prove using this assumption that the theorem is true for value  $k+1$ . Since  $k$  can be any number, this proves the theorem.

# INDUCTION

---

- Theorem:  $\sum_{i=1}^N i = \frac{N(N+1)}{2}$
- Base Case: For  $N = 1$ :  $\sum_{i=1}^1 i = \frac{1(1+1)}{2} \Leftrightarrow 1 = 1$
- Inductive Case: Assume that for a particular  $k$ :  $\sum_{i=1}^k i = \frac{k(k+1)}{2}$  and prove for  $k+1$ :

$$\begin{aligned}\sum_{i=1}^{k+1} i &= \frac{k(k+1)}{2} + (k+1) \quad (\text{inductive hypothesis}) \\ &= \frac{k(k+1)+2(k+1)}{2} \\ &= \frac{(k+1)(k+2)}{2} \\ &= \frac{(k+1)((k+1)+1)}{2}\end{aligned}$$

# TAIL RECURSION

---

- The **recursive *sum*** function builds stack and needs to perform ‘+’ op when unwinding stack after base case is reached
- The **tail recursive *sumTail*** function has nothing left to do after the recursion reaches base case.
- **Tail recursive =** The last operation of a function is a recursive call
- Tail recursion **can be optimized away** by executing call in current stack frame and returning its result rather than creating a new stack frame

Sum the elements of an integer array

```
1 int sum(int *ar, int size)
2 {
3     int temp = 0;
4
5     if (size == 0)
6         return 0;
7
8     temp = ar[0] + sum(ar + 1, size - 1);
9     return temp;
10}
11
12 int sumTail(int *ar, int size, int sum)
13 {
14     if (size == 0)
15         return sum;
16
17     return sumTail(ar + 1, size - 1, sum + ar[0]);
18}
```

# DON'T DUPLICATE WORK ...

---

- Fibonacci numbers:  $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$
  - This is an inefficient implementation since computation of the same sub result is duplicated:
  - $\text{fib}(6) = \text{fib}(5) + \text{fib}(4)$   
 $\text{fib}(5) = \text{fib}(4) + \text{fib}(3)$   
 **$\text{fib}(4) = \text{fib}(3) + \text{fib}(2)$**   
 **$\text{fib}(3) = \text{fib}(2) + \text{fib}(1)$**   
 **$\text{fib}(4) = \text{fib}(3) + \text{fib}(2)$**   
 **$\text{fib}(3) = \text{fib}(2) + \text{fib}(1)$**   
...  
• Time complexity is  $O(2^n)$
- 

```
1 unsigned int fib(unsigned int n)
2 {
3     if (n == 0 || n == 1)
4         return n;
5     else
6         return fib(n - 1) + fib(n - 2);
7 }
```

# REFLECTION

---

Why is the time complexity of the fib function  $O(2^n)$ ?

```
1 unsigned int fib(unsigned int n)
2 {
3     if (n == 0 || n == 1)
4         return n;
5     else
6         return fib(n - 1) + fib(n - 2);
7 }
```

# DON'T DUPLICATE WORK ...

---

- Fibonacci numbers:  $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$
- A tail recursive implementation:
- Time complexity is  $\mathbf{O}(n)$
- $F_n = \text{fibTail}(n, 0, 1)$



```
1 unsigned int fibTail(unsigned int n, int a, int b)
2 {
3     if (n == 0)
4         return a;
5     if (n == 1)
6         return b;
7
8     return fibTail(n - 1, b, a + b);
9 }
```

# REFLECTION

---

- Use pen and paper to execute the previous fibTail function and convince yourself that it is equivalent to the non-tail-recursive fib function.

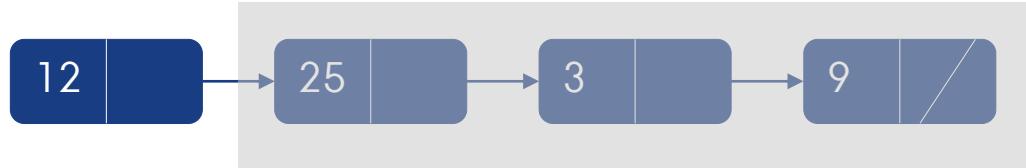
Fibonacci sequence.  
Hint: expand the function for  $n=7$  until  $n=1$ , and compare the  $a$  and  $b$  parameters with the

# RECURSIVE DATA STRUCTURES

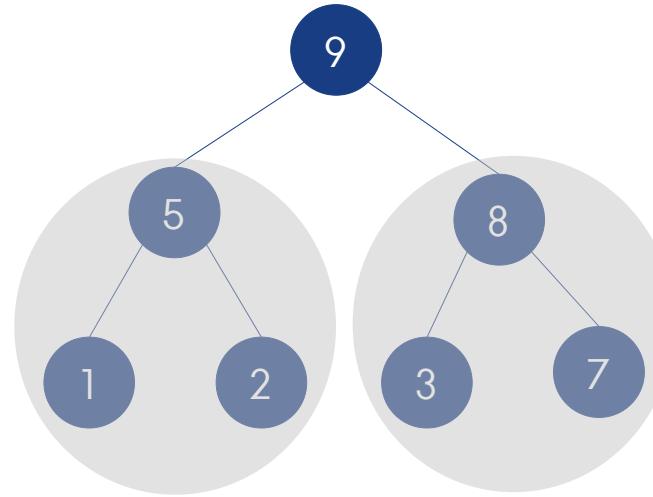
---

- Many data structures are inherently recursive and allow for recursive implementations of e.g. searching or sorting
- E.g. a linked list is a node followed by a list:

```
struct Node {  
    int value;  
    Node *next;  
}
```
- Recursive solutions **are typically not as efficient as non-recursive** (e.g. iterative) solutions, but they often provide for **simpler (and hence less error-prone) implementations**



List is a node followed by a list

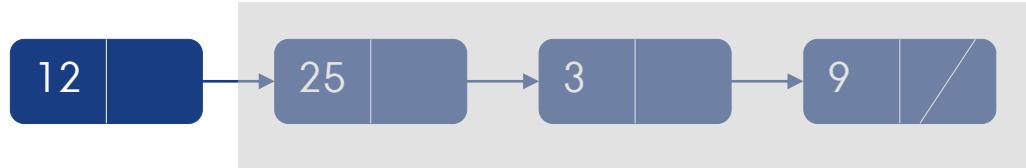


Binary tree is a node with left and right sub trees

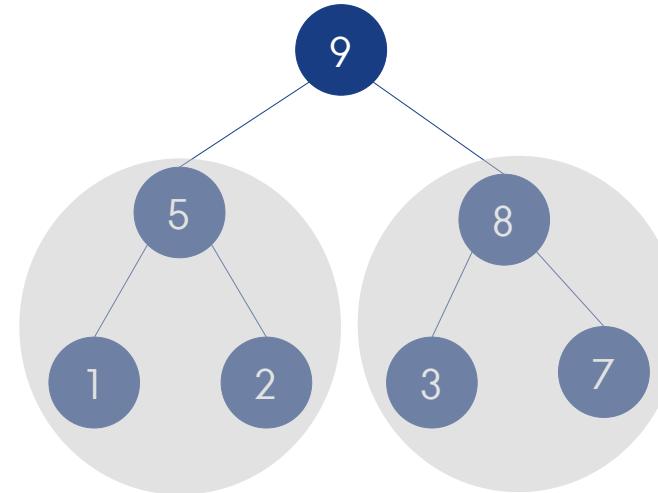
# RECURSIVE DATA STRUCTURES

---

- Recursive implementations that either are not tail recursive or where the compiler cannot optimize will usually be slower than iterative implementations
- A function call takes additional time due to register saving and stack frame creation
- A function call requires more expensive branching instruction in assembly
- Recursive implementation typically requires more memory (stack)



List is a node followed by a list



Binary tree is a node with left and right sub trees

# REFLECTION

---

- Draft a recursive implementation of the search algorithm in a linked list.

# SORTING ARRAYS

# AGENDA

---

- The array sorting problem
  - MergeSort and its complexity
  - QuickSort and its complexity
  - A lower bound for comparison-based sorting

# THE ARRAY SORTING PROBLEM

---

## Definition:

Given an array  $\langle A_0, A_1, \dots, A_{N-1} \rangle$  of  $N$  elements, compute a permutation of the input  $\langle A'_0, A'_1, \dots, A'_{N-1} \rangle$  such that:

$$A'_0 \leq A'_1 \leq \dots \leq A'_{N-1}.$$

It is a **fundamental** problem in algorithm design, for which there are many known algorithms (Wikipedia lists > 45). It is also easy to characterize **average-case** instances (random).

Sorting occurs frequently as an **intermediate step**, and different approaches illustrate different **algorithm design** strategies.

# ARRAY SORTING IN C++

---

The C++ STL already provides sorting algorithms through the interface below:

```
void sort(Iterator begin, Iterator end);  
void sort(Iterator begin, Iterator end, Comparator cmp);
```

It assumes that the iterators have **random access**:

```
std::sort( v.begin(), v.end() );  
std::sort( v.begin(), v.end(), less<int>{ } );  
std::sort( v.begin(), v.begin() + (v.end() - v.begin()) / 2 );
```

# STL EXAMPLE

---

```
1 #include <algorithm>
2 #include <iostream>
3 #include <vector>
4
5 using namespace std;
6 template<class T>
7 void show(vector<T> v) {
8     cout << "[";
9     for (T i:v)
10        cout << v[i] << ", ";
11    cout << "]";
12 }
13
14 int main() {
15     vector<int> a = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
16     cout << "The vector before sorting is: \n";
17     show(a);
18
19     sort(a.begin(), a.end());
20
21     cout << "\n \n The vector after sorting is: \n";
22     show(a);
23
24     return 0;
25 }
```

stlsort.cpp

The vector before sorting is:

[5, 7, 2, 0, 3, 4, 9, 6, 8, 1, ]

The vector after sorting is:

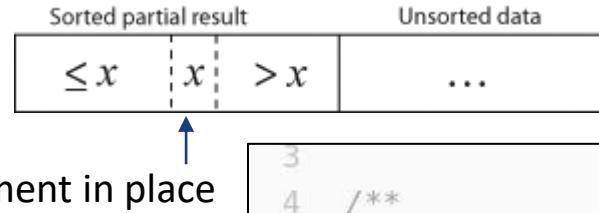
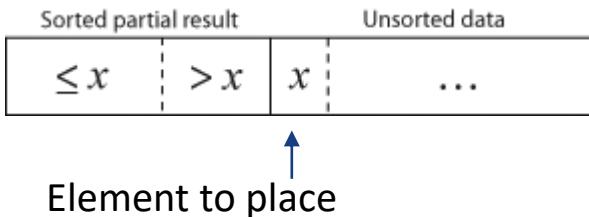
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ]

STL's sort implements a **divide-and-conquer** algorithm for breaking the problem in smaller pieces until the subproblems can be solved efficiently by an **iterative** sort. See exercise



# INSERTION SORT IN C++

The array[0..p-1] is sorted. Find the place for p. Do until end of array



insertion\_sort.h

```
3  /**
4   * Simple insertion sort.
5   */
6  template <typename Comparable>
7  void insertionSort(vector<Comparable> &a) {
8      for (int p = 1; p < a.size(); ++p) {
9          Comparable tmp = std::move(a[p]);
10
11         int j;
12         for (j = p; j > 0 && tmp < a[j - 1]; --j) {
13             a[j] = std::move(a[j - 1]);
14         }
15     }
16     a[j] = std::move(tmp);
17 }
18 }
19 }
```

# ARRAY SORTING IN C++

---

A version using iterators close to C++ standard is below.

The iterator version is actually **less readable**, so we will stick to the explicit versions using Comparable templates.

Notice the instantiation of the Comparator object.

```
20  /*
21   * This is the more public version of insertion sort.
22   * It requires a pair of iterators and a comparison
23   * function object.
24   */
25  template <typename Iterator, typename Comparator>
26  void insertionSort(const Iterator& begin, const Iterator& end, Comparator less){
27      if (begin == end)
28          return;
29
30      for (Iterator j, p = begin + 1; p != end; ++p) {
31          auto tmp = std::move(*p);
32          for (j = p; j != begin && less(tmp, *(j - 1)); --j)
33              *j = std::move(*(j - 1));
34          *j = std::move(tmp);
35      }
36  }
37
38  /*
39   * The two-parameter version calls the three parameter version, using C++11.
40   */
41  template <typename Iterator>
42  void insertionSort(const Iterator& begin, const Iterator& end) {
43      insertionSort(begin, end, less<decltype(*begin)>{ });
44 }
```

insertion\_sort.h

# REFLECTION

---

1. Use a drawing tool to run the insertion sort on the array below. Draw the array after each iteration.

34	8	64	51	32	21
----	---	----	----	----	----

2. What are the worst case and best case inputs to the insertion sort?
3. For the worst case, what is the complexity of the insertion sort? In the best case, what is the complexity of the insertion sort?



Photo by [Anthony Tran](#) on [Unsplash](#)

# AGENDA

---

- ✓ The array sorting problem
- MergeSort and its complexity
- QuickSort and its complexity
- A lower bound for comparison-based sorting

# REFLECTION

---

If I give you the following two sorted arrays,

A1 = [1, 13, 24, 26]

A2 = [2, 15, 27, 38]

how would you merge them into a new sorted array?

Res = [ \_\_, \_\_, \_\_, \_\_, \_\_, \_\_, \_\_, \_\_ ]

1. Draw the resulting sorted array.
2. Draft the algorithm that performs the merge of any two sorted arrays to a new sorted array  
(hint: the complexity should be O(N))

# DIVIDE-AND-CONQUER SOLUTIONS

---

→ *To break the quadratic bound, we need a new way to think about array sorting.*

We can think of breaking the problem in smaller ones:

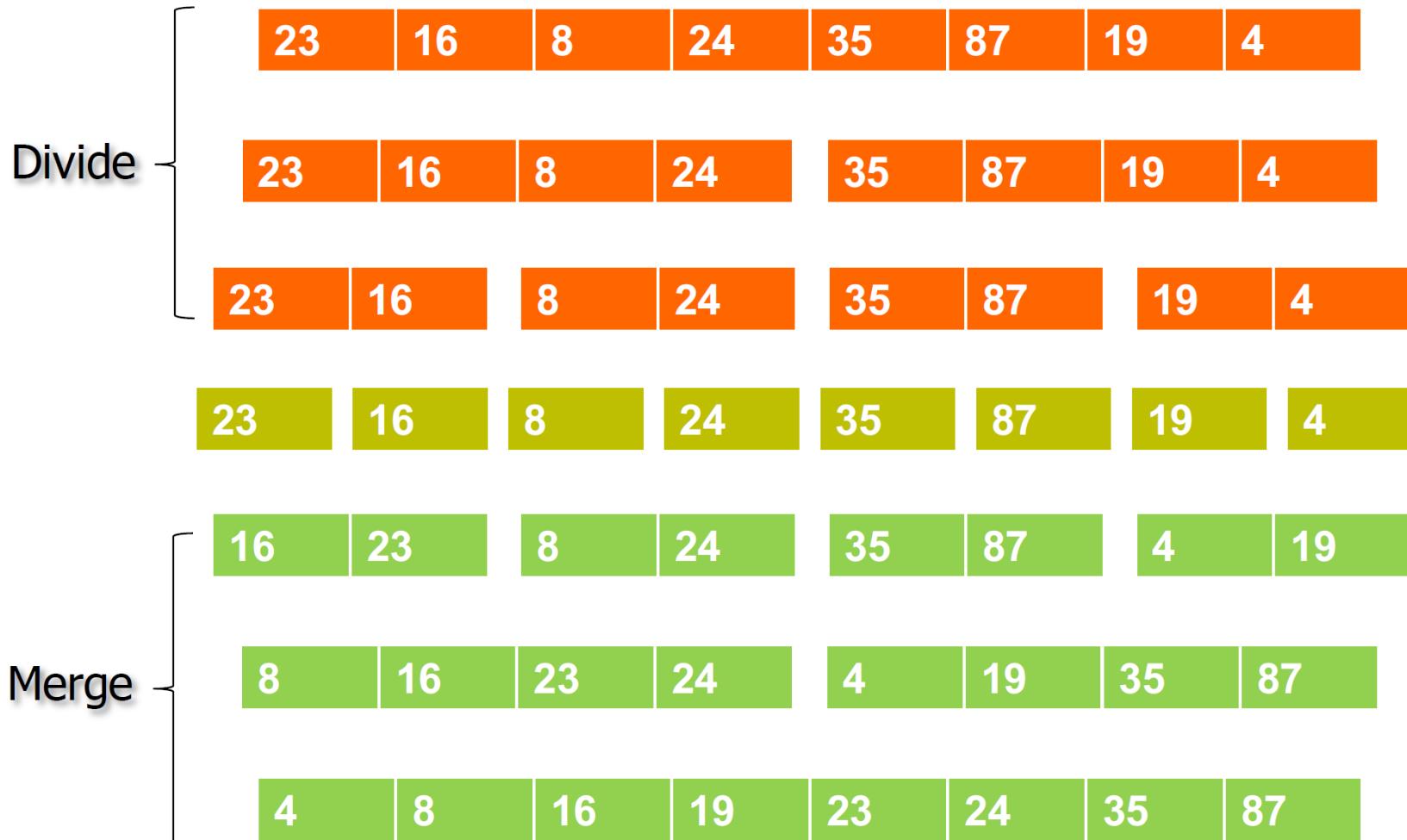
1. **Split** the array in two parts
2. **Sort** each part independently
3. **Merge** the sorted parts preserving the order

This intuition gives another algorithm: **MergeSort!**

*How to merge two sorted arrays?*

# GENERAL PRINCIPLE

---



# MERGESORT

---

Merging two sorted arrays requires **temporary space**. This version merges the sorted subarrays  $A[\text{leftPos}..\text{leftEnd}]$  and  $A[\text{rightPos}..\text{rightEnd}]$ .

They can have **different sizes**. After the main loop, the remaining loops handle the rest.

merge\_sort.h

```
1 #ifndef MERGE_SORT_H
2 #define MERGE_SORT_H
3
4 /**
5  * Internal method that merges two sorted halves of a subarray.
6  * a is an array of Comparable items.
7  * tmp is an array to place the merged result.
8  * leftPos is the left-most index of the subarray.
9  * rightPos is the index of the start of the second half.
10 * rightEnd is the right-most index of the subarray.
11 */
12 template <typename Comparable>
13 void merge(vector<Comparable>& a, vector<Comparable>& tmp,
14            int leftPos, int rightPos, int rightEnd) {
15     int leftEnd = rightPos - 1;
16     int pos = leftPos;
17     int numElements = rightEnd - leftPos + 1;
18
19     // Main loop
20     while (leftPos <= leftEnd && rightPos <= rightEnd) {
21         if (a[leftPos] <= a[rightPos]) {
22             tmp[pos++] = std::move(a[leftPos++]);
23         } else {
24             tmp[pos++] = std::move(a[rightPos++]);
25         }
26     }
27
28     while (leftPos <= leftEnd) { // Copy rest of first half
29         tmp[pos++] = std::move(a[leftPos++]);
30     }
31
32     while (rightPos <= rightEnd) { // Copy rest of right half
33         tmp[pos++] = std::move(a[rightPos++]);
34     }
35
36     // Copy tmp back
37     for (int i = 0; i < numElements; ++i, --rightEnd) {
38         a[rightEnd] = std::move(tmp[rightEnd]);
39     }
40 }
```

# MERGESORT

Merging two sorted arrays requires temporary space. This version merges the sorted subarrays  $A[\text{leftPos}..\text{leftEnd}]$  and  $A[\text{rightPos}..\text{rightEnd}]$ .

	Left				Right			
A	2	3	4	5	1	2	3	6
tmp	1	2	2	3	3	4	5	6

```
1 #ifndef MERGE_SORT_H
2 #define MERGE_SORT_H
3
4 /**
5  * Internal method that merges two sorted halves of a subarray.
6  * a is an array of Comparable items.
7  * tmp is an array to place the merged result.
8  * leftPos is the left-most index of the subarray.
9  * rightPos is the index of the start of the second half.
10 * rightEnd is the right-most index of the subarray.
11 */
12 template <typename Comparable>
13 void merge(vector<Comparable>& a, vector<Comparable>& tmp,
14            int leftPos, int rightPos, int rightEnd) {
15     int leftEnd = rightPos - 1;
16     int pos = leftPos;
17     int numElements = rightEnd - leftPos + 1;
18
19     // Main loop
20     while (leftPos <= leftEnd && rightPos <= rightEnd) {
21         if (a[leftPos] <= a[rightPos]) {
22             tmp[pos++] = std::move(a[leftPos++]);
23         } else {
24             tmp[pos++] = std::move(a[rightPos++]);
25         }
26     }
27
28     while (leftPos <= leftEnd) { // Copy rest of first half
29         tmp[pos++] = std::move(a[leftPos++]);
30     }
31
32     while (rightPos <= rightEnd) { // Copy rest of right half
33         tmp[pos++] = std::move(a[rightPos++]);
34     }
35
36     // Copy tmp back
37     for (int i = 0; i < numElements; ++i, --rightEnd) {
38         a[rightEnd] = std::move(tmp[rightEnd]);
39     }
40 }
```

# MERGESORT

---

The recursive algorithm applies the divide and conquer technique.

```
42  /**
43   * Internal method that makes recursive calls.
44   * a is an array of Comparable items.
45   * tmp is an array to place the merged result.
46   * left is the left-most index of the subarray.
47   * right is the right-most index of the subarray.
48   */
49  template <typename Comparable>
50  void mergeSort(vector<Comparable>& a, vector<Comparable>& tmp,
51                 int left, int right) {
52     if (left < right) {
53         int center = (left + right) / 2;
54         mergeSort(a, tmp, left, center);
55         mergeSort(a, tmp, center + 1, right);
56         merge(a, tmp, left, center + 1, right);
57     }
58 }
59
60 /**
61  * Mergesort algorithm (external).
62  */
63 template <typename Comparable>
64 void mergeSort(vector<Comparable>& a) {
65     vector<Comparable> tmp(a.size());
66     mergeSort(a, tmp, 0, a.size() - 1);
67 }
```

merge\_sort.h

# REFLECTION

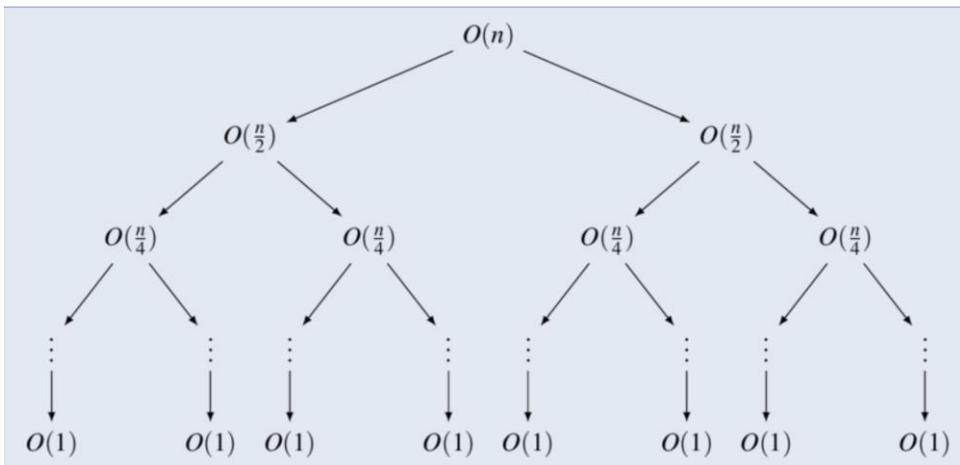
---

1. Follow the code in the previous two slides to sort the following array. Use a drawing tool, and sketch the different function calls in a tree fashion.  
 $A=[24, 13, 26, 1, 2, 27, 38, 15]$
2. How many steps were carried out (hint: count the number of levels in the tree, and count the number of steps in each function call)

# MERGESORT

---

**Time complexity for  $O(\log(n))$  levels  
each requiring linear  $O(n)$  work for  
merging =>  $O(n * \log(n))$**



```
42  /**
43   * Internal method that makes recursive calls.
44   * a is an array of Comparable items.
45   * tmp is an array to place the merged result.
46   * left is the left-most index of the subarray.
47   * right is the right-most index of the subarray.
48   */
49  template <typename Comparable>
50  void mergeSort(vector<Comparable>& a, vector<Comparable>& tmp,
51                  int left, int right) {
52      if (left < right) {
53          int center = (left + right) / 2;
54          mergeSort(a, tmp, left, center);
55          mergeSort(a, tmp, center + 1, right);
56          merge(a, tmp, left, center + 1, right);
57      }
58  }
59 /**
60  * Mergesort algorithm (external).
61  */
62 template <typename Comparable>
63 void mergeSort(vector<Comparable>& a) {
64     vector<Comparable> tmp(a.size());
65     mergeSort(a, tmp, 0, a.size() - 1);
66 }
67 }
```

merge\_sort.h

# AGENDA

---

- ✓ The array sorting problem
- ✓ MergeSort and its complexity
- QuickSort and its complexity
- A lower bound for comparison-based sorting

# REFLECTION

---

Consider the following array:

$$A = [ 1, 13, 2, 15, 26, 27, 38, 24 ]$$

If the array were to be sorted, is it possible that the element 15 ends up in a different position than it already holds (position 3)?

In your working group, each person should sort the array manually, and then compare where 15 ends up. Why did you all get the same position (3) for 15?

# QUICKSORT

---

→ *To perform sorting **inplace**, we need a new way to think!*

We can think of breaking the problem in smaller ones:

1. **Split** the array in three parts
2. **Aproximately** sort by constructing a partition  
$$A[l, \dots, p - 1] \leq A[p] \leq A[p+1, \dots, r].$$
3. **Sort** each of the subsections.

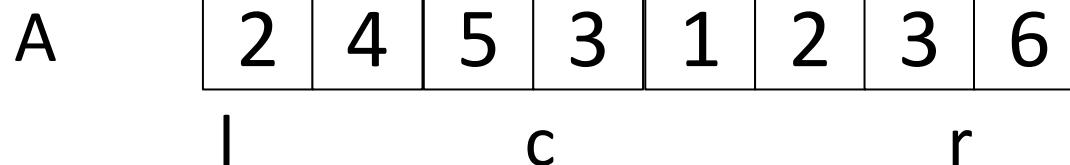
This intuition gives another algorithm: **QuickSort**!

*How to create the partition?*

# QUICKSORT

---

We elect the middle element as the pivot. This is **median of 3 partitioning**.



After sorting the three elements:



And moving the pivot:



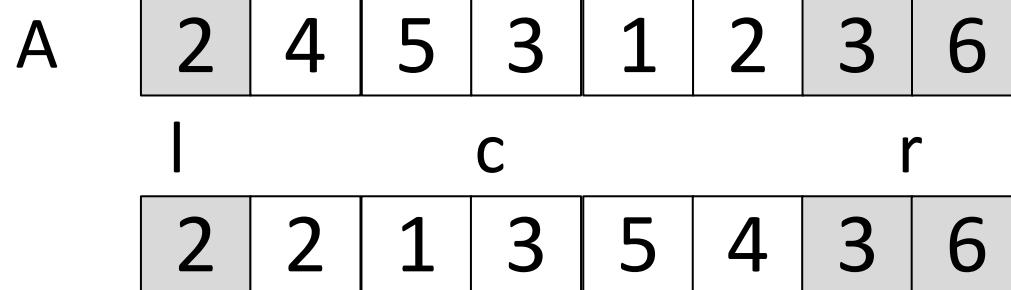
```
1 #ifndef QUICK_SORT_H
2 #define QUICK_SORT_H
3
4 /**
5  * Order left, center, and right and hide the pivot.
6  * Then compute partition, restore the pivot and return its position.
7 */
8 template <typename Comparable>
9 int partition(vector<Comparable>& a, int left, int right) {
10     int center = (left + right) / 2;
11
12     if (a[center] < a[left])
13         std::swap(a[left], a[center]);
14     if (a[right] < a[left])
15         std::swap(a[left], a[right]);
16     if (a[right] < a[center])
17         std::swap(a[center], a[right]);
18
19     // Place pivot at position right - 1
20     std::swap(a[center], a[right - 1]);
21
22     // Now the partitioning
23     Comparable& pivot = a[right - 1];
24     int i = left, j = right - 1;
25     do {
26         while (a[++i] < pivot);
27         while (pivot < a[--j]);
28         if (i < j) {
29             std::swap(a[i], a[j]);
30         }
31     } while (i < j);
32
33     std::swap(a[i], a[right - 1]); // Restore pivot
34     return i;
35 }
```

quick\_sort.h

# QUICKSORT

---

After sorting the two halves around the pivot:



And restoring the pivot:



```
1 #ifndef QUICK_SORT_H
2 #define QUICK_SORT_H
3
4 /**
5  * Order left, center, and right and hide the pivot.
6  * Then compute partition, restore the pivot and return its position.
7 */
8 template <typename Comparable>
9 int partition(vector<Comparable>& a, int left, int right) {
10     int center = (left + right) / 2;
11
12     if (a[center] < a[left])
13         std::swap(a[left], a[center]);
14     if (a[right] < a[left])
15         std::swap(a[left], a[right]);
16     if (a[right] < a[center])
17         std::swap(a[center], a[right]);
18
19     // Place pivot at position right - 1
20     std::swap(a[center], a[right - 1]);
21
22     // Now the partitioning
23     Comparable& pivot = a[right - 1];
24     int i = left, j = right - 1;
25     do {
26         while (a[++i] < pivot);
27         while (pivot < a[--j]);
28         if (i < j) {
29             std::swap(a[i], a[j]);
30         }
31     } while (i < j);
32
33     std::swap(a[i], a[right - 1]); // Restore pivot
34     return i;
35 }
```

quick\_sort.h

# REFLECTION

---

“Run” the partition algorithm ( $\text{left}=0$ ,  $\text{right}=7$ ) in the following array, using your favorite drawing tool:

$A=[24, 13, 26, 1, 2, 27, 38, 15]$

# QUICKSORT

---

The external call starts the **recursion** at the top, which handles the rest.

The partition returns the **splitting position**, so we can sort each side correctly.

Notice the **base case** of the recursion.

```
37 /**
38  * Internal quicksort method that makes recursive calls.
39  * Uses median-of-three partitioning and a cutoff of 10.
40  * a is an array of Comparable items.
41  * left is the left-most index of the subarray.
42  * right is the right-most index of the subarray.
43 */
44 template <typename Comparable>
45 void quickSort(vector<Comparable>& a, int left, int right) {
46     if (right - left > 1) {
47         int i = partition(a, left, right);
48         quickSort(a, left, i - 1); // Sort small elements
49         quickSort(a, i + 1, right); // Sort large elements
50     } else { // Do an insertion sort on the subarray
51         if (a[left] > a[right]) {
52             std::swap(a[left], a[right]);
53         }
54     }
55 }
56 /**
57  * Quicksort algorithm (driver).
58 */
59 template <typename Comparable> void quickSort(vector < Comparable > &a) {
60     quickSort(a, 0, a.size() - 1);
61 }
62 }
```

quick\_sort.h

# REFLECTION

---

“Run” the quicksort algorithm on the following array using your favourite drawing tool.

$$A=[24, 13, 26, 1, 2, 27, 38, 15]$$

What is the time complexity of the quicksort in the best case, and what would A look like in the best case?

Similarly, what is the time complexity in the worst case, and what would A look like in the worst case?



# QUICKSORT

---

The best-case complexity will be preserved even if the partitions are not perfect and split on a **constant fraction** of the array.

QuickSort will generally behave closer to the best-case. The worst-case is only triggered by **unlikely** worst-case partitions.

The average-case is located **between both**. When **random input** is given, **more and less balanced** partitions will alternate, so complexity will be  $O(N \log N)$ .

That is why QuickSort is implemented **everywhere**.

# AGENDA

---

- ✓ The array sorting problem
- ✓ MergeSort and its complexity
- ✓ QuickSort and its complexity
- A lower bound for comparison-based sorting

# A LOWER BOUND FOR SORTING

---

We have studied several different algorithms for **sorting by comparison**, where the result of comparisons determines the relative position of two elements.

We also studied the **upper bounds** for comparisons executed by these algorithms. The general sorting problem is **another problem** where a lower bound for the number of operations for any algorithm that solves the problem by comparison.

→ *Any algorithm which sorts  $n$  elements using comparisons executes at least  $\Omega(n \log n)$  comparisons in the worst case.*

# A LOWER BOUND FOR SORTING

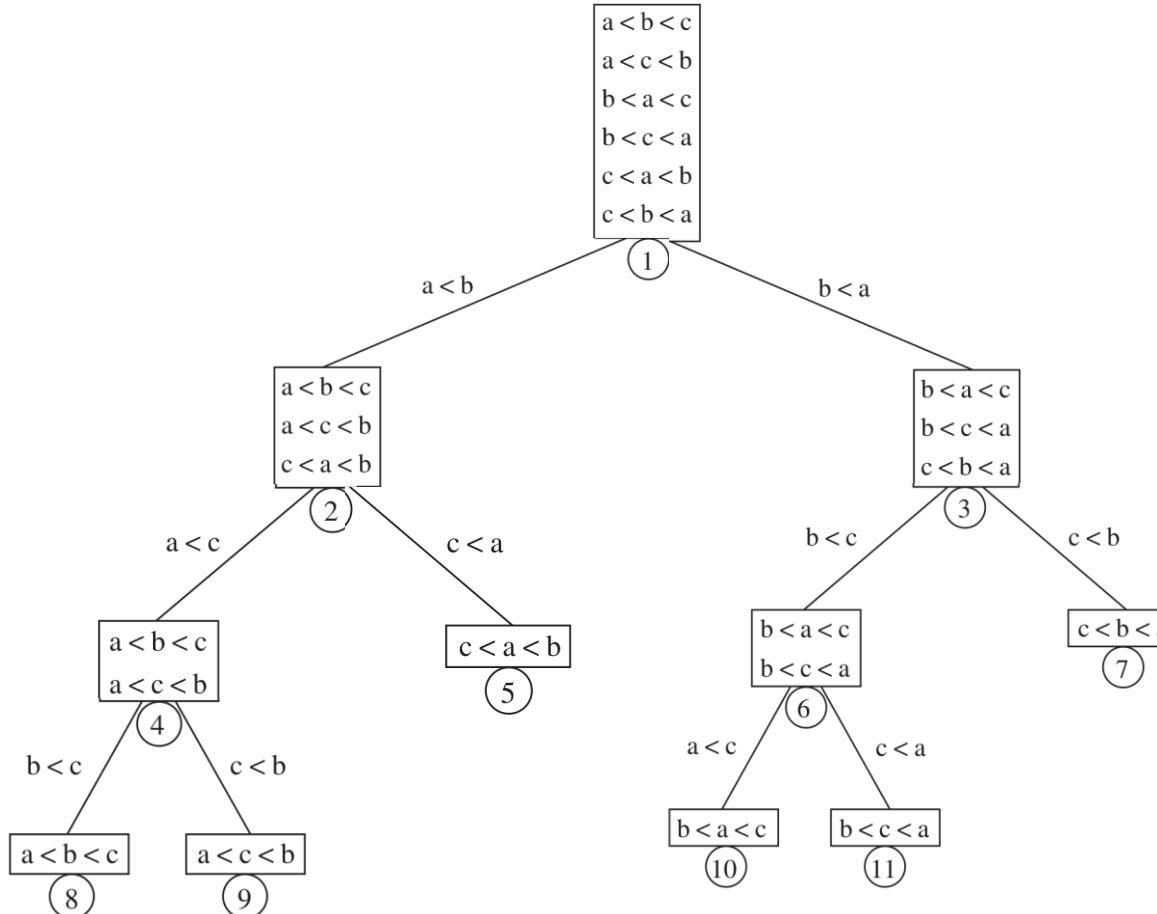
---

Let's build a **decision tree** in which internal nodes represent comparisons performed by the algorithm and the leaves represent possible answers.

Assume the input array is contains three elements [a, b, c]

# A LOWER BOUND FOR SORTING

---



# A LOWER BOUND FOR SORTING

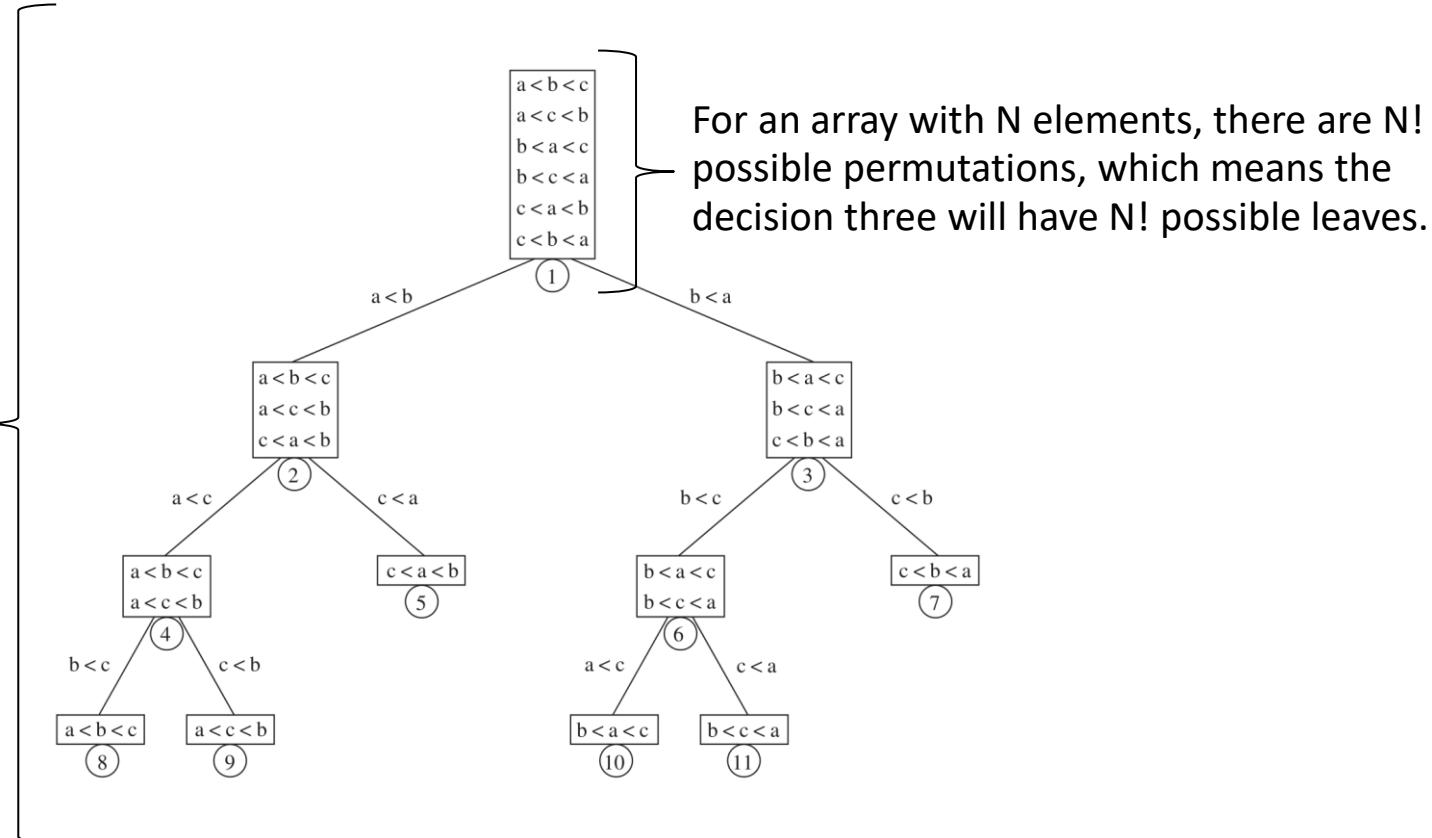
One algorithm is a path from the root to a leaf in this tree.

The maximum height represents the number of comparisons an algorithm has to make, at the very least.

The height of the tree is equal to  $\lceil \log(L) \rceil$  for L leaves.

Therefore, height is  $\lceil \log(N!) \rceil$

Conclusion: any sorting algorithm will need to conduct at least  $\lceil \log(N!) \rceil$  comparisons.



# A LOWER BOUND FOR SORTING

---

We then have:

$$\lceil \log(N!) \rceil = \Omega(n \lg n),$$

because  $\lg(n!) \geq n/4 \lg n$ , for  $n \geq 16$ .

# REFLECTION

---

For sorting, we can change the problem **slightly** and obtain faster algorithms in **special cases**.

## Definition:

Given an array  $\langle A_0, A_1, \dots, A_{N-1} \rangle$  of  $N$  non-negative integers upper bounded by  $M$ , compute a permutation of the input  $\langle A'_0, A'_1, \dots, A'_{N-1} \rangle$  such that:

$$A'_0 \leq A'_1 \leq \dots \leq A'_{N-1}.$$

Sorting an array where the maximum is known takes  $O(M+N)$ .

How?

# PRIORITY QUEUES

# AGENDA

---

- Augmenting queues with priorities
  - Binary heaps and operations
  - The C++ STL priority queue
  - HeapSort and its complexity

# AUGMENTING QUEUES WITH PRIORITIES

---

There are many **applications** of queues in computer systems:

- A printing service stores a queue of **printing jobs**
- A web server has a queue of **requests** to respond
- The operating system has a queue of **programs** to run

However, notice that in these applications we may be interested in resolving **certain** items faster than others (**short** jobs, **high-precedence** requests).

Priority queues are also useful to implement **greedy algorithms**, which take local optimization decisions.

# AUGMENTING QUEUES WITH PRIORITIES

---

We have seen ways to implement a queue, but none of them allowed to express some sort of **priority**.

## Definition:

A priority queue is an ADT in which each element additionally has a **priority** associated with it. In a priority queue, an element with high priority is served before an element with low priority.

A typical priority queue has **two operations**: `insert` which inserts an element, and `deleteMin` which **extracts** and returns the minimum element.

# AUGMENTING QUEUES WITH PRIORITIES

---

*How can we implement such a data structure?*

There are several ways to implement a priority queue:

1. **Single linked list**, for insertions in  $O(1)$  and **extraction** in  $O(N)$ .
2. **Sorted linked list**, which inverts these complexities.

We need a better **balance** between the two operations, so we can perform both operations in  $O(\log N)$ . In the following, we will **only** deal with the priorities of the objects.

# AGENDA

---

- ✓ Augmenting queues with priorities
- Binary heaps and operations
  - The C++ STL priority queue
  - HeapSort and its complexity

# BINARY HEAPS

---

Binary heaps, or just *heaps* have two properties: **structure** and **order**.

## Definition:

A *heap* is a **complete** binary tree. (**structure property**)

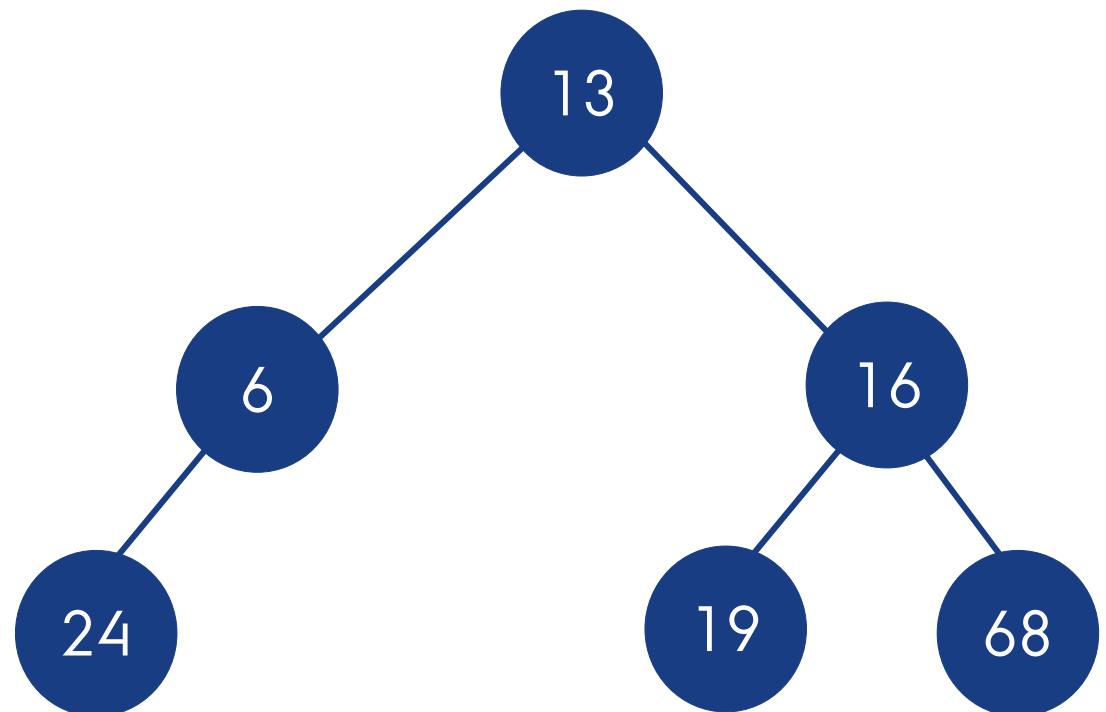
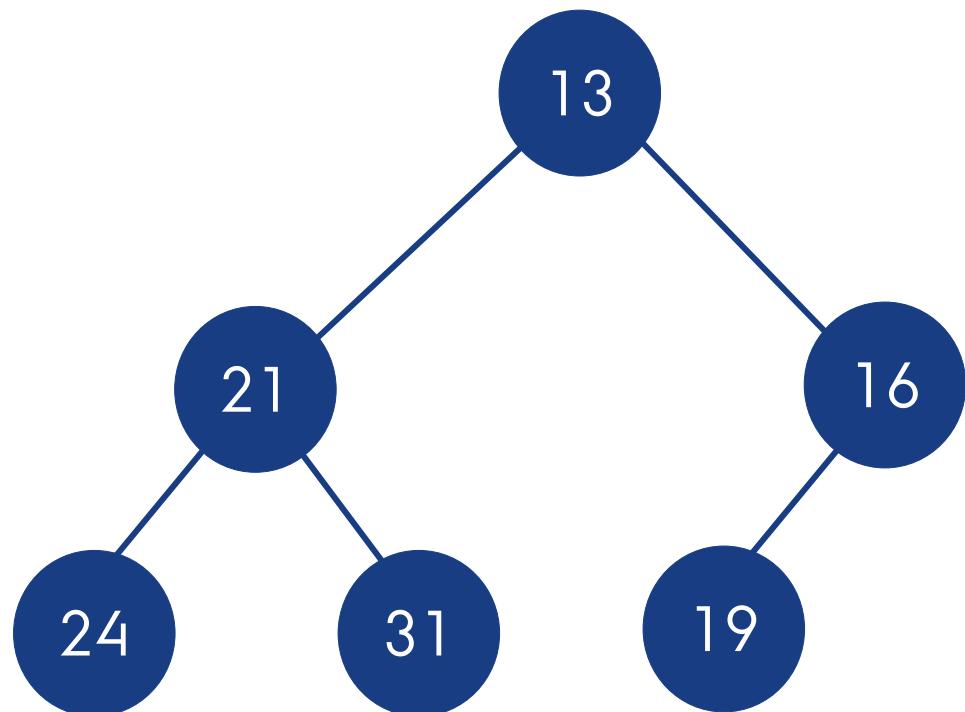
Given a min-heap, for every node  $x$ , its **key is smaller or equal than the keys of its children**. The minimum element is the **root**. (**order property**)

A complete binary tree of height  $h$  has between  $2^h$  and  $2^{h+1}-1$  nodes, so a tree with  $N$  nodes has height  $O(\log N)$ .

# BINARY HEAPS

---

The left tree is a heap, the right one **violates** the properties.



# BINARY HEAPS

---

A heap can be **linearized** in an array by just **concatenating** the levels, so no pointers or links are necessary. A **limitation** is fixing the heap size *in advance*, but **resizing** is possible.

In representation with root at index 1, a node with position  $i$  has the left child in position  $2i$  and the right child in position  $2i+1$ .

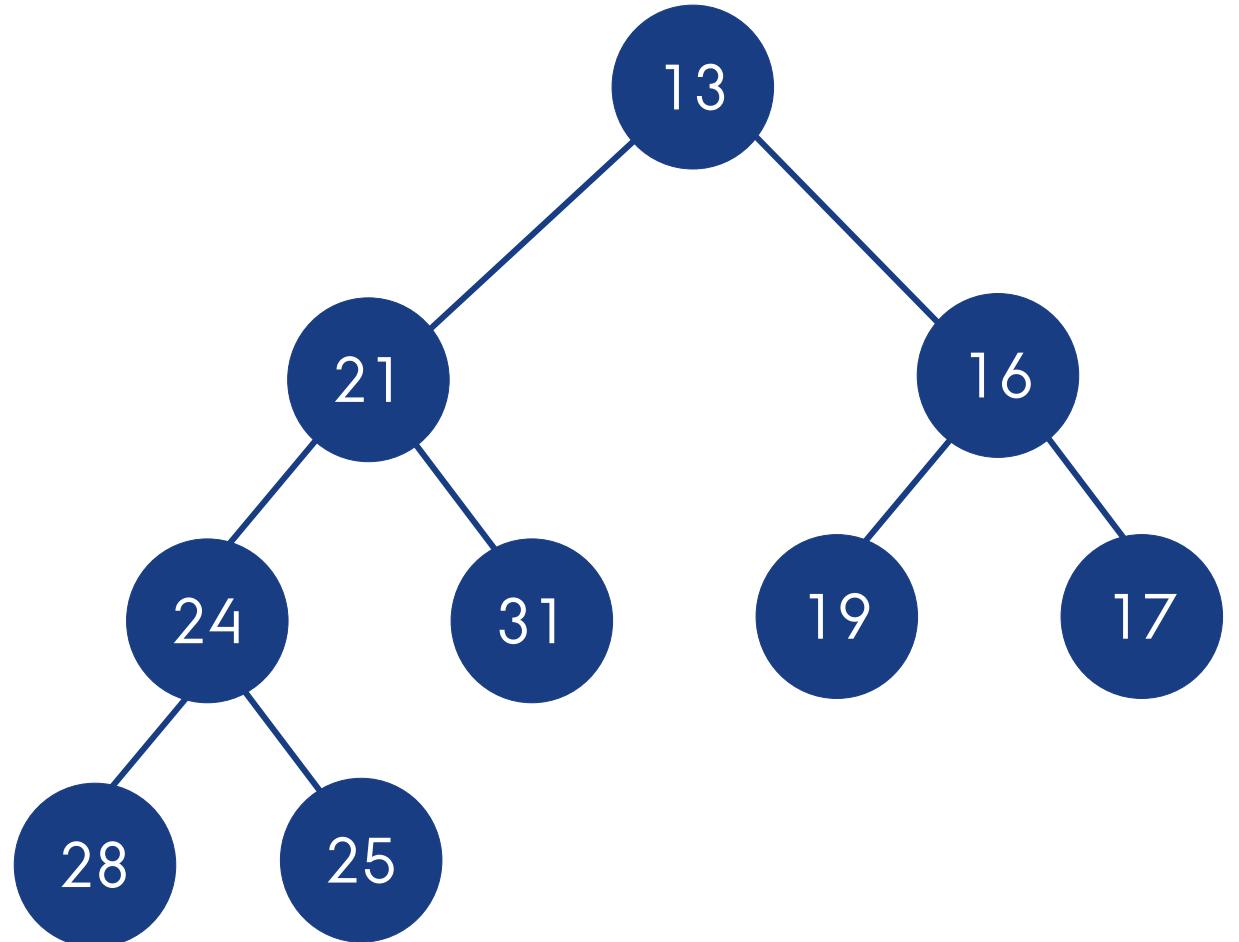
-	13	21	16	24	31	19	-
0	1	2	3	4	5	6	7

We could just as well have root at index 0, and then left child would be at  $2i + 1$  and right child at  $2i + 2$ .

# REFLECTION

---

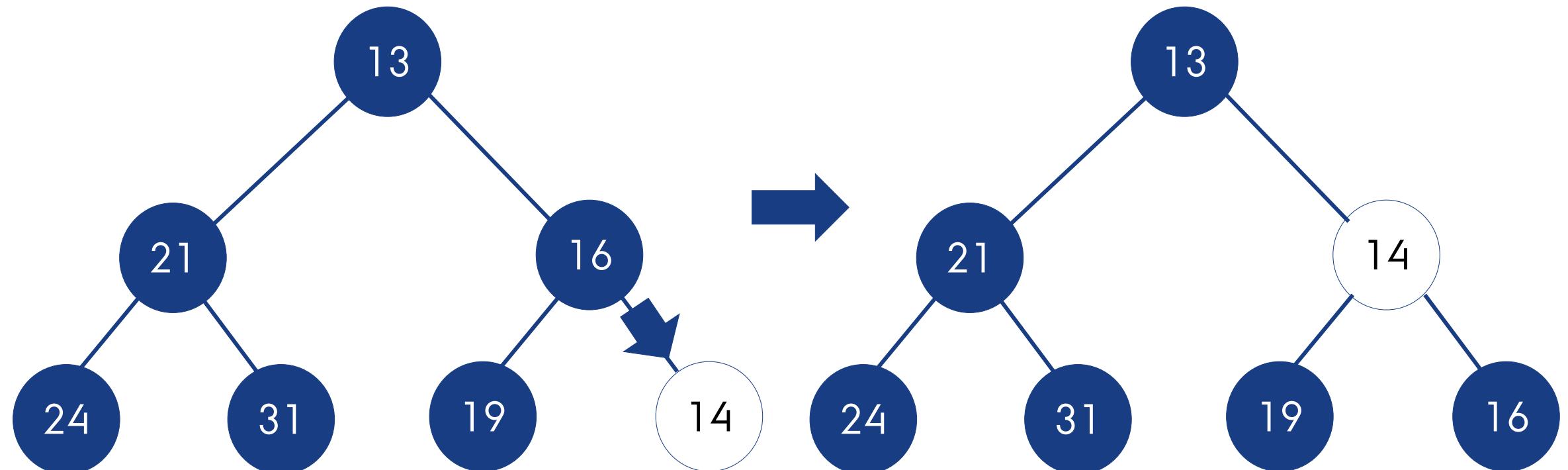
How is the following tree **linearized** in an array?



# HEAP OPERATIONS

---

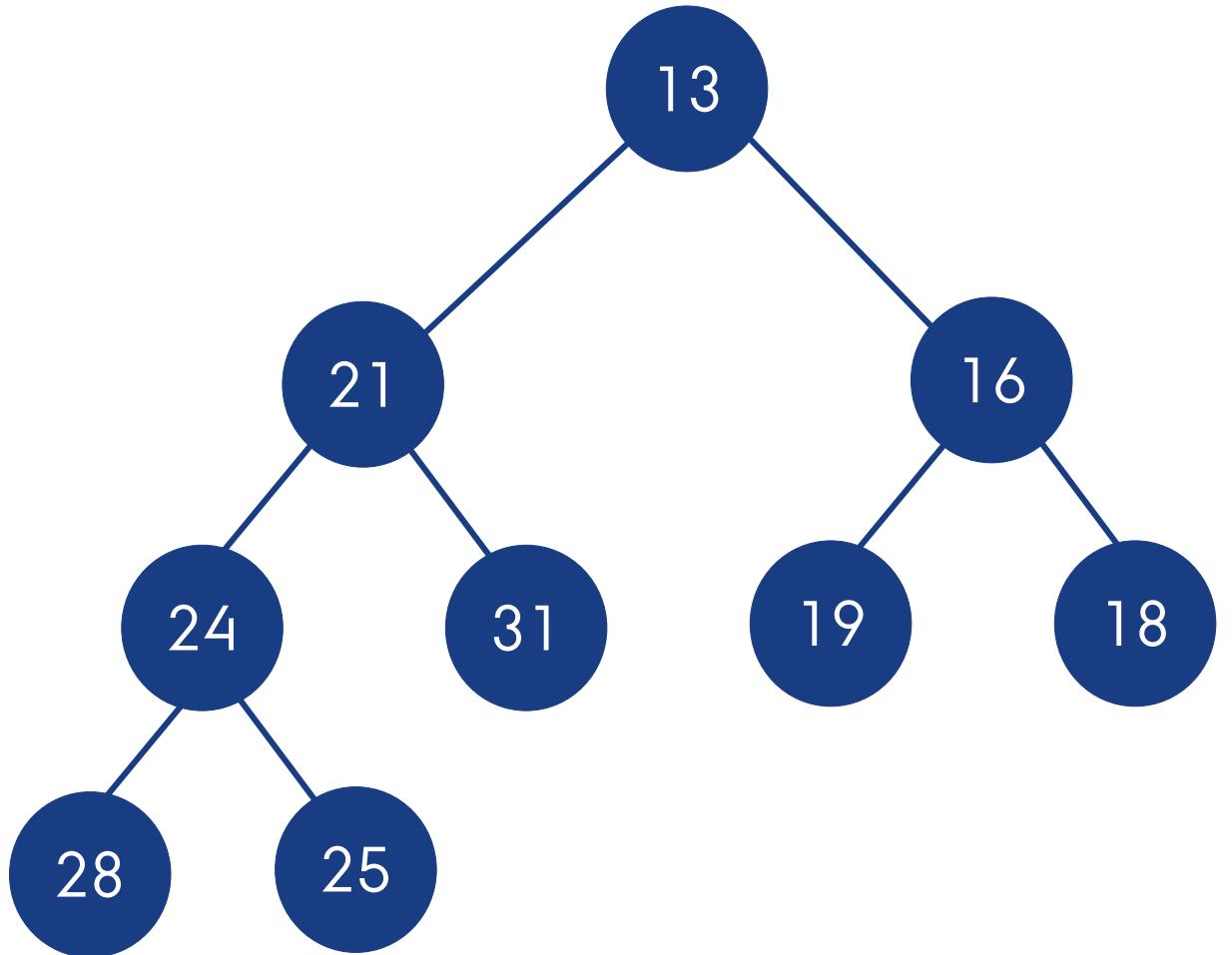
To insert an element  $x$  in the heap, we create a new node with  $x$  at the bottom and **move it up (percolate up)** until the **order** property **holds**. Let us add element  $x = 14$  to the previous heap.



# REFLECTION

---

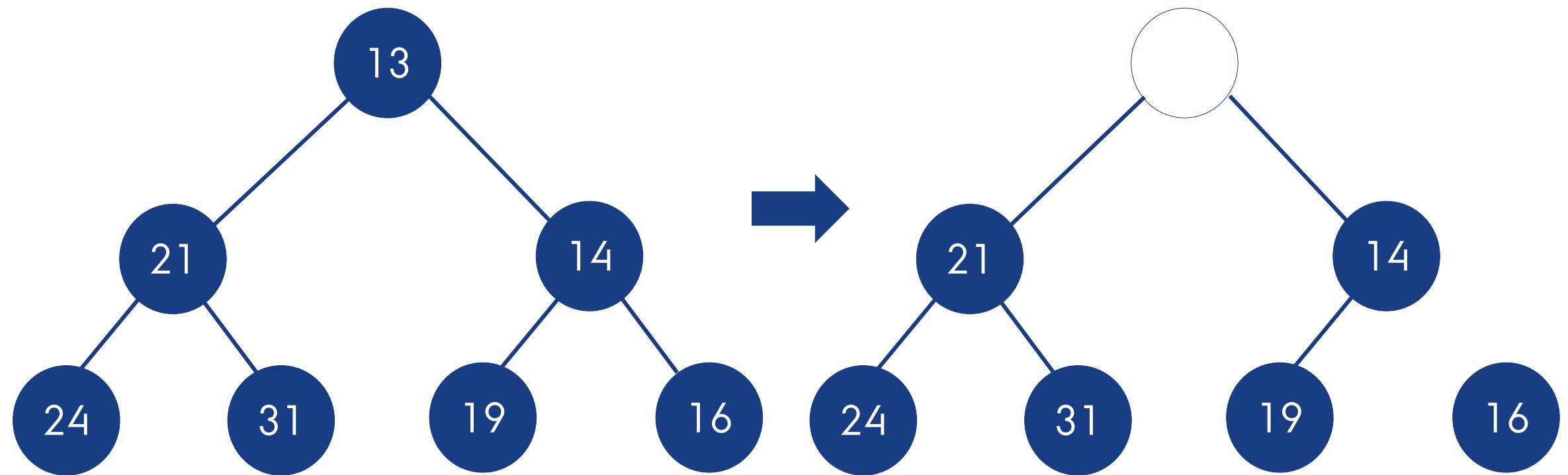
Add the element 17 to this heap?



# HEAP OPERATIONS

---

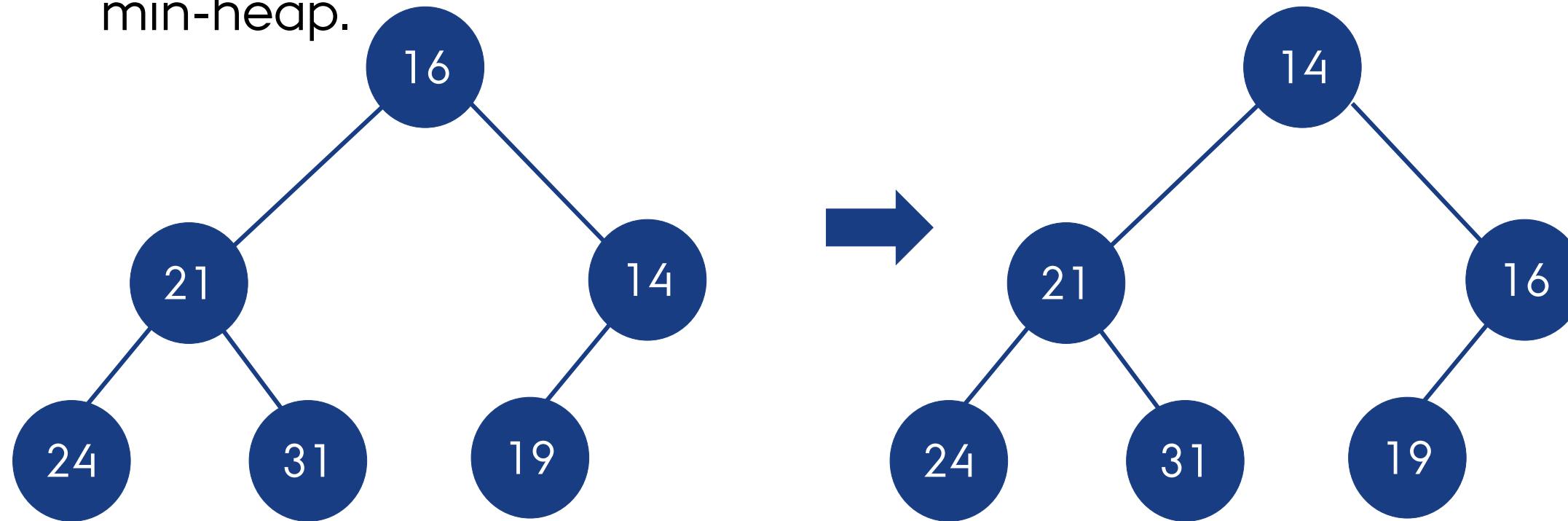
To *extract* the minimum element in the heap, we create a **hole** in the root and decrease the size by 1. Now we need to find a **new place** for the **last** element of the heap (**percolate down**).



# HEAP OPERATIONS

---

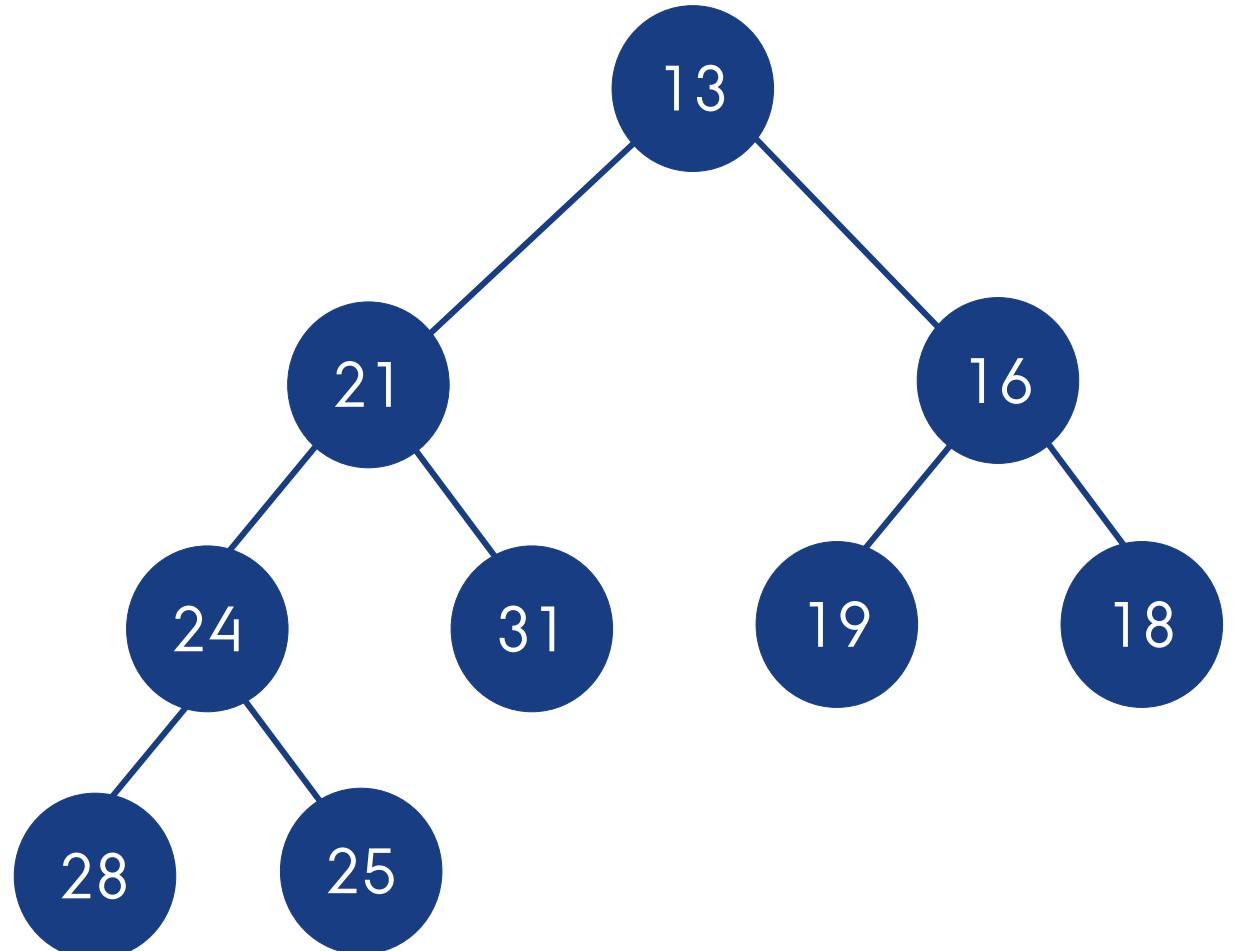
We move the hole **down (percolate down)** in the direction of the **smallest** child until we find the **correct** place for the last node. This operation is called `minHeapify`(in book), because it **restores** the min-heap.



# REFLECTION

---

Remove the smallest element from this heap?



# REFLECTION

---

Draw how to build a heap from array:

5, 7, 2, 3, 10, 1, 23



# HEAP OPERATIONS

---

The **worst-case complexity** of `minHeapify` (percolate down) is  $O(\log N)$  just like the previous operations, since it walks a **path** from the root to at most a leaf.

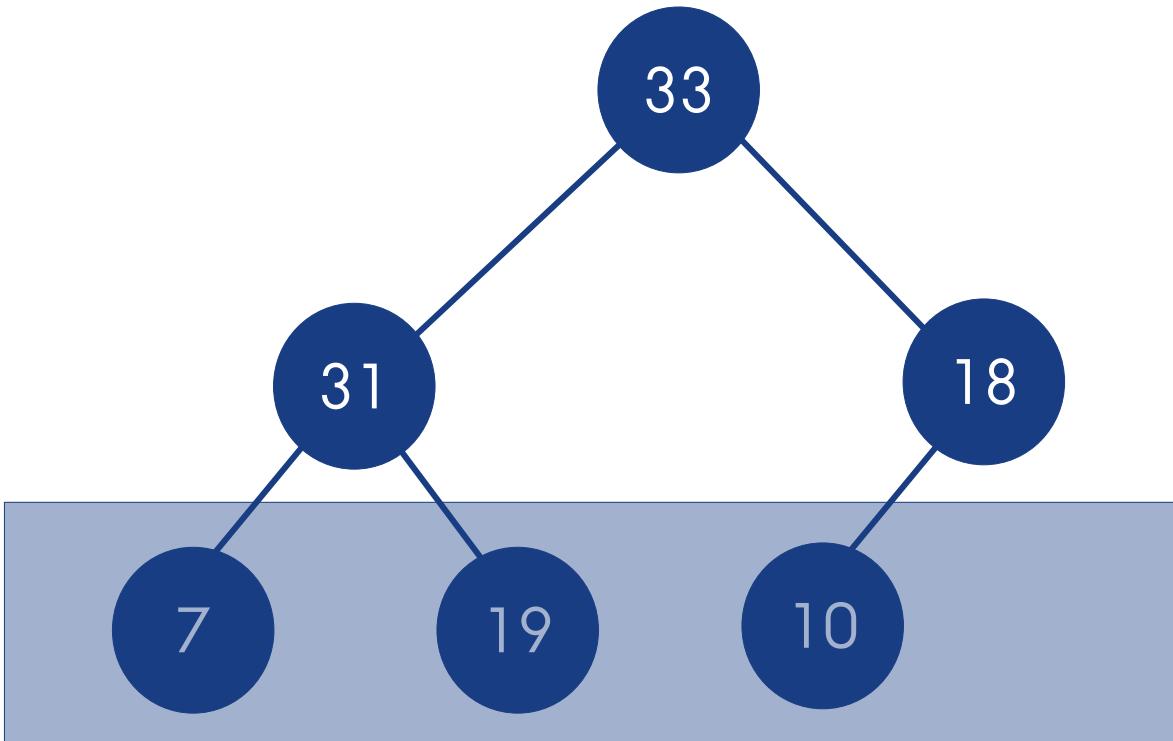
The complexity to build a heap is more involved. There are  $N$  calls to percolate down, so overall it takes  $O(N \log N)$ ?

Can this be performed more efficiently?

# HEAPIFY

---

When we heapify a “heap”, we need the two sub-heaps to fulfil the heap properties

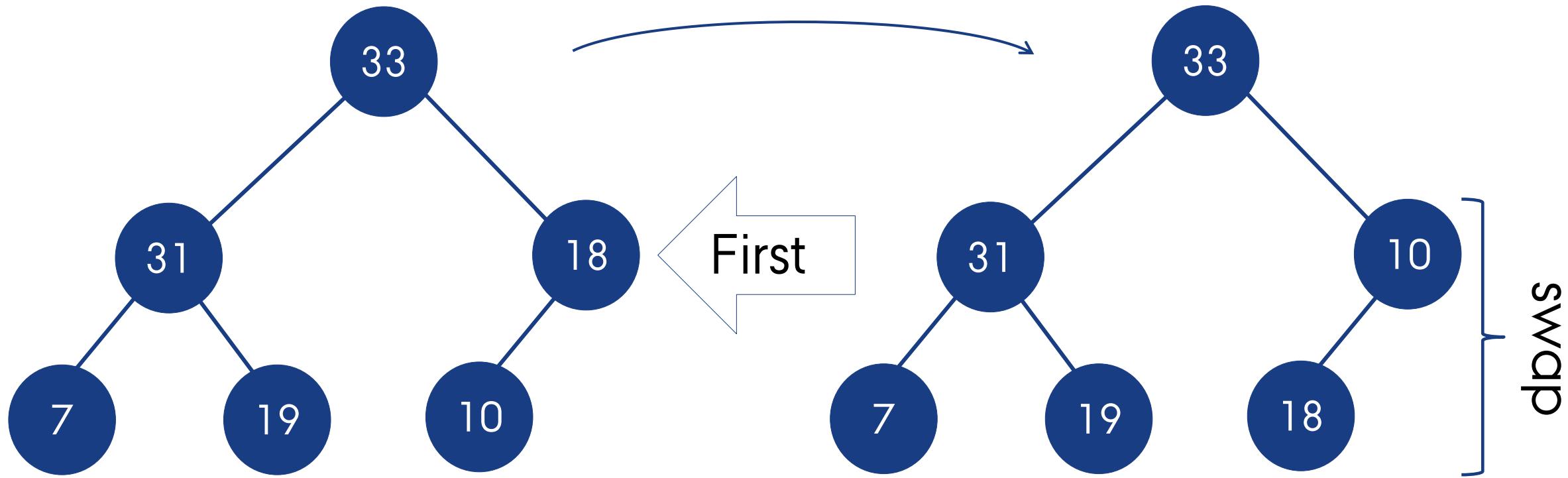


These do all fulfil the heap property

# HEAPIFY

---

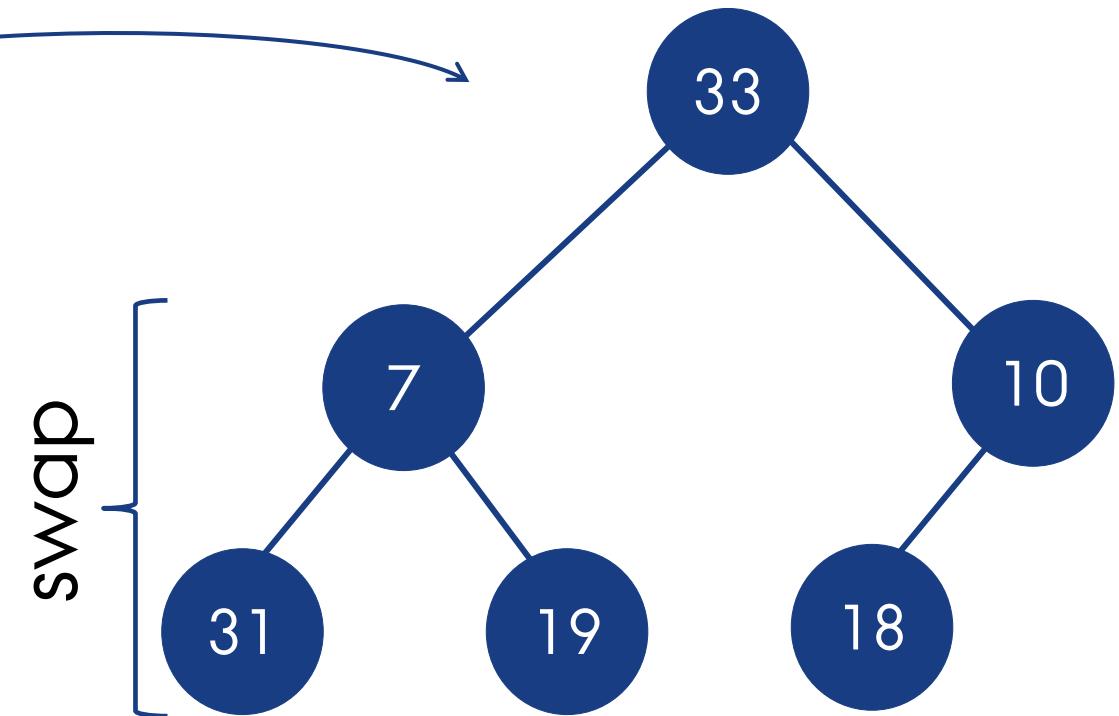
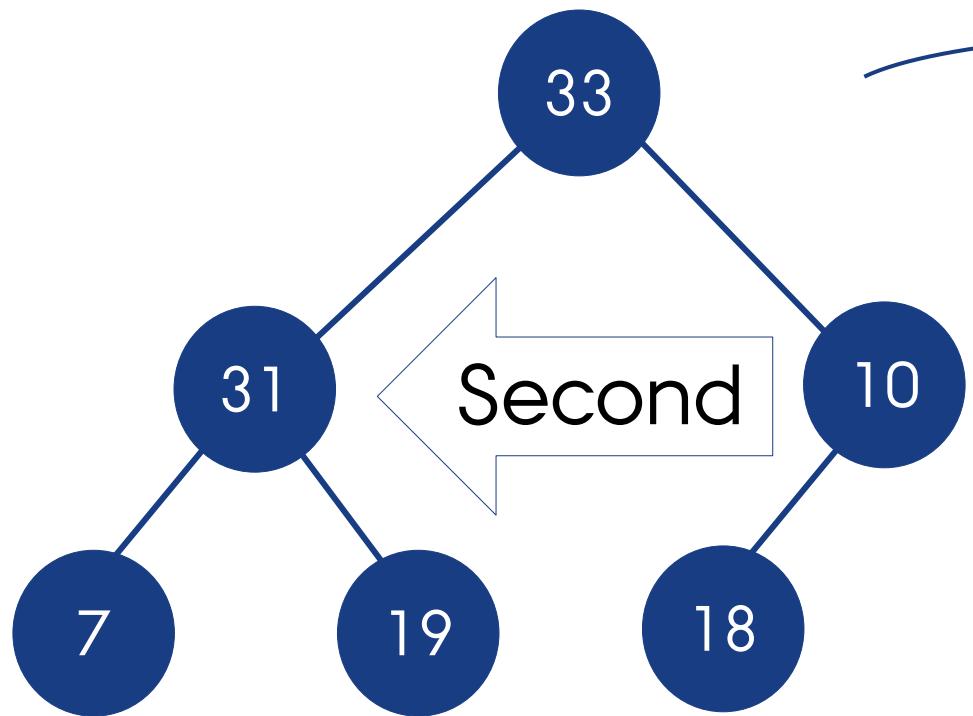
Step 1:



# HEAPIFY

---

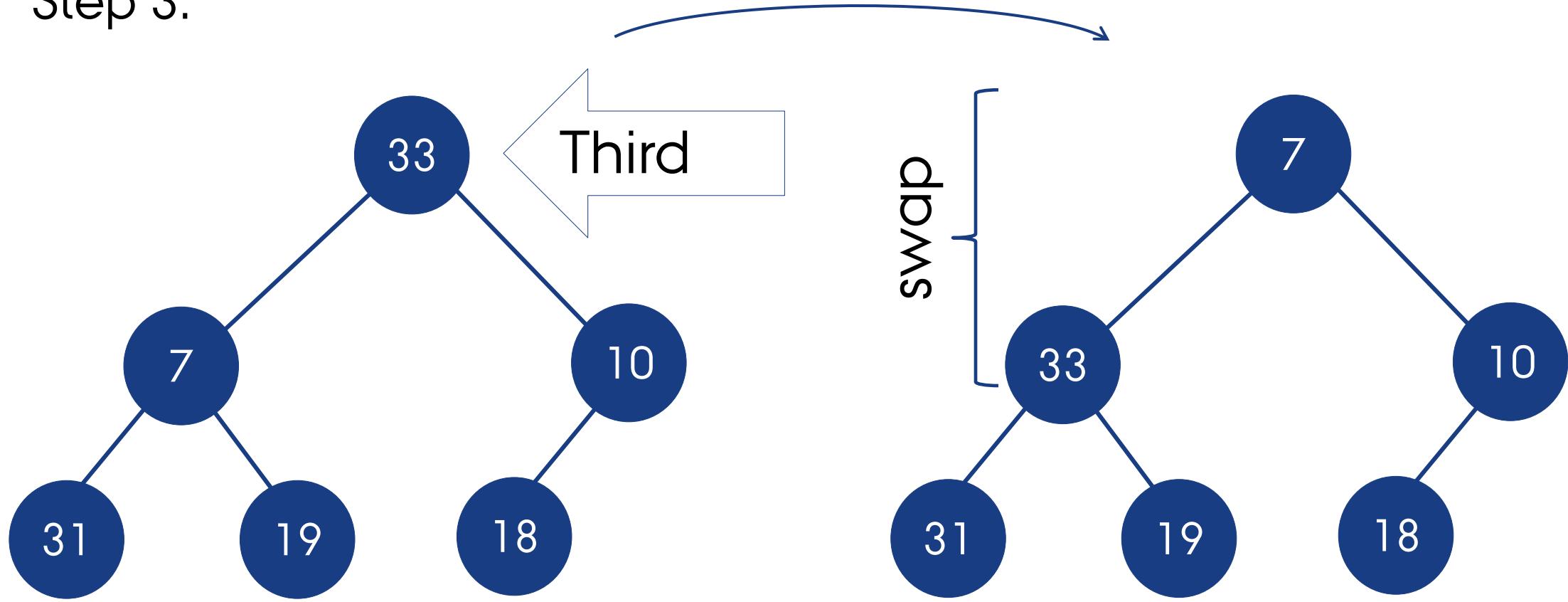
Step 2:



# HEAPIFY

---

Step 3:

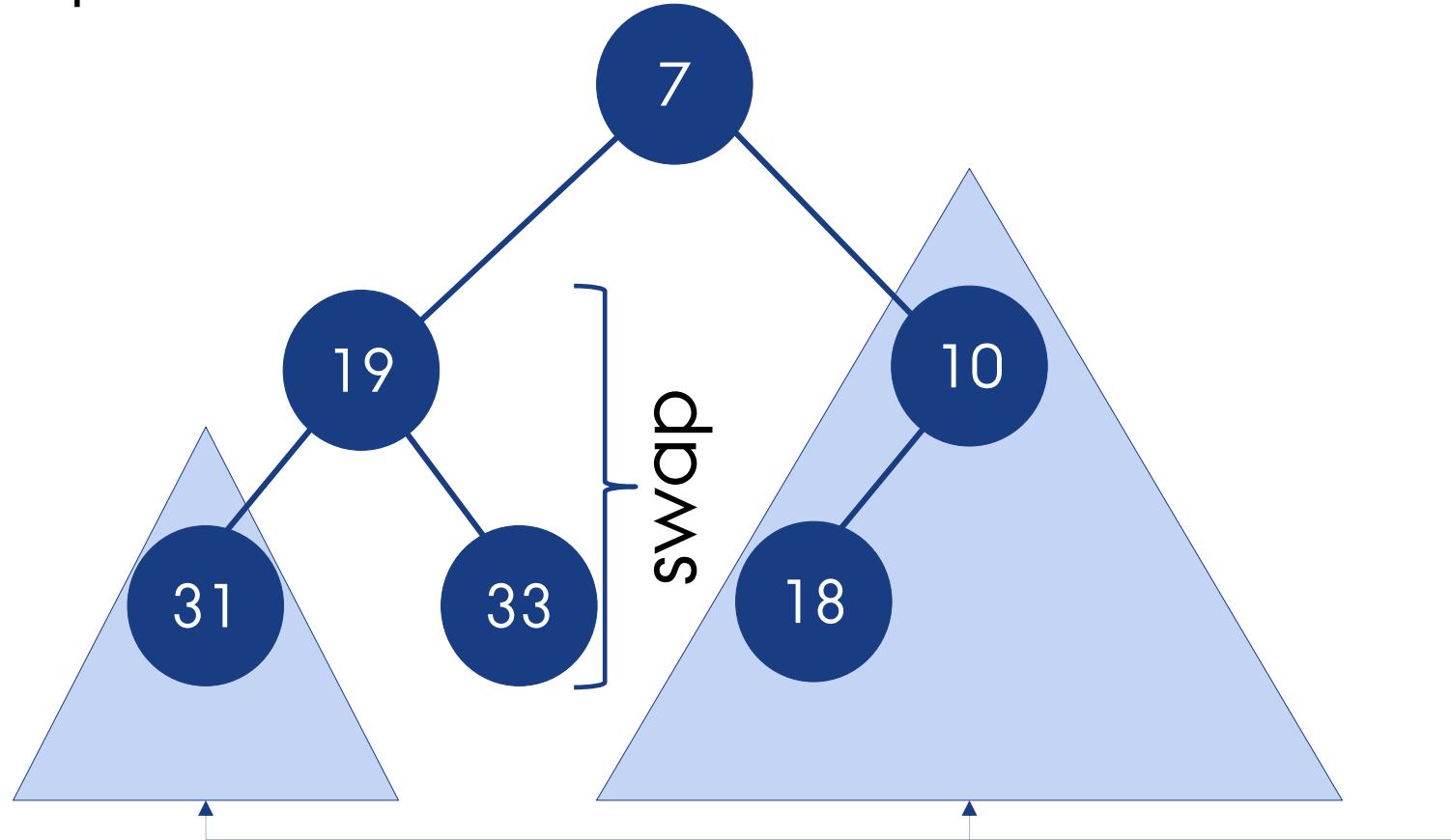


More?

# HEAPIFY

---

Step 3b:



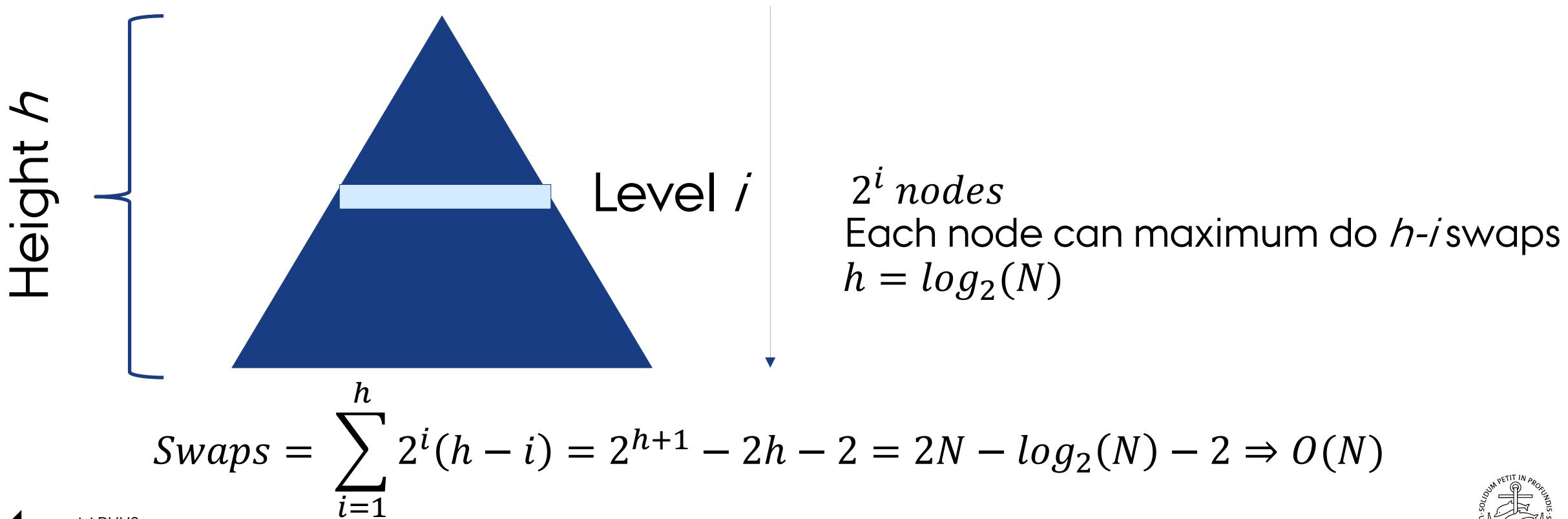
Untouched sub-trees

# HEAP OPERATIONS

---

For a node “x lines” from the bottom, we can max do x swaps

How many nodes are on level x?



# OTHER HEAP OPERATIONS

---

Another operation that can be implemented are `decreaseKey` which **decreases** a node  $x$  by a positive **amount**  $y$ . It works by moving the node **up**.

Conversely, an `increaseKey` operation increases a node  $x$  by a positive **amount**  $y$ . It works by moving the node **down**.

Finally, a `remove` operation can be supported by **decreasing** the node by its value and then **extracting** the minimum.

# AGENDA

---

- ✓ Augmenting queues with priorities
- ✓ Binary heaps and operations
- The C++ STL priority queue
- HeapSort and its complexity

# THE C++ PRIORITY QUEUE

---

The C++ priority queue is implemented in `std::priority_queue` using a *max-heap*. (Can also implement a min priority queue by using `std::greater<T>` as comparison operator)

The interface to the most important member functions is:

```
template<typename T>
class PriorityQueue
{
public:
    virtual void push(const T& x) = 0;
    virtual void pop() = 0;
    virtual T top() = 0;
    virtual bool empty() const = 0;
    virtual ~PriorityQueue() {}
};
```

→ *How can we implement this interface with our min-heap?*

# AGENDA

---

- ✓ Augmenting queues with priorities
- ✓ Binary heaps and operations
- ✓ The C++ STL priority queue
- HeapSort and its complexity

# HEAPSORT

---

*We can extract a minimum element of an array in logarithmic time. We can use this to sort the array!*

The HeapSort algorithm builds a heap from the input array and **consecutively** extracts the  $\Delta$ minimum elements in sequence.

We only need to be careful about **where** to store extracted elements, which requires some modifications to minHeapify. The heap will also be changed so that the root is in **position 0**.

# HEAPSORT

---

A basic version could look like this:

```
template<typename T>
void heapsort(vector<T>& array)
{
    MinHeap<T> heap(array); // Heapify array
    array.clear();           // Clear orig. array

    while(!heap.isEmpty())
    {
        T smallest = heap.peek(); // Inspect smallest value
        heap.remove();          // remove it from Heap
        array.push_back(smallest); // Push it to orig. vector
    }
}
```

→ *What is the overall time and memory complexity?*

# HEAPSORT - C++ OPTIMIZED

```
inline size_t leftChild(size_t i) { return 2*i+1; }
inline size_t rightChild(size_t i) { return 2*i+2; }

// 'node' is starting point, 'n' is size of heap
template <typename T>
void minHeapify(vector<T> &array, int node, int n)
{
    int child;
    T tmp = std::move(array[node]);
    for (; leftChild(node) < n; node = child) {
        child = leftChild(node);
        if (child != n - 1) // Within range
            if (array[rightChild(node)] < array[child])
                child = rightChild(node);
        if (array[child] < tmp)
            array[node] = std::move(array[child]);
        else
            break;
    }
    array[node] = std::move(tmp);
}
```

```
template <typename T>
void heapSort(vector<T> &array)
{
    // Build Min-Heap
    for (int i = array.size() / 2 - 1; i >= 0; --i)
        minHeapify(array, i, array.size());

    // Swap smallest with last element
    // and re-heapify sub-array
    for (int j = array.size() - 1; j > 0; --j) {
        std::swap(array[0], array[j]);
        minHeapify(array, 0, j);
    }

    // Reverse to have ascending order
    std::reverse(array.begin(), array.end());
}
```

→ *Time and memory complexity?*

# BINARY SEARCH TREES

# RECAP OF LAST WEEK'S LECTURE

---

- ✓ In the last lecture we ...

# HEAP OPERATIONS

---

The **worst-case complexity** of `minHeapify` is  $O(\log N)$  just like the previous operations, since it walks a **path** from the root to at most a leaf.

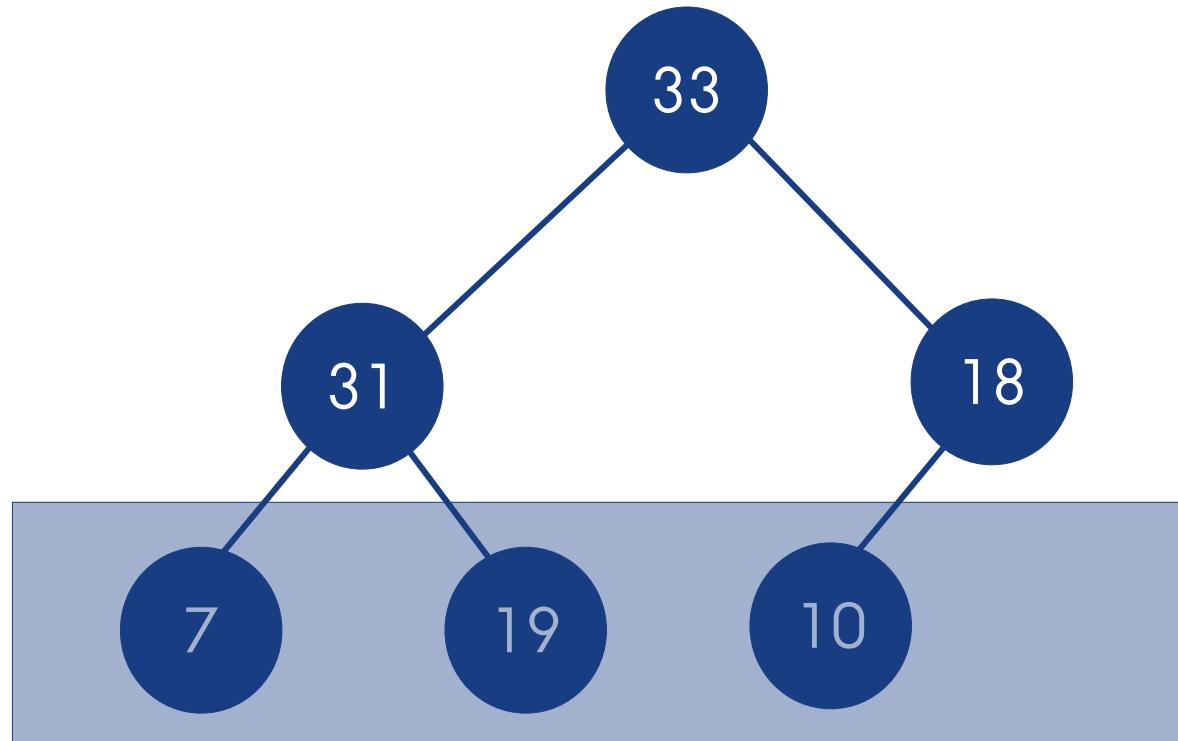
The complexity of `buildHeap` is more involved. There are  $N/2$  calls to `minHeapify`, so overall it takes  $O(N \log N)$ ?

Note however that the calls will heapify **increasingly taller** heaps. Our estimate is actually **pessimistic**.

# MINHEAPIFY

---

When we heapify an “heap”, we need the two sub-heaps to fulfil the heap properties

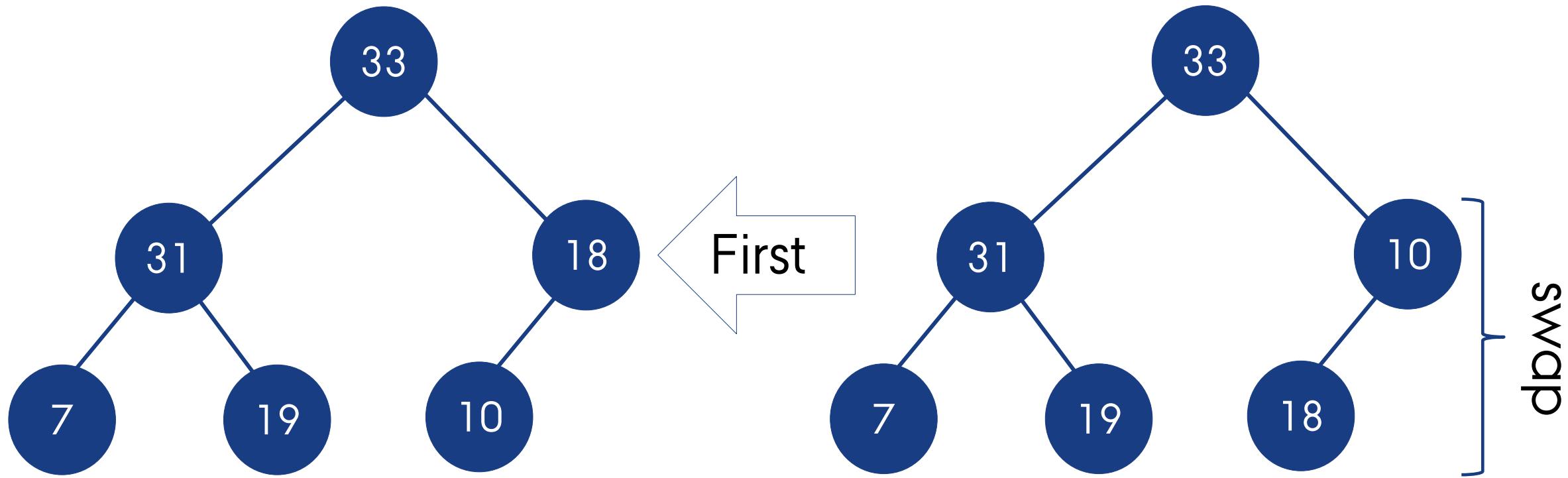


These do all fulfil the heap property

# MINHEAPIFY

---

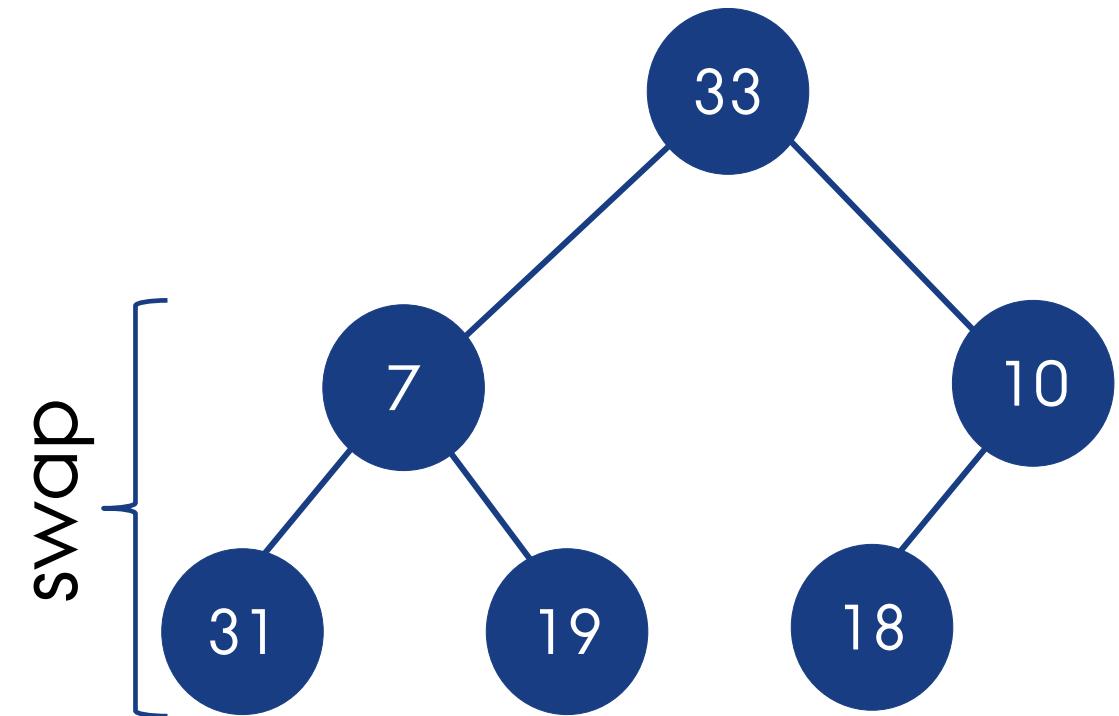
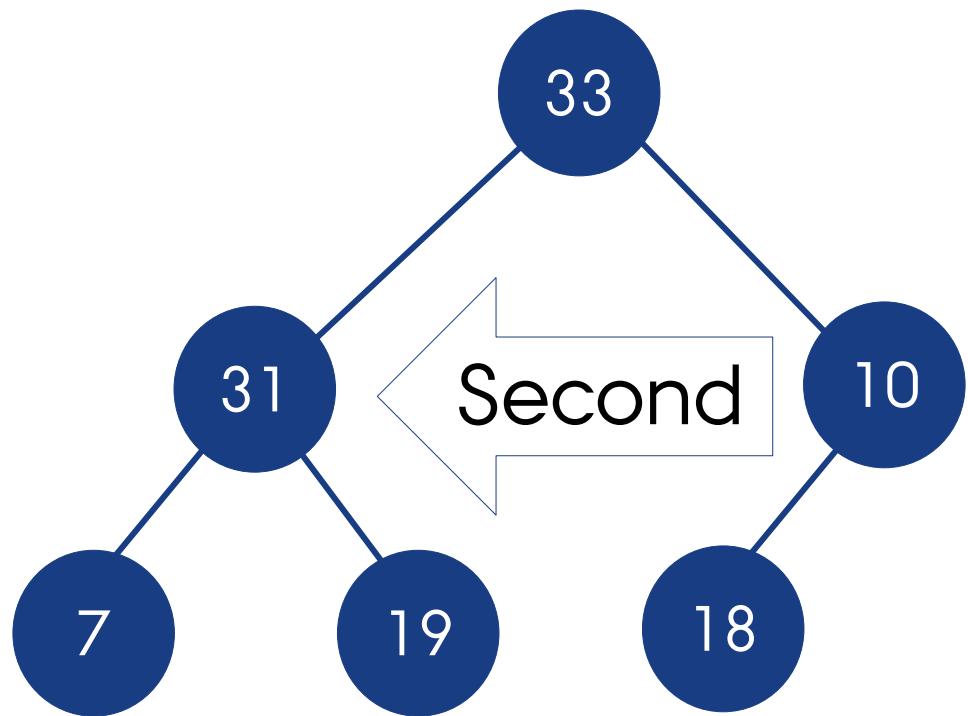
Step 1:



# MINHEAPIFY

---

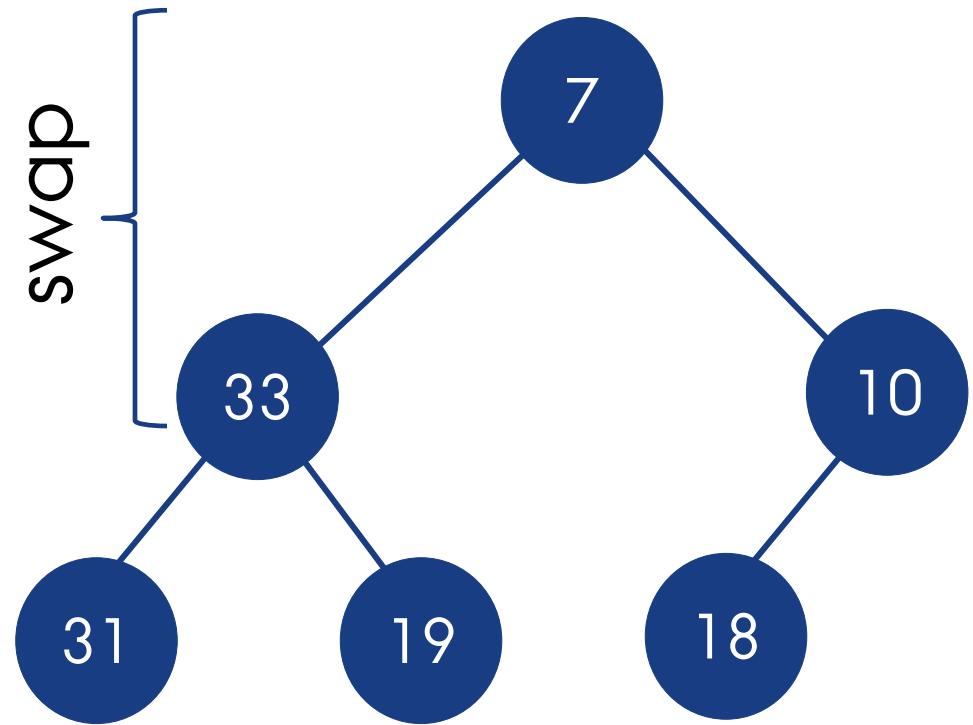
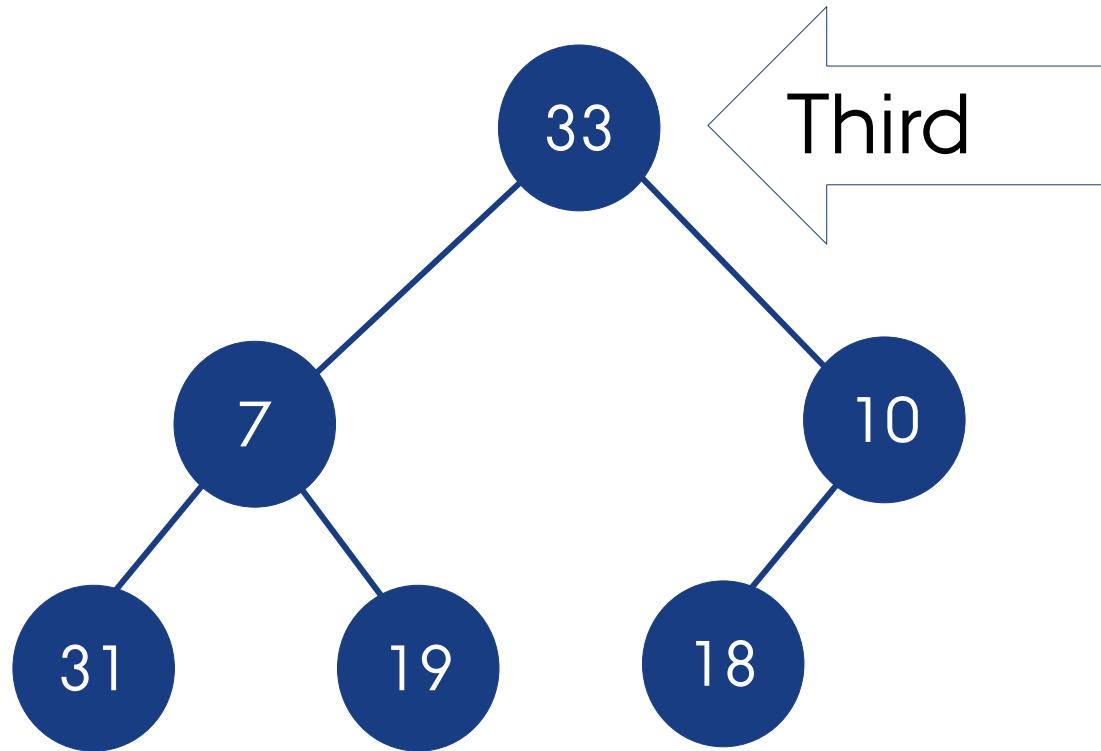
Step 2:



# MINHEAPIFY

---

Step 3:

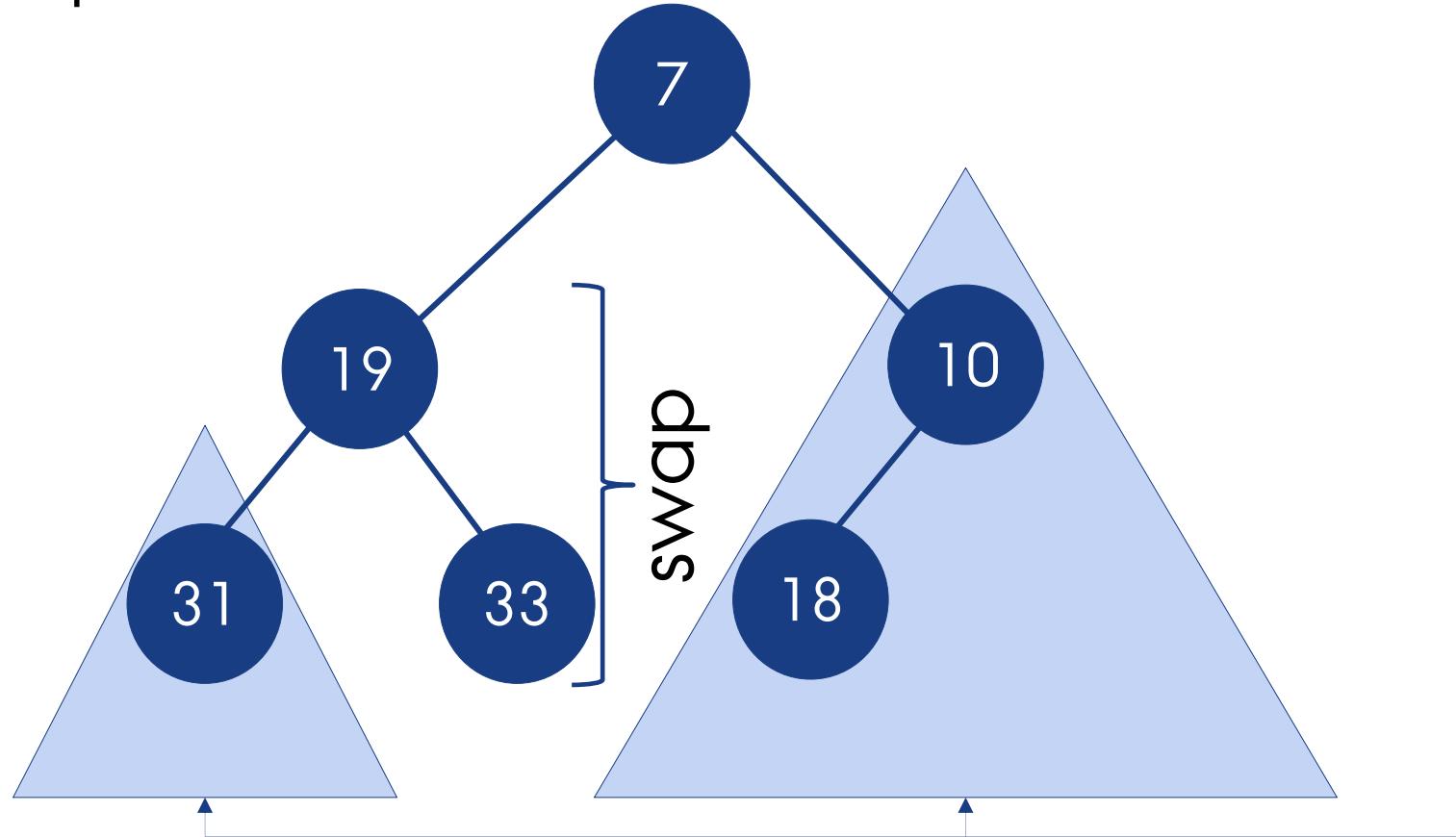


More?

# HEAPIFY

---

Step 3b:

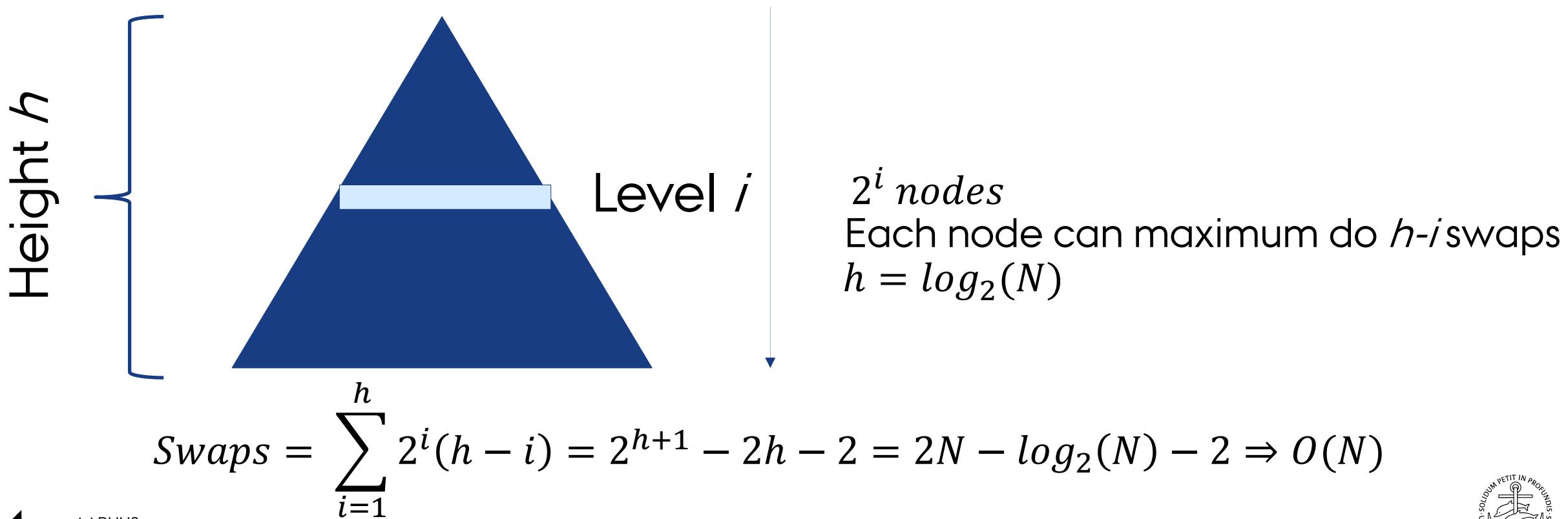


# HEAP OPERATIONS

---

For a node “x lines” from the bottom, we can max do x swaps

How many nodes are on level x?



# AGENDA

---

- Binary Search Trees (BSTs)
  - BST operations and their complexity
  - Balanced Trees
  - AVL Trees and rotations

# THE NEED FOR FASTER DATA STRUCTURES

---

There are several computational tasks which require efficient operations over data structures:

- Implementing a **file system**
- Storing and evaluating large **arithmetic expressions**
- Searching and modifying a **dynamic ordered set**

We got logarithmic operations with **heaps**, but no **fast search**.

Hence, with the data structures we have seen so far, we can only perform these operations in **linear time**, which is not fast enough for large inputs.

# BINARY SEARCH TREES

---

Just like heaps, *binary search trees* are special kinds of trees with **structure** and **order** properties.

## Definition:

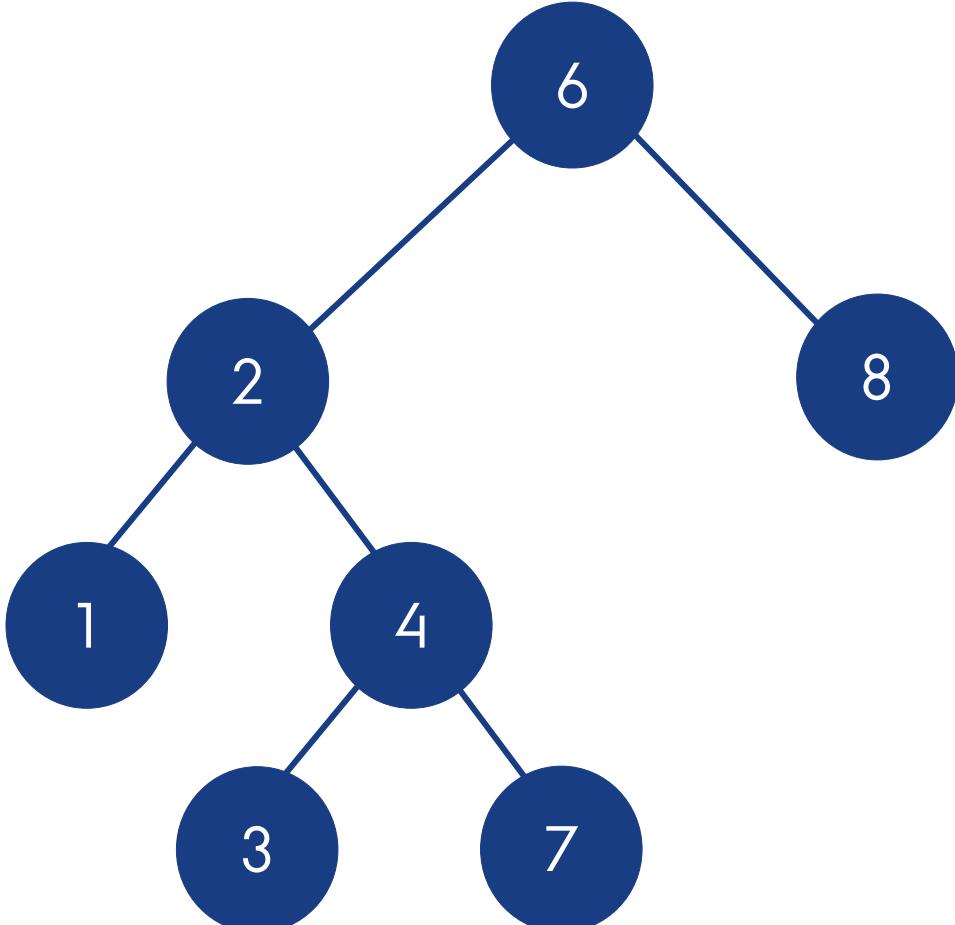
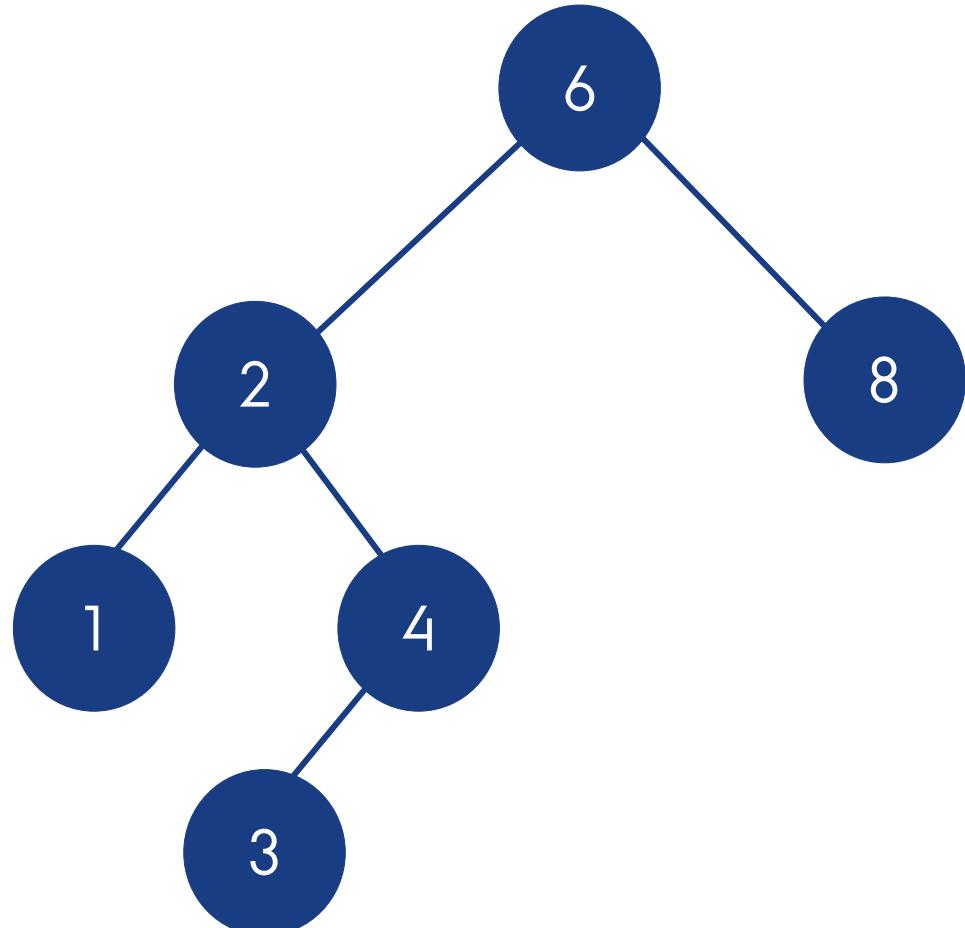
A *binary search tree (BST)* is a **binary** tree such that for every node  $y$ , the values of all the items in its left subtree are **smaller** than the item in  $y$ , and the values of all the items in its right subtree are **larger** than the item in  $y$ .

A typical BST has **three operations**: **insert**, **contains** and **remove** which respectively **add**, **search** and **delete** a key  $x$ .

# BINARY SEARCH TREES

---

The left is a BST, the right one **violates** the **order** property. *Why?*



# AGENDA

---

- ✓ Binary Search Trees (BSTs)
- BST operations and their complexity
  - Balanced Trees
  - AVL Trees and rotations

# BST INTERFACE

binary\_search\_tree.h

## Templated binary search tree:

- Constructors and assignment operator are standard
- Interface hides **internal implementation**
- Internal node type needs **pointers** for children and space for **element**
- Finally, notice the **root node** at the very end.

```
1 #ifndef _BINARY_SEARCH_TREE_H_
2 #define _BINARY_SEARCH_TREE_H_
3
4 #include <stdexcept>
5 #include <algorithm>
6 using namespace std;
7
8 template<typename Comparable>
9 class BinarySearchTree {
10 public:
11     BinarySearchTree() : root {nullptr} {}
12
13     BinarySearchTree(const BinarySearchTree& rhs) : root {nullptr} {
14         root = clone(rhs.root);
15     }
16
17     ~BinarySearchTree() { makeEmpty(); }
18
19     BinarySearchTree& operator=(const BinarySearchTree& rhs) {
20         BinarySearchTree copy(rhs);
21         std::swap(*this, copy);
22         return *this;
23     }
24
25     const Comparable& findMin() const; // find min element
26     const Comparable& findMax() const; // find max element
27     bool isEmpty() const; // test for emptiness
28     void printTree(ostream& out = cout) const;
29     void makeEmpty(); // empty tree
30     void insert(const Comparable& x); // insert item
31     bool contains(const Comparable& x) const; // look for item
32     void remove(const Comparable& x); // remove item
33
34 private:
35     struct BinaryNode {
36         Comparable element;
37         BinaryNode *left;
38         BinaryNode *right;
39
40         BinaryNode(const Comparable& theElement, BinaryNode* lt, BinaryNode* rt) :
41             element {theElement}, left {lt}, right {rt} {}
42     };
43
44     BinaryNode *root;
```

# BINARY SEARCH TREES

---

We will structure our implementation in two layers:

1. A **template outer layer**, with the ADT interface
2. A **private inner layer**, implemented recursively

The inner layer can also be implemented iteratively, at the cost of code **legibility**. The outer layer **will just call** the inner layer through operations on the **internal node** type.

This will be the most complex implementation so far in the course, so details will be omitted (look at Brightspace). :)

# BST INTERFACE

binary\_search\_tree.tpp

## Templated binary search tree:

- Design of outer layer is **simple**, inner layer handles details
- Outer layer calls internal recursive functions from the **root node**
- Notice again the use of **exceptions**

```
1  /**
2   * Find the smallest item in the tree.
3   * Throw UnderflowException if empty.
4   */
5  template<typename Comparable>
6  const Comparable& BinarySearchTree<Comparable>::findMin() const {
7      if (isEmpty()) throw underflow_error("tree is empty.");
8      return findMin(root)->element;
9  }
10 /**
11  * Find the largest item in the tree.
12  * Throw UnderflowException if empty.
13 */
14 template<typename Comparable>
15 const Comparable& BinarySearchTree<Comparable>::findMax() const {
16     if (isEmpty()) throw underflow_error("tree is empty.");
17     return findMax(root)->element;
18 }
19 /**
20  * Returns true if x is found in the tree.
21  */
22 template <typename Comparable>
23 bool BinarySearchTree<Comparable>::contains(const Comparable& x) const {
24     return contains(x, root);
25 }
26 /**
27  * Test if the tree is logically empty.
28  * Return true if empty, false otherwise.
29 */
30 template<typename Comparable>
31 bool BinarySearchTree<Comparable>::isEmpty() const {
32     return root == nullptr;
33 }
34 }
```

# BST INTERFACE

binary\_search\_tree.hpp

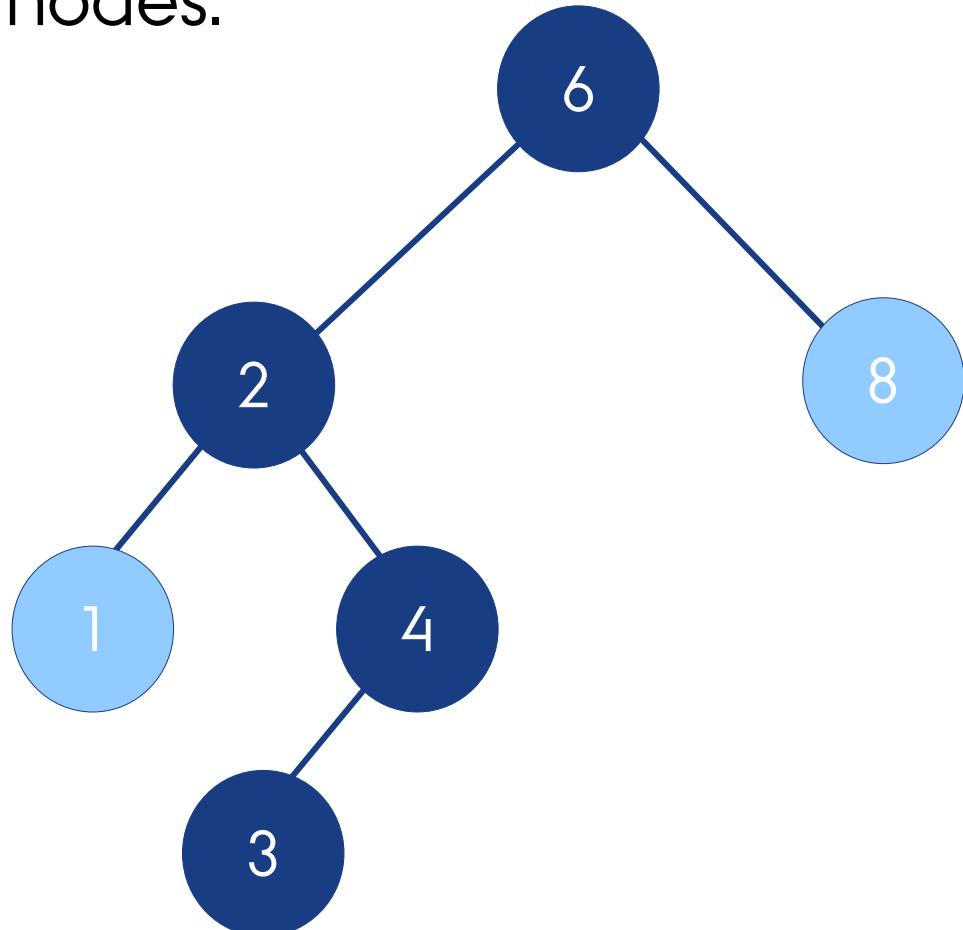
## Templated binary search tree:

- Design of outer layer is **simple**, inner layer handles details
- Outer layer calls internal recursive functions from the **root node**
- Notice again the use of **exceptions**

```
37  /**
38   * Print the tree contents in sorted order.
39   */
40  template<typename Comparable>
41  void BinarySearchTree<Comparable>::printTree(ostream& out) const {
42      if (isEmpty()) {
43          out << "Empty tree" << endl;
44      } else
45          printTree(root, out);
46  }
47
48  /**
49   * Make the tree logically empty.
50   */
51  template<typename Comparable>
52  void BinarySearchTree<Comparable>::makeEmpty() {
53      makeEmpty(root);
54  }
55
56  /**
57   * Insert x into the tree; duplicates are ignored.
58   */
59  template<typename Comparable>
60  void BinarySearchTree<Comparable>::insert(const Comparable& x) {
61      insert(x, root);
62  }
63
64  /**
65   * Remove x from the tree. Nothing is done if x is not found.
66   */
67  template<typename Comparable>
68  void BinarySearchTree<Comparable>::remove(const Comparable& x) {
69      remove(x, root);
70  }
```

# BST OPERATIONS

Finding the **minimum** and **maximum** looks for the extreme nodes.

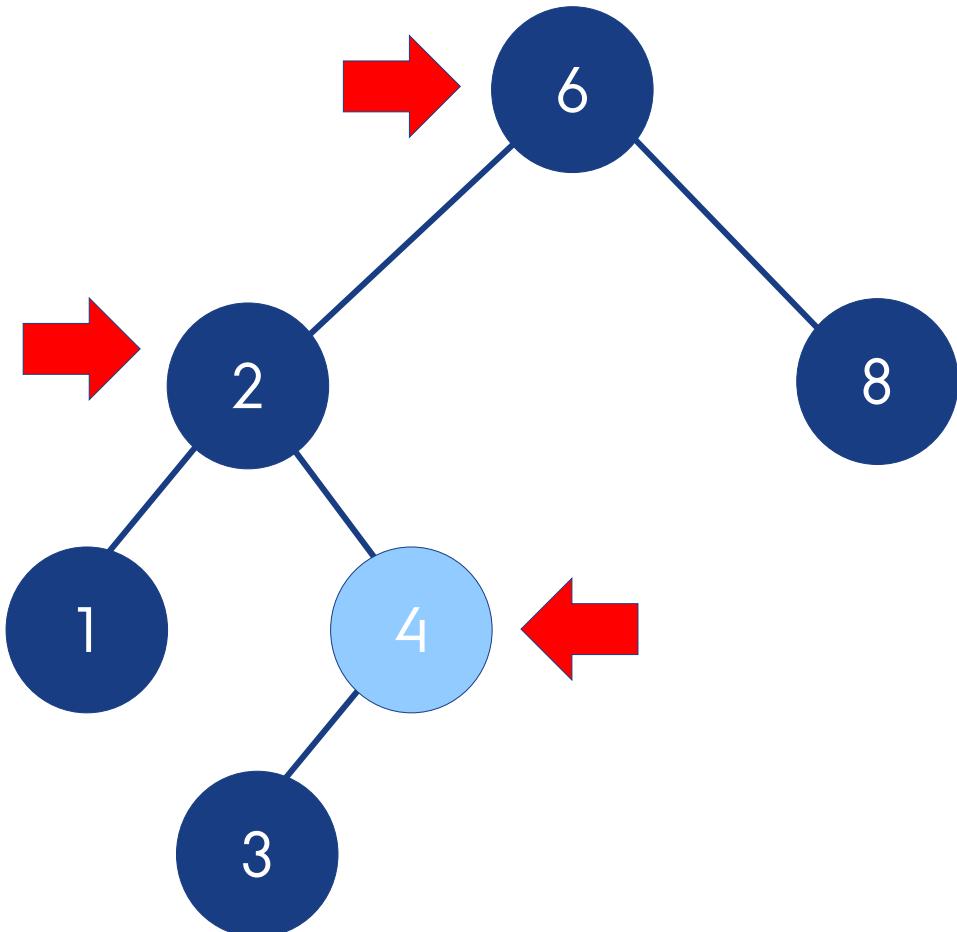


```
88     /**
89      * Internal method to find the smallest item in a subtree t.
90      * Return node containing the smallest item.
91      */
92     BinaryNode *findMin(BinaryNode* t) const {
93         if (t == nullptr)
94             return nullptr;
95         if (t->left == nullptr)
96             return t;
97         return findMin(t->left);
98     }
99
100    /**
101     * Internal method to find the largest item in a subtree t.
102     * Return node containing the largest item.
103     */
104    BinaryNode *findMax(BinaryNode* t) const {
105        if (t != nullptr)
106            while (t->right != nullptr)
107                t = t->right;
108        return t;
109    }
```

binary\_search\_tree.h

# BST OPERATIONS

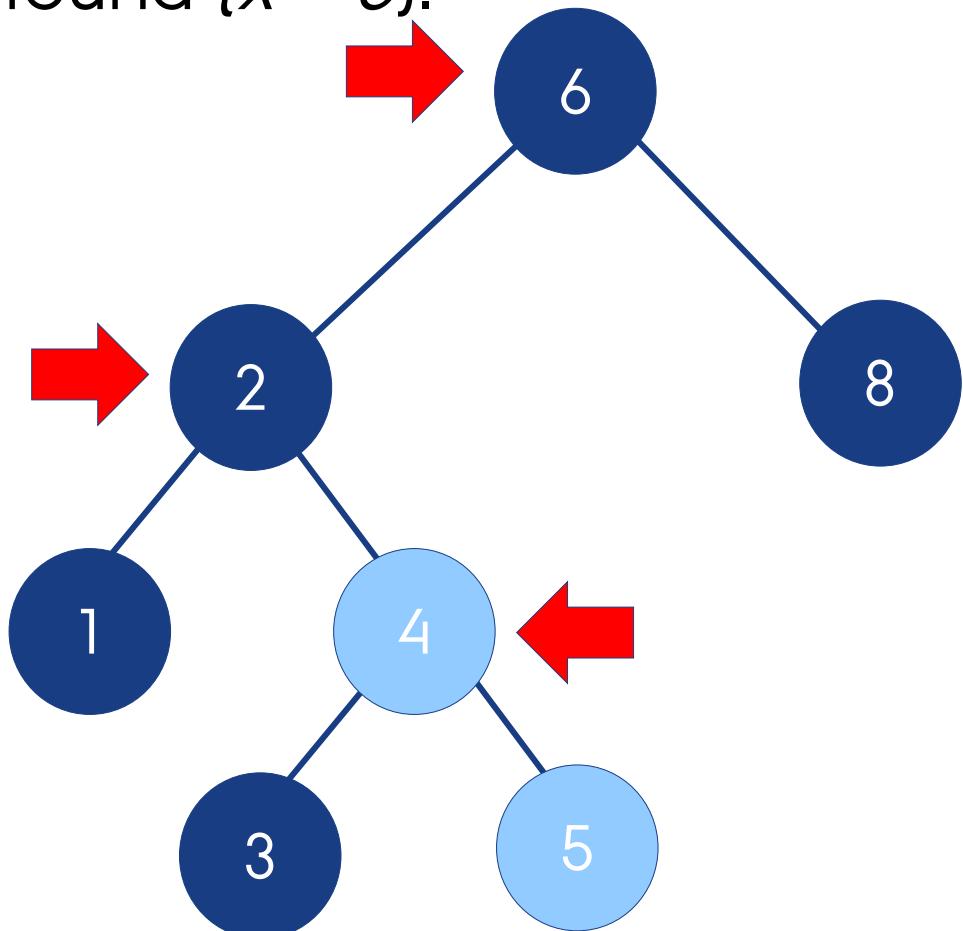
Searching starts from root, following the **order**. For example,  $x = 4$ .



```
binary_search_tree.h
111 /**
112  * Internal method to test if an item is in a subtree.
113  * x is item to search for.
114  * t is the node that roots the subtree.
115 */
116 bool contains(const Comparable& x, BinaryNode* t) const {
117     if (t == nullptr)
118         return false;
119     else if (x < t->element)
120         return contains(x, t->left);
121     else if (t->element < x)
122         return contains(x, t->right);
123     else
124         return true; // Match
125 }
```

# BST OPERATIONS

**Insertion** starts from root, following the **order** until a position is found ( $x = 5$ ).



```
binary_search_tree.h
46  /**
47   * Internal method to insert into a subtree.
48   * x is the item to insert.
49   * t is the node that roots the subtree.
50   * Set the new root of the subtree.
51   */
52 void insert(const Comparable& x, BinaryNode*& t) {
53     if (t == nullptr)
54       t = new BinaryNode{x, nullptr, nullptr};
55     else {
56       if (x < t->element)
57         insert(x, t->left);
58       else if (t->element < x)
59         insert(x, t->right);
60       else; // Duplicate; do nothing
61     }
62 }
```

# BST OPERATIONS

---

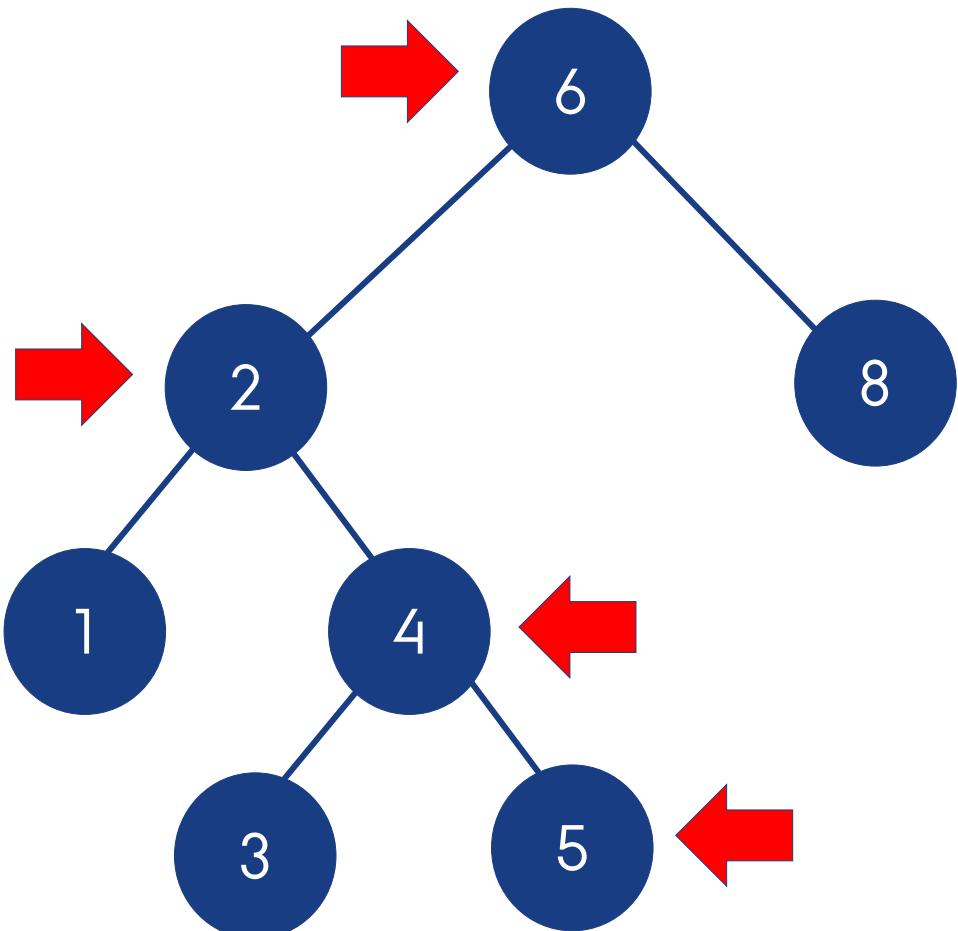
Deletion is **complicated** and has *three cases* to handle:

1. If the node has **no children**, delete it
2. If the node has **one child**, adjust the parent and delete the node
3. If the node has **two children**, replace the node with the smallest key in the right subtree

Let us see an example of how each case is handled.

# BST OPERATIONS

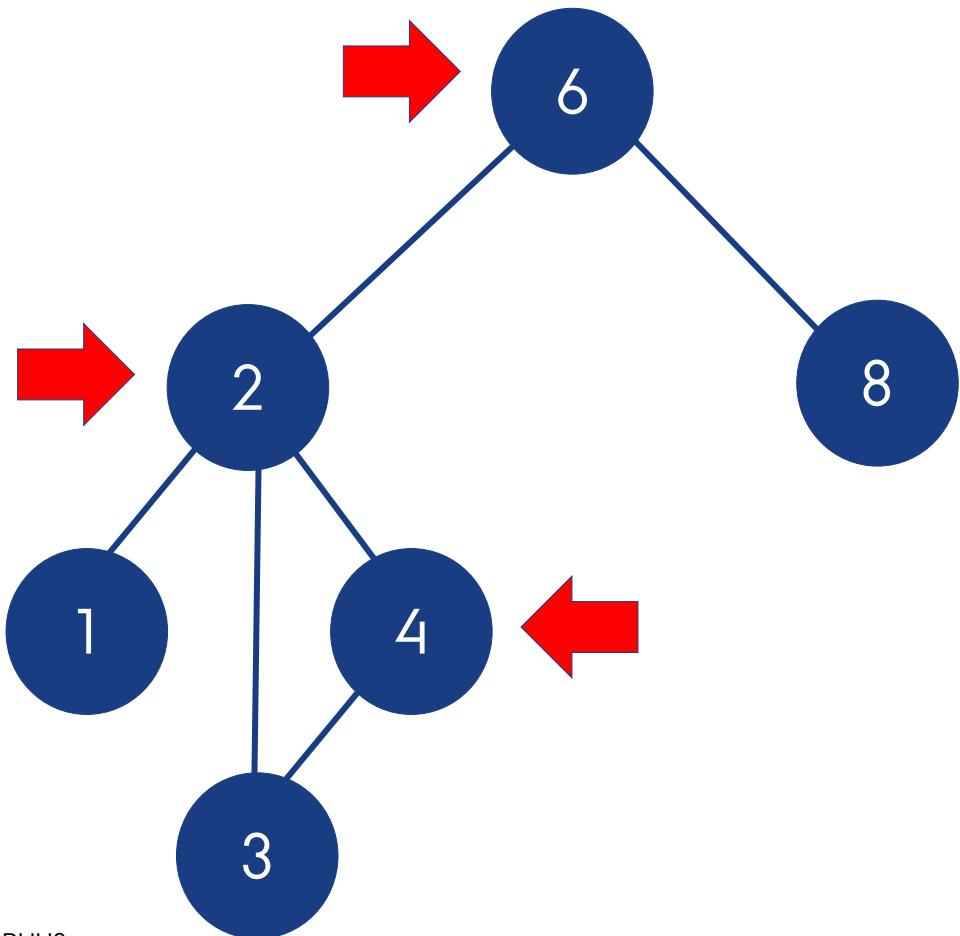
Deletion starts from root, following the **order** until item found ( $x = 5$ ).



```
binary_search_tree.h
64 /**
65  * Internal method to remove from a subtree.
66  * x is the item to remove.
67  * t is the node that roots the subtree.
68  * Set the new root of the subtree.
69 */
70 void remove(const Comparable& x, BinaryNode*& t) {
71     if (t == nullptr)
72         return; // Item not found; do nothing
73     if (x < t->element)
74         remove(x, t->left);
75     else if (t->element < x)
76         remove(x, t->right);
77     else if (t->left != nullptr && t->right != nullptr) // Two children
78     {
79         t->element = findMin(t->right)->element;
80         remove(t->element, t->right);
81     } else {
82         BinaryNode*oldNode = t;
83         t = (t->left != nullptr) ? t->left : t->right;
84         delete oldNode;
85     }
86 }
```

# BST OPERATIONS

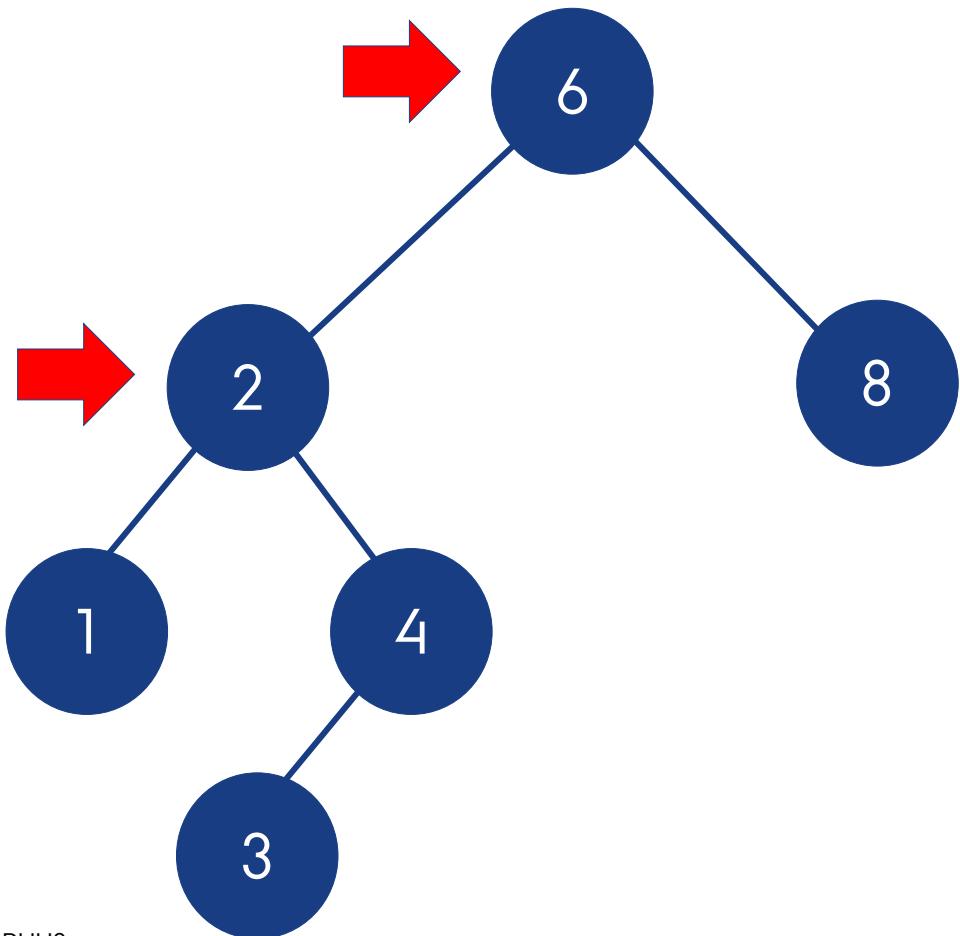
Deletion starts from root, following the **order** until item found ( $x = 4$ ).



```
binary_search_tree.h
64 /**
65  * Internal method to remove from a subtree.
66  * x is the item to remove.
67  * t is the node that roots the subtree.
68  * Set the new root of the subtree.
69 */
70 void remove(const Comparable& x, BinaryNode* &t) {
71     if (t == nullptr)
72         return; // Item not found; do nothing
73     if (x < t->element)
74         remove(x, t->left);
75     else if (t->element < x)
76         remove(x, t->right);
77     else if (t->left != nullptr && t->right != nullptr) // Two children
78     {
79         t->element = findMin(t->right)->element;
80         remove(t->element, t->right);
81     } else {
82         BinaryNode*oldNode = t;
83         t = (t->left != nullptr) ? t->left : t->right;
84         delete oldNode;
85     }
86 }
```

# BST OPERATIONS

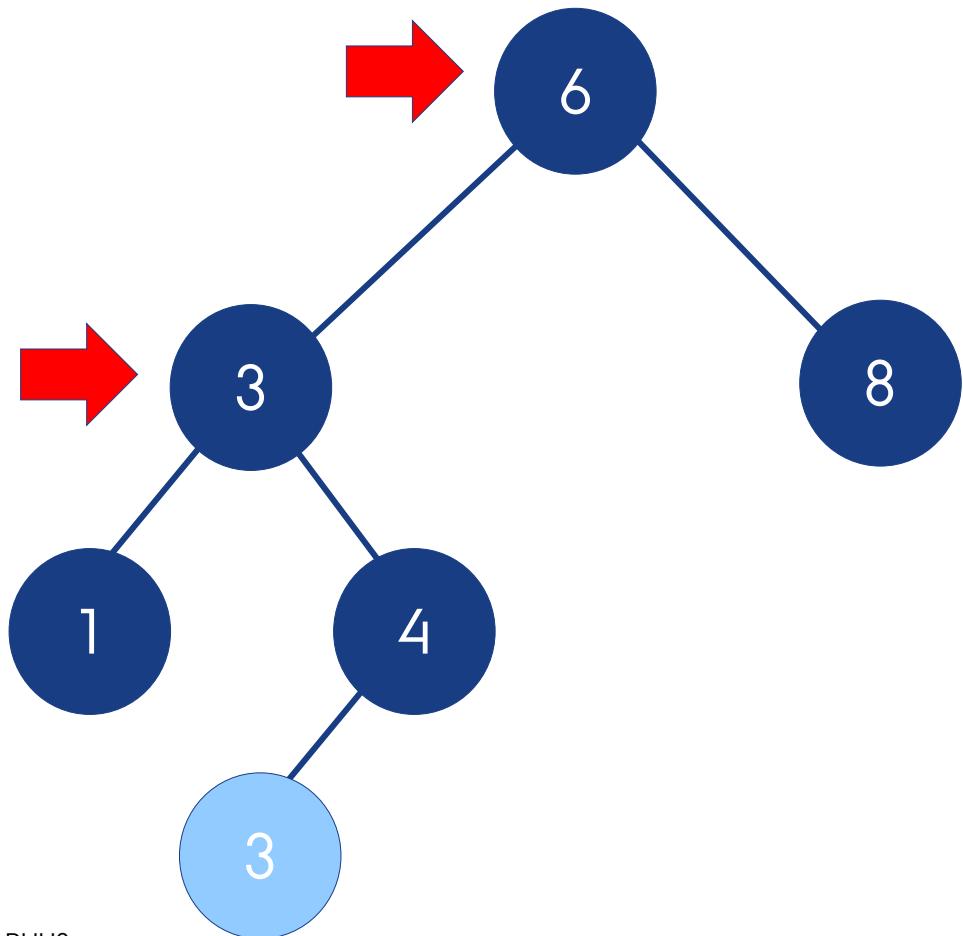
Deletion starts from root, following the **order** until item found ( $x = 2$ ).



```
binary_search_tree.h
64 /**
65  * Internal method to remove from a subtree.
66  * x is the item to remove.
67  * t is the node that roots the subtree.
68  * Set the new root of the subtree.
69 */
70 void remove(const Comparable& x, BinaryNode* &t) {
71     if (t == nullptr)
72         return; // Item not found; do nothing
73     if (x < t->element)
74         remove(x, t->left);
75     else if (t->element < x)
76         remove(x, t->right);
77     else if (t->left != nullptr && t->right != nullptr) // Two children
78     {
79         t->element = findMin(t->right)->element;
80         remove(t->element, t->right);
81     } else {
82         BinaryNode*oldNode = t;
83         t = (t->left != nullptr) ? t->left : t->right;
84         delete oldNode;
85     }
86 }
```

# BST OPERATIONS

Deletion starts from root, following the **order** until item found ( $x = 2$ ).



```
binary_search_tree.h
64  /**
65   * Internal method to remove from a subtree.
66   * x is the item to remove.
67   * t is the node that roots the subtree.
68   * Set the new root of the subtree.
69   */
70 void remove(const Comparable& x, BinaryNode*&t) {
71     if (t == nullptr)
72         return; // Item not found; do nothing
73     if (x < t->element)
74         remove(x, t->left);
75     else if (t->element < x)
76         remove(x, t->right);
77     else if (t->left != nullptr && t->right != nullptr) // Two children
78     {
79         t->element = findMin(t->right)->element;
80         remove(t->element, t->right);
81     } else {
82         BinaryNode*oldNode = t;
83         t = (t->left != nullptr) ? t->left : t->right;
84         delete oldNode;
85     }
86 }
```

# REFLECTION

---

1. Propose an argument for why the time complexity of binary search is better than  $O(n)$ ?
2. Show by drawing how a BST looks after insert of the following elements (in order):  
5, 1, 4, 10, 19, 20, 8, 7
3. Show by drawing  
how the value 4  
is removed from  
the following BST:

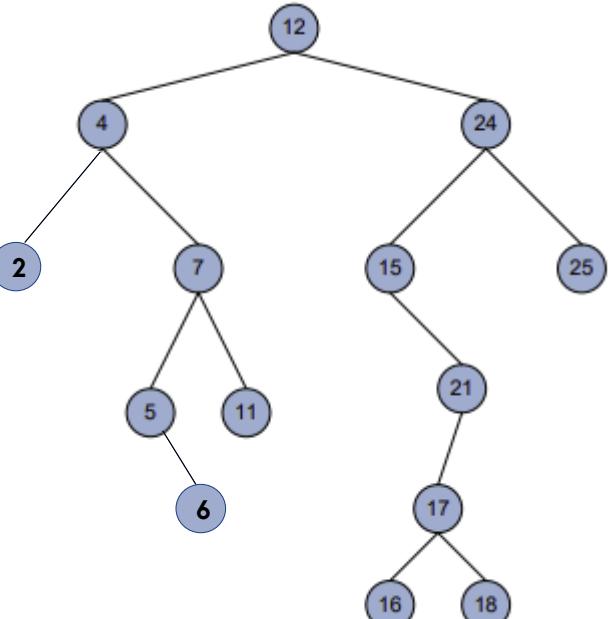


Photo by [Anthony Tran](#) on [Unsplash](#)

# BST OPERATIONS

---

*What about the complexity of the operations?*

All these operations start from the root and reach an **intermediate node**, so they run in  $O(h)$  for a node of height  $h$ .

In average, the height of a node is  $O(\log N)$ , so all operations (search, insert, remove) are  $O(\log N)$  in average case.

# REFLECTION

---

1. All BST operations are  $O(\log n)$  average case.  
What are their complexities in worst case?
2. Comparing a sorted array and a BST, when would you use one over the other?



# AGENDA

---

- ✓ Binary Search Trees (BSTs)
- ✓ BST operations and their complexity
- Balanced Trees
- AVL Trees and rotations

# BALANCED TREES

---

→ *We need to work harder to obtain worst-case logarithmic time!*

A solution is to keep the binary search tree *balanced*. There are different approaches: **AVL tree**, Red-black Tree, ...

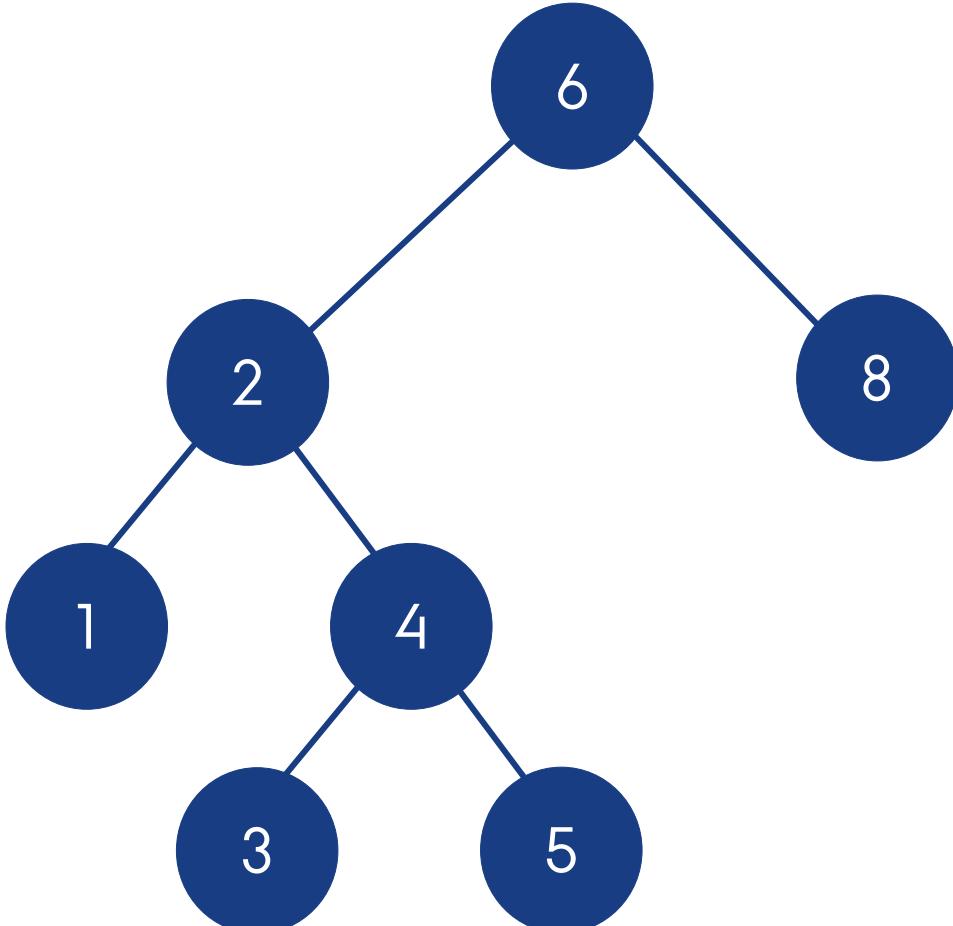
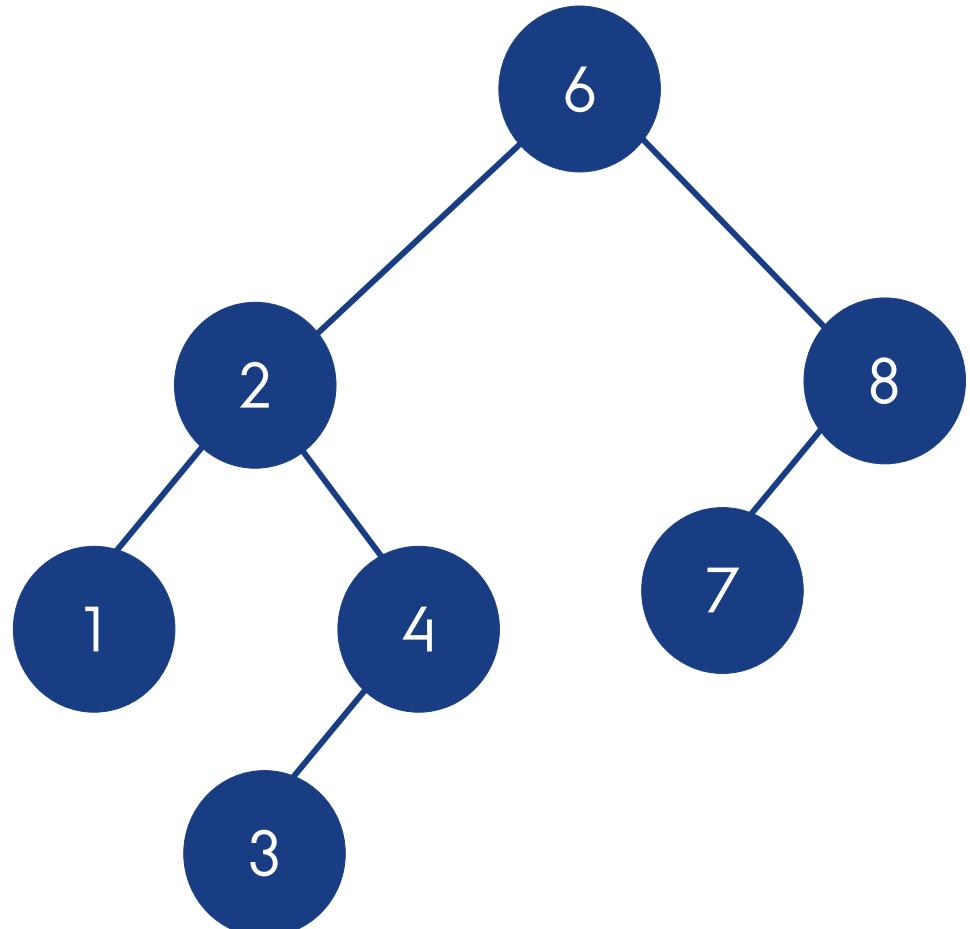
AVL Trees balance property: tolerate **difference in height of at most one between left and right subtrees of any node**

The main 3 operations are performed in  **$O(\log N)$**  (worst-case)

# BALANCED TREES

---

The left is AVL, the right one **violates** the **balance** property. *Why?*



# AGENDA

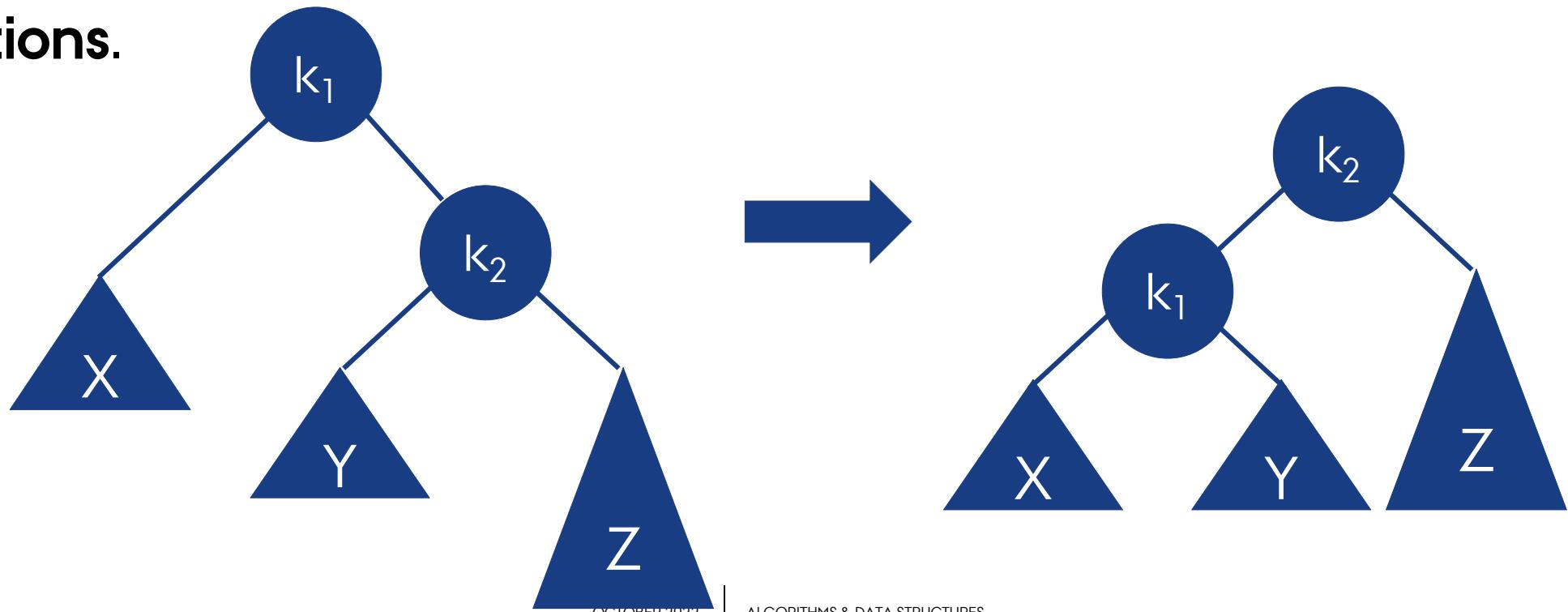
---

- ✓ Binary Search Trees (BST)
- ✓ BST operations and their complexity
- ✓ Balanced Trees
- AVL Trees and rotations

# AVL TREES

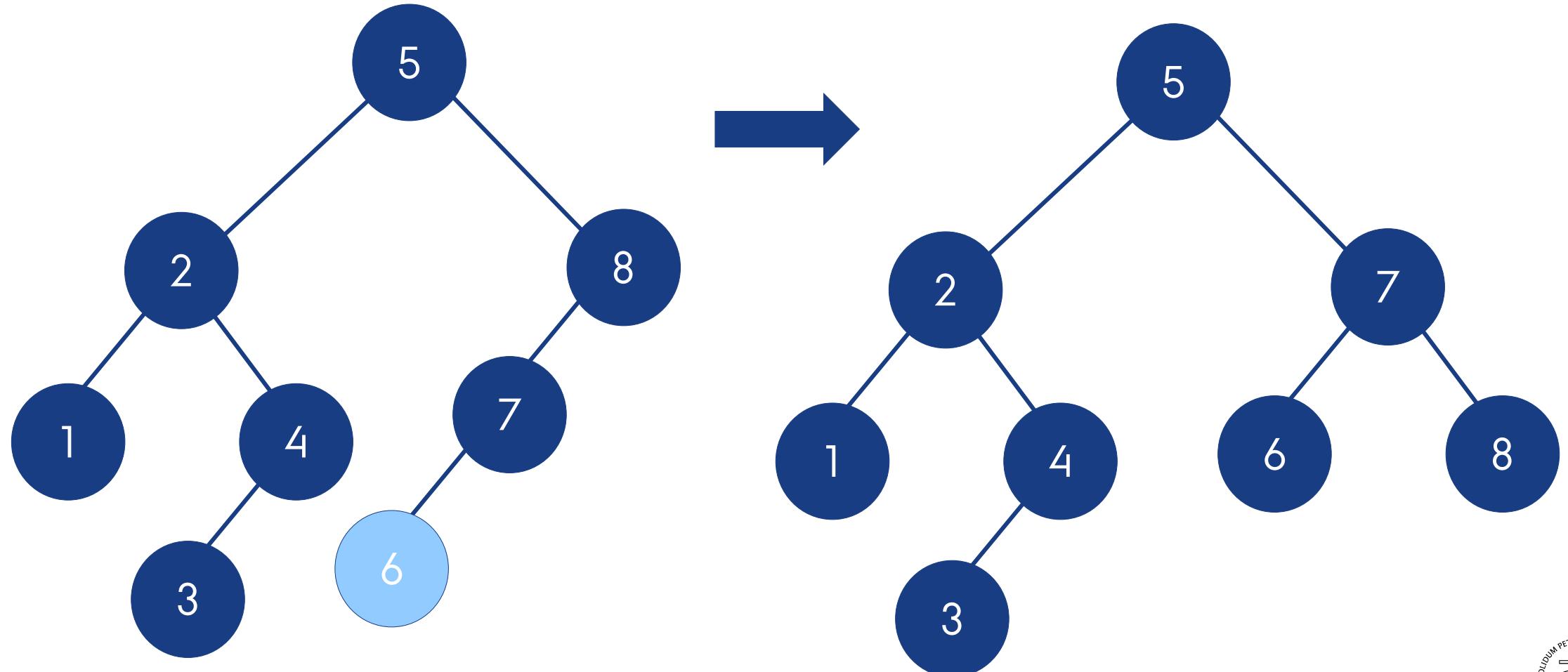
---

Only **insertions** and **deletions** can violate the **balance** property.  
For example, insertion can insert a node in a **longer** subtree and increase the difference to 2 (*outer* imbalance).  
We can modify these to **restore** the property by using **single rotations**.



# AVL TREES

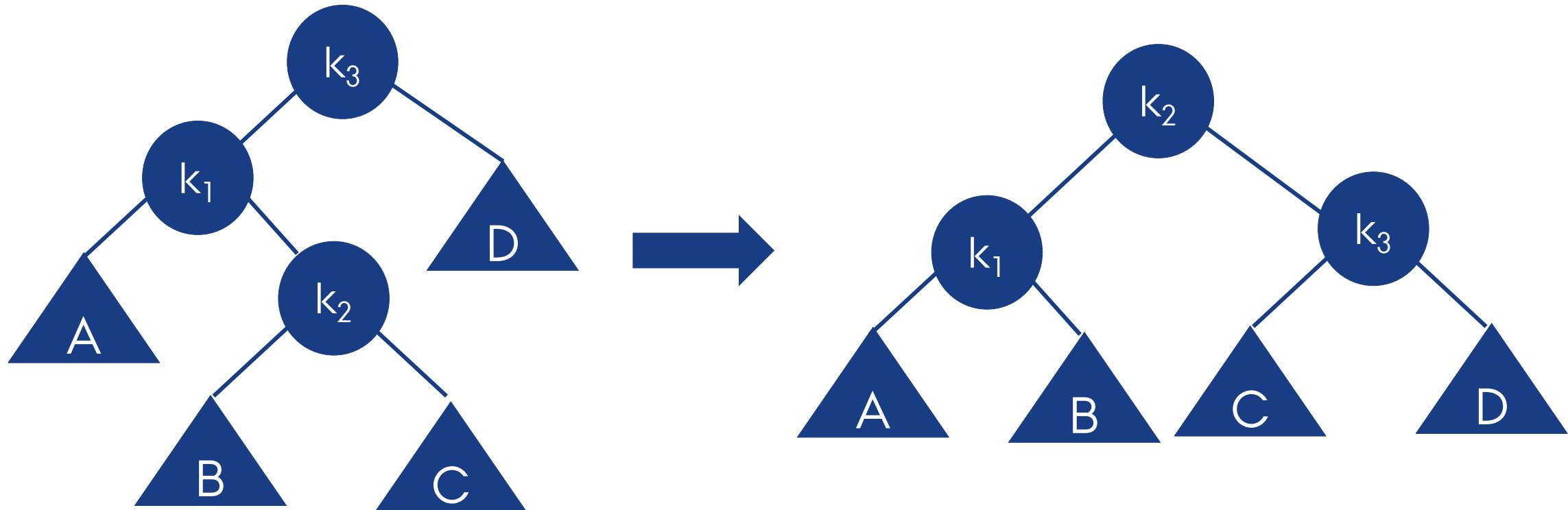
Let us see one example of **insertion** with **rotation** ( $x = 6$ ).



# AVL TREES

---

Insertions need to enforce the **balance** bottom-up. A more complicated case (*inner imbalance*) needs **double rotations**.



# AVL TREES

---

## Templated AVL tree:

- **Augment** nodes with height
- Modified insertion and deletion will balance a tree at the **end of recursion**
- **Balance** operation will test for the two cases (*inner or outer* imbalance) and perform **single** or **double** rotations to restore balance property

Check **repository** for details.

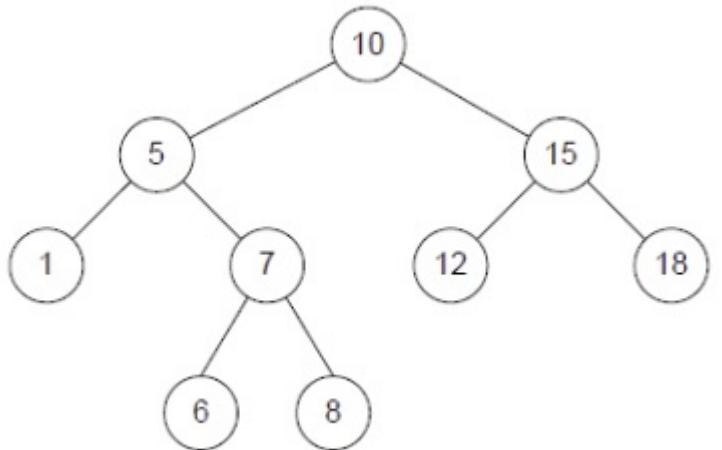
avl\_tree.h

```
115     static const int ALLOWED_IMBALANCE = 1;  
116  
117     // Assume t is balanced or within one of being balanced  
118     void balance(AvlNode * &t) {  
119         if (t == nullptr)  
120             return;  
121  
122         if (height(t->left) - height(t->right) > ALLOWED_IMBALANCE)  
123             if (height(t->left->left) >= height(t->left->right))  
124                 rotateWithLeftChild(t);  
125             else  
126                 doubleWithLeftChild(t);  
127             else if (height(t->right) - height(t->left) > ALLOWED_IMBALANCE)  
128                 if (height(t->right->right) >= height(t->right->left))  
129                     rotateWithRightChild(t);  
130                 else  
131                     doubleWithRightChild(t);  
132  
133         t->height = max(height(t->left), height(t->right)) + 1;  
134     }
```

# REFLECTION

---

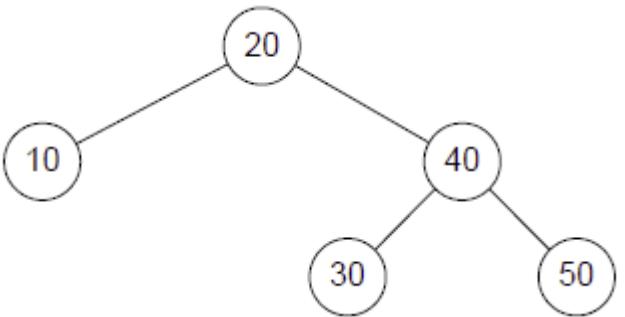
1. Insert 9 in the AVL tree below. Does it cause any imbalances, and if so which rotations do you use to resolve. How does the resulting AVL tree look?



# REFLECTION

---

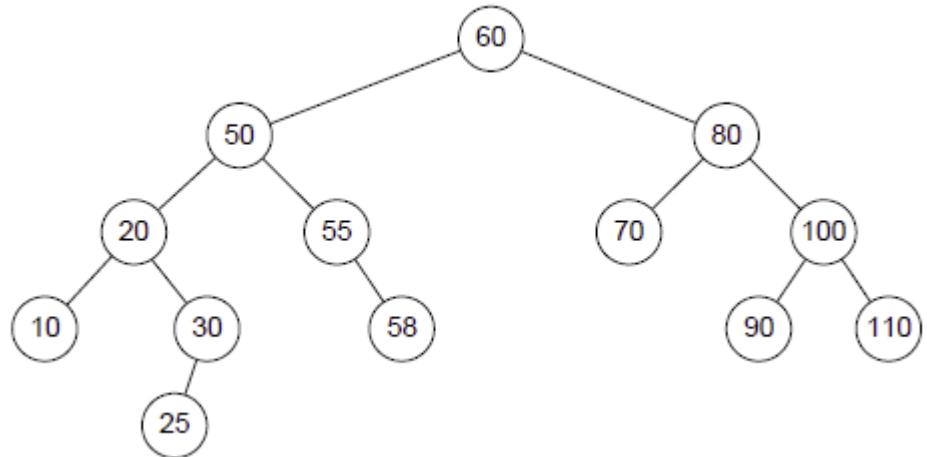
2. Insert 25 in the AVL tree below. Does it cause any imbalances, and if so which rotations do you use to resolve. How does the resulting AVL tree look?



# REFLECTION

---

3. How does this AVL tree look after deletion of the value 10?



4. Build an AVL tree by inserting the following values in the order stated: 15, 20, 24, 10, 13, 7, 30, 36, 25

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>



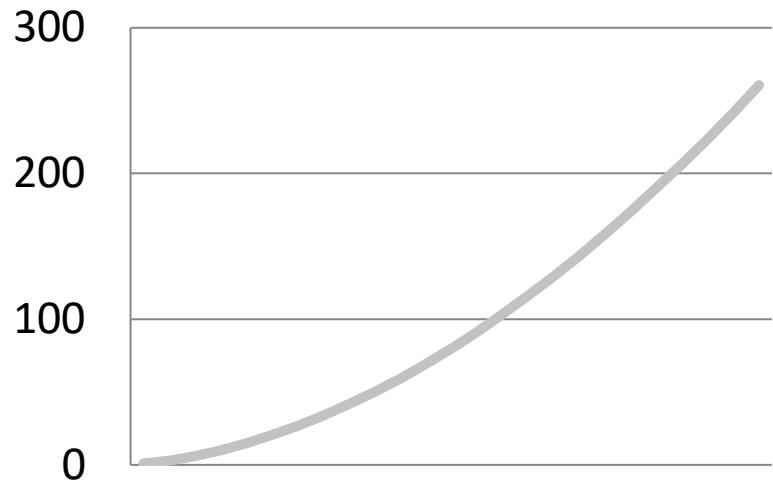
Photo by [Anthony Tran](#) on [Unsplash](#)

# GRAPHS

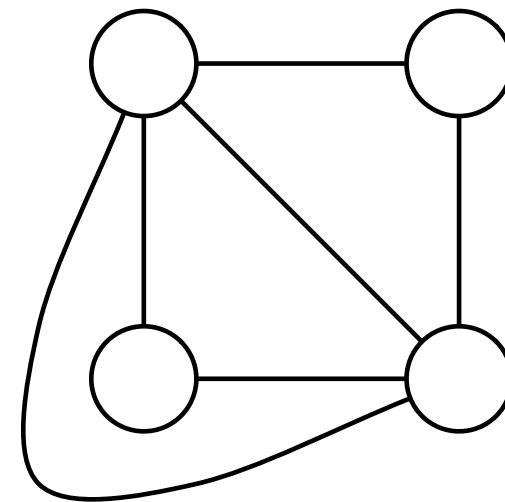
# GRAPHS

---

Not this:



This:



# AGENDA

---

- Definitions and representations
  - Depth-First Search (DFS) and applications
  - Breadth-First Search (BFS) and applications
  - Single-source Shortest-Path Algorithms

# GRAPHS

---

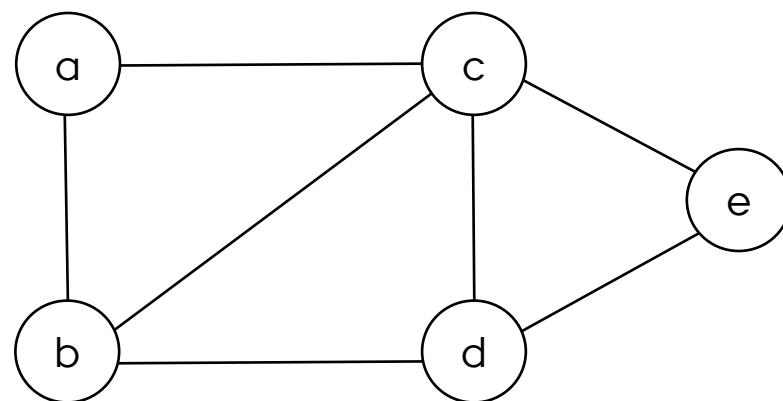
## Definition:

A *graph*  $G = (V, E)$  consists of a set of *vertices*  $V$ , and a set of *edges*  $E$ . Each edge is a pair  $(v, w)$ , where  $v, w \in V$ . We denote by  $|V|$  &  $|E|$  the **cardinality** of the sets of vertices and edges.

## Example:

$$V = \{a, b, c, d, e\}$$

$$E = \{(a,b), (a,c), (b,c), (b,d), (c,d), (c,e), (d,e)\}$$



# GRAPH DEFINITION

---

A graph,  $G$ , is a pair ( $V, E$ ) of sets

$V$  is the set of vertices

$$V = \{a, b, c, d\}$$

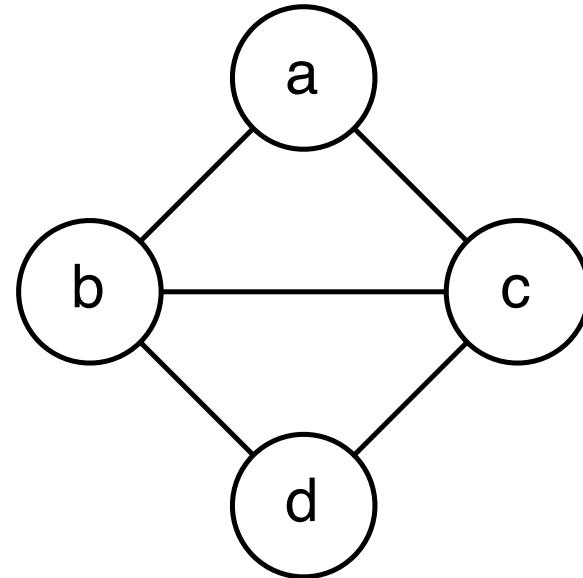
$V$  is non-empty

$E$  is the set of edges

$$E = \{ \{a,b\}, \{a,c\}, \{b,c\}, \{b,d\}, \{c,d\} \}$$

Elements in  $E$  are sets containing two vertices from  $V$

- Note:  $E$  is a set of sets, not a set of pairs

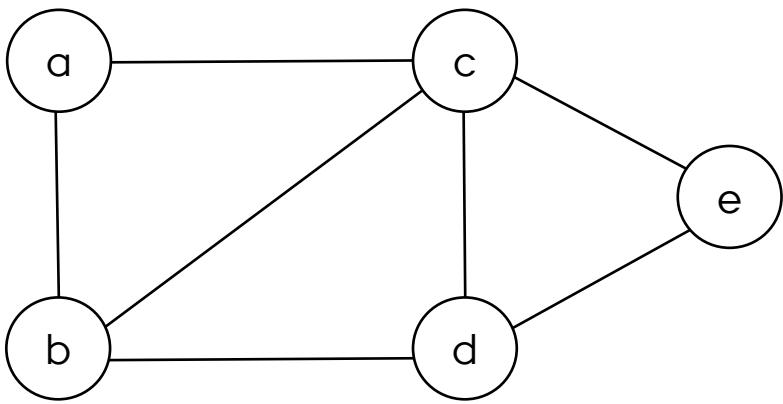


# GRAPHS

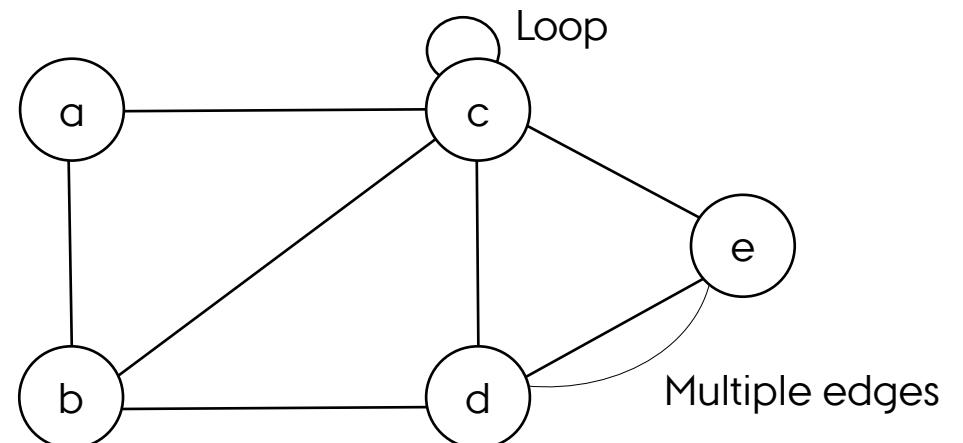
---

## Definition:

Given an edge  $(a, b)$ , we call  $a$  and  $b$  the *ends* and they are **adjacent**. The edge is *incident* to vertices  $a$  and  $b$ . A graph is simple if it does not have **loops** or **multiple edges**.



Simple graph



Multiple edges



# DEGREES

---

**Definition:** The *degree* of a vertex  $v$ , denoted by  $D(v)$  is the number of edges incident to  $v$ .

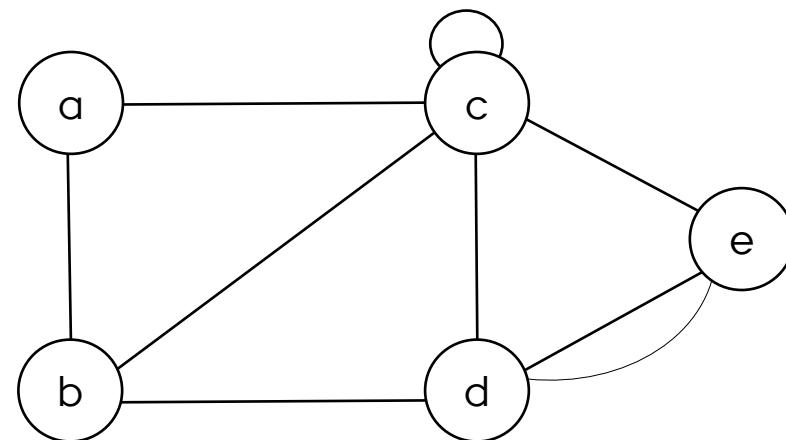
**Handshaking lemma:** For all graphs  $G$ , we have that the *sum of degrees* of all vertices is  $2|E|$ .

**Example:**

$$D(a) = 2 \quad D(d) = 4$$

$$D(b) = 3 \quad D(e) = 3$$

$$D(c) = 6$$



# PATHS AND CYCLES

---

## Definition:

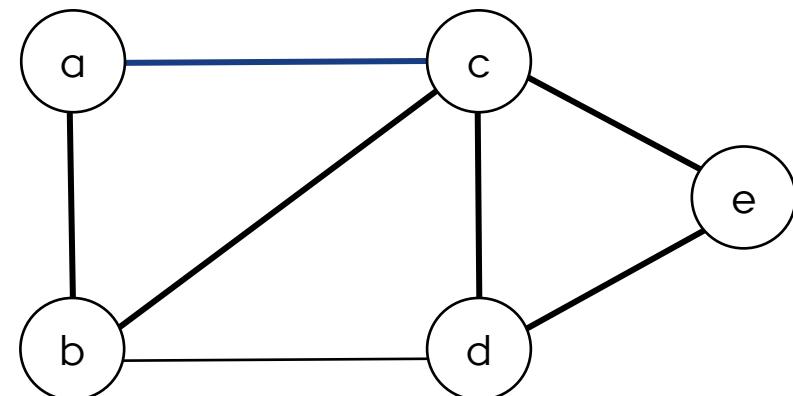
A *path* is a sequence of vertices  $(w_1, w_2, w_3, \dots, w_N)$  such that  $(w_i, w_{i+1}) \in E$  for  $1 \leq i < N$ . The length of a path is the number of edges on the path, equal to  $N - 1$ . A *simple path* has no **repetition of vertices**. A *cycle* is a path such that  $w_1 = w_N$ .

## Example:

$(a, b, c, d, e, c)$  is a **path**

$(a, b, c, d, e)$  is a **simple path**

$(a, b, c, d, e, c, a)$  is a **cycle**

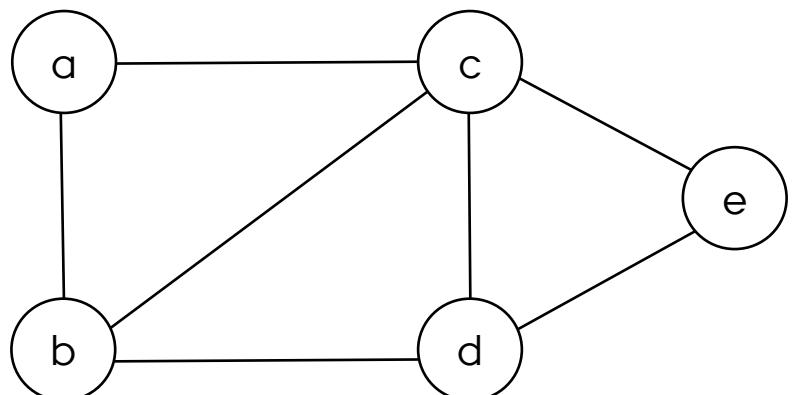


# CONNECTIVITY

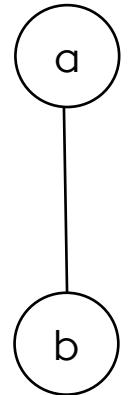
---

## Definition:

A graph  $G$  is connected if there is a path from  $u$  to  $v$  for every pair of vertices  $u$  and  $v$ . When  $G$  is not connected, we can **partition**  $G$  in *connected components*. Vertices  $u$  and  $v$  are in the **same** connected component if there is a **path** from  $u$  to  $v$ .



Connected graph



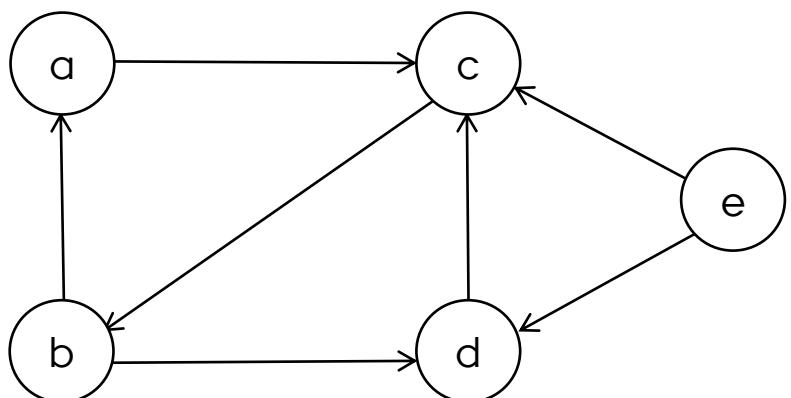
Graph with 2 connected components

# CONNECTIVITY

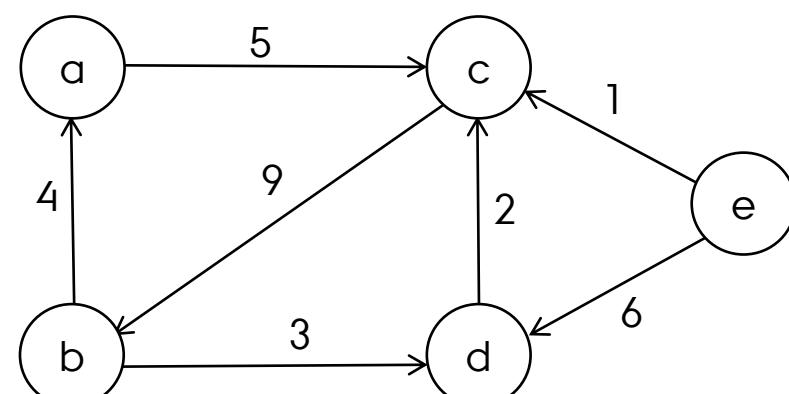
---

## Definition:

If the edges are ordered, then the graph is **directed** (*digraph*). Sometimes an edge has a **third** component, known as either a *weight* or a *cost*.



Directed graph



Weighted directed graph

# GRAPH ALGORITHMS

---

Graphs are **abstract** structures which can **model** real-world **problems**. For example, a graph can represent connections between cities and roads or a computer network.

We study graph algorithm to solve certain problems:

- **Shortest path**: given a set of cities and the distance between them, determine the shortest path between cities A and B.
- **Minimum spanning tree (next weeks)**: given a set of computers, where every pair of computers can be connected by fiber, find an interconnection network which minimizes the amount of fiber.

# GRAPH REPRESENTATIONS

---

The complexity of algorithms to solve problems **modeled** as graph problems strongly depends on the **internal representation** of the graph.

There are two canonical representations: **adjacency matrix** and **adjacency lists**.

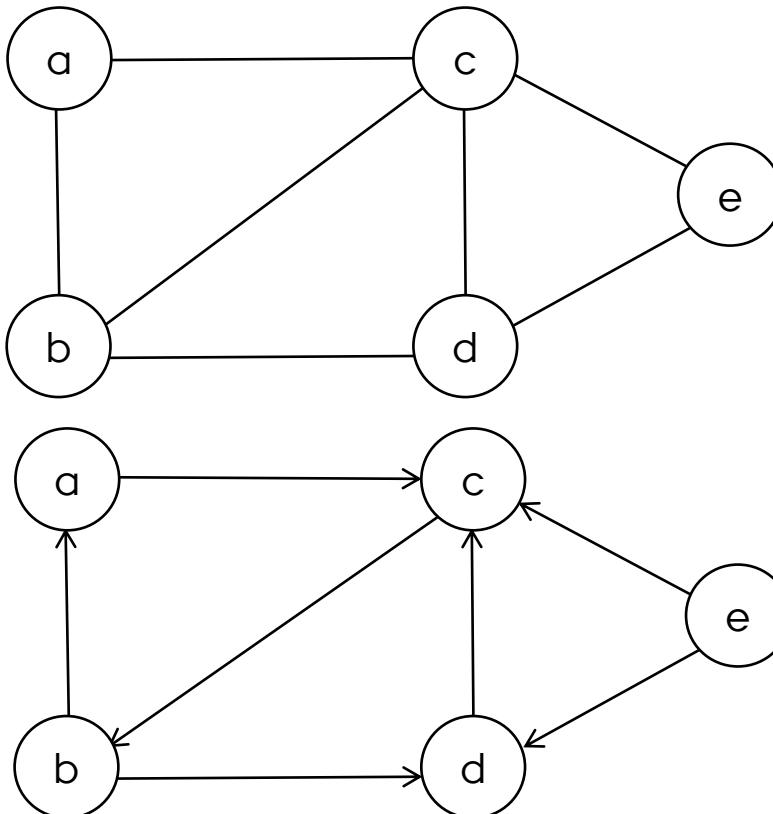
→ *Deciding what's best for some specific algorithm depends on the nature of the operations that define the algorithm complexity.*

# ADJACENCY MATRIX

Let  $G = (V, E)$  a simple graph. The **adjacency matrix** of  $G$  is a square matrix  $A$  of dimension  $|V|$  such that  $A[i,j] = 1$  iff  $(i,j) \in V$ .

Easy to **verify** if  $(u,v) \in V$ , but **space** complexity  $\Theta(V^2)$ .

Adequate to represent **dense** graphs.



	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	1	0
c	1	1	0	1	1
d	0	1	1	0	1
e	0	0	1	1	0

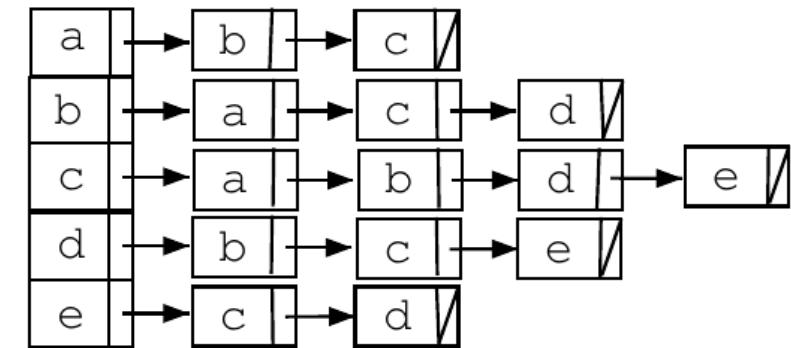
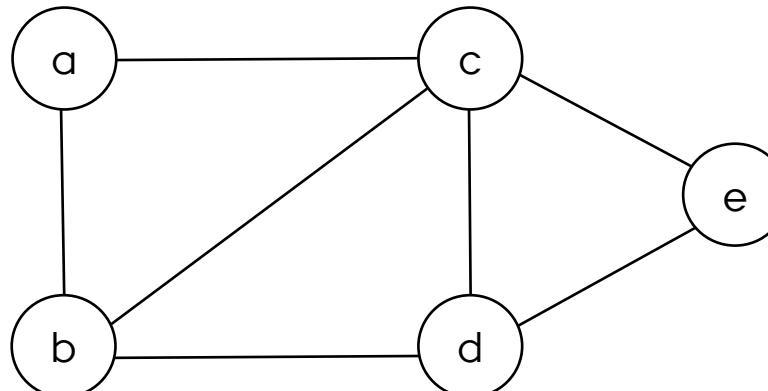
  

	a	b	c	d	e
a	0	0	1	0	0
b	1	0	0	1	0
c	0	1	0	0	0
d	0	0	1	0	0
e	0	0	1	1	0

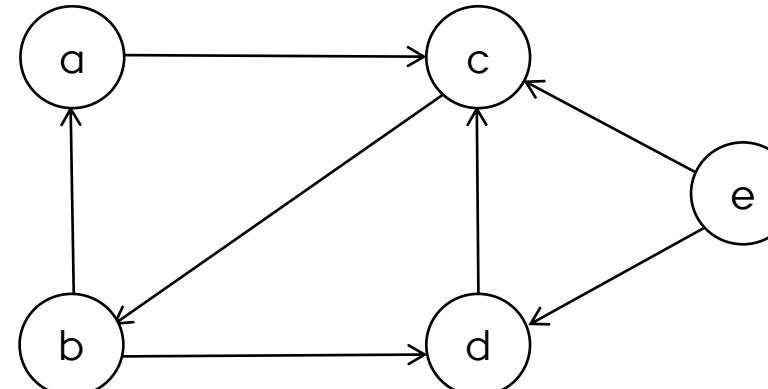
# ADJACENCY LISTS

Let  $G = (V, E)$  a simple graph. Every vertex  $v$  has a **linked list** of the vertices **adjacent** to  $v$ . Vertices can be stored in any order.

Easy to **enumerate neighborhood**, but **space**  $\Theta(|V| + |E|)$ .



Adequate to represent **sparse** graphs.



# GRAPH TRAVERSAL

---

Graphs are data structures more complex than lists, arrays and binary trees, hence methods to **explore/traverse** (*directed or undirected*) graphs are needed.

Different *approaches* for **searching** in graphs:

1. **DFS**: Depth-First Search
2. **BFS**: Breadth-First Search

We can obtain information about the **structure** of the graph that can be useful to design efficient algorithms for certain problems.

# AGENDA

---

- ✓ Definitions and representations
- Depth-First Search (DFS) and applications
- Breadth-First Search (BFS) and applications
- Single-source Shortest-Path Algorithms

# DEPTH-FIRST SEARCH

---

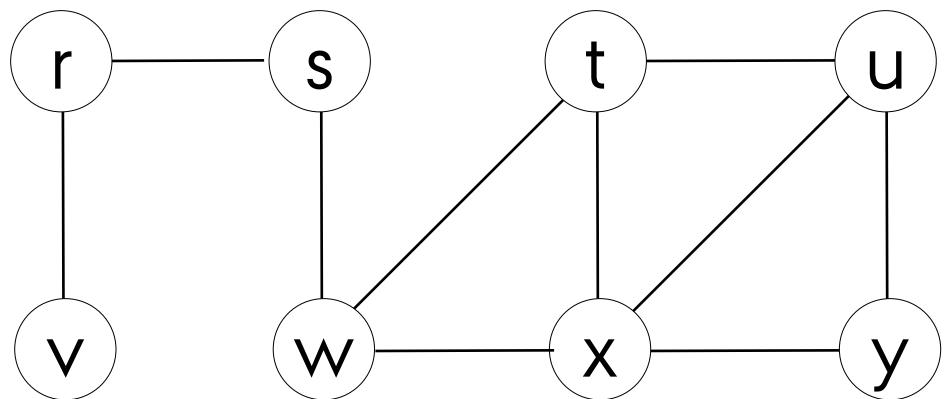
Consists in an approach that explores the graph at **depth**, which means that adjacent vertices are visited as **soon as they are discovered**.

The algorithm also computes other **useful** information while it explores the graph:

- A depth-first forest that stores the **predecessor** in the search
- A **sorting** of the nodes starting from the **source vertex**

# DEPTH-FIRST SEARCH

DFS explores the **neighborhood** before finishing a vertex. Let us see the behavior starting from  $t$ .

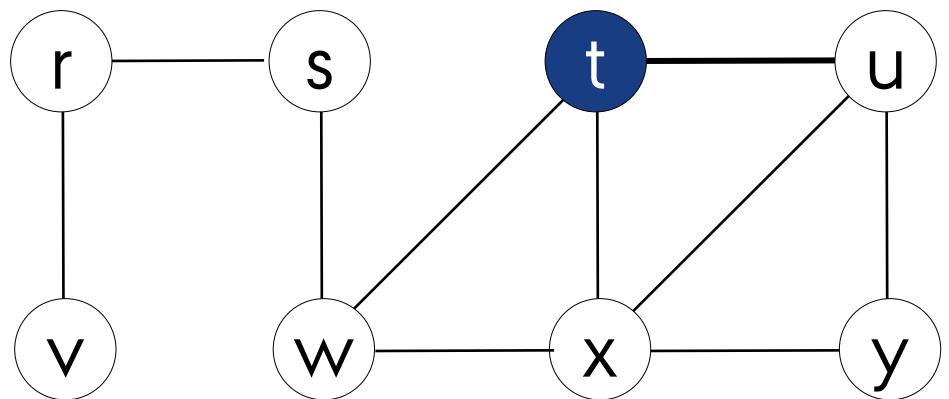


```

1  using namespace std;
2  #include <vector>
3  #include <queue>
4  #include <stack>
5  #include <iostream>
6  #include <climits>
7
8  #define INFINITY INT_MAX
9
10 class Graph {
11     private:
12         vector<vector<int>> adj;
13
14     void dfs(int v, vector<bool>& visited, vector<int>& path, stack<int>& sorting) {
15         visited[v] = true;
16         for (auto w : adj[v]) {
17             if (!visited[w]) {
18                 path[w] = v;
19                 dfs(w, visited, path, sorting);
20             }
21         }
22         sorting.push(v);
23     }
24
25     public:
26     Graph(int vertices = 0) : adj(vertices) { }
27
28     void addEdge(int u, int v);
29     void addDirectedEdge(int u, int v);
30     void dfs(int v, vector<int>& path);
31     void bfs(int s, vector<int>& path, vector<int>& dist);
32     void topsort(int v);
33     int components();
34     void print();
35 };
36
37 void Graph::dfs(int vertex, vector<int>& path) {
38     stack<int> sorting;
39     vector<bool> visited(adj.size());
40
41     for (int v = 0; v < adj.size(); ++v) {
42         visited[v] = false;
43     }
44     dfs(vertex, visited, path, sorting);
45 }
  
```

# DEPTH-FIRST SEARCH

DFS explores the **neighborhood** before finishing a vertex. Let us see the behavior starting from  $t$ .

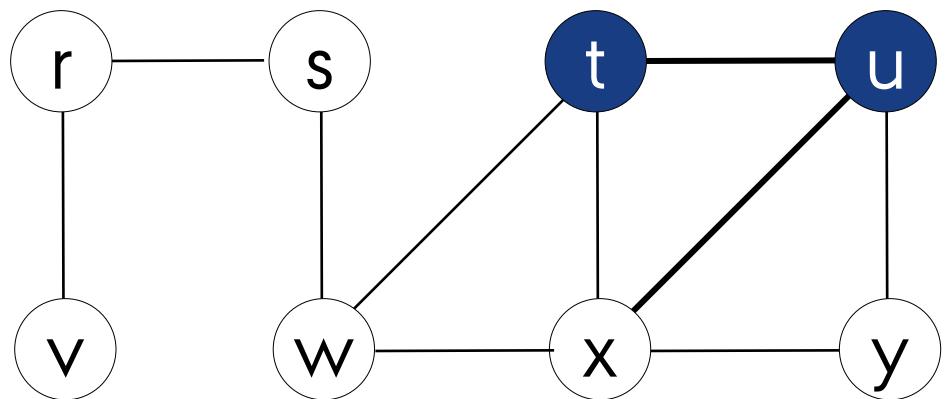


```

1  using namespace std;
2  #include <vector>
3  #include <queue>
4  #include <stack>
5  #include <iostream>
6  #include <climits>
7
8  #define INFINITY INT_MAX
9
10 class Graph {
11 private:
12     vector<vector<int>> adj;
13
14     void dfs(int v, vector<bool>& visited, vector<int>& path, stack<int>& sorting) {
15         visited[v] = true;
16         for (auto w : adj[v]) {
17             if (!visited[w]) {
18                 path[w] = v;
19                 dfs(w, visited, path, sorting);
20             }
21         }
22         sorting.push(v);
23     }
24
25 public:
26     Graph(int vertices = 0) : adj(vertices) { }
27
28     void addEdge(int u, int v);
29     void addDirectedEdge(int u, int v);
30     void dfs(int v, vector<int>& path);
31     void bfs(int s, vector<int>& path, vector<int>& dist);
32     void topsort(int v);
33     int components();
34     void print();
35 };
36
37 void Graph::dfs(int vertex, vector<int>& path) {
38     stack<int> sorting;
39     vector<bool> visited(adj.size());
40
41     for (int v = 0; v < adj.size(); ++v) {
42         visited[v] = false;
43     }
44     dfs(vertex, visited, path, sorting);
45 }
```

# DEPTH-FIRST SEARCH

DFS explores the **neighborhood** before finishing a vertex. Let us see the behavior starting from  $t$ .

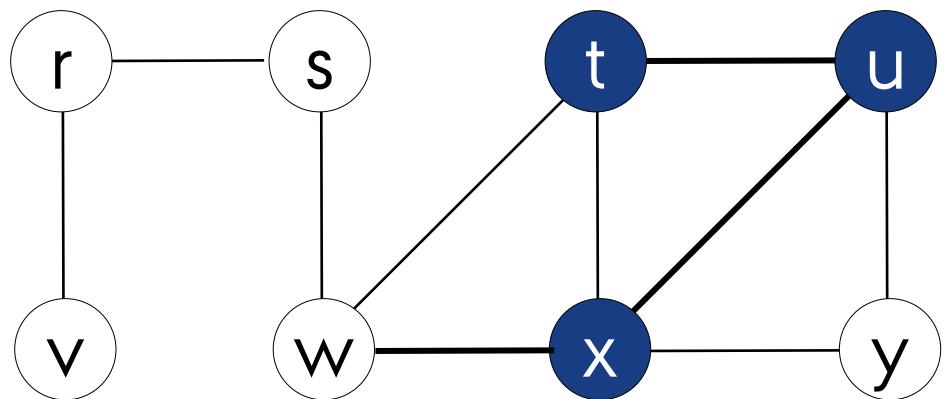


```

1  using namespace std;
2  #include <vector>
3  #include <queue>
4  #include <stack>
5  #include <iostream>
6  #include <climits>
7
8  #define INFINITY INT_MAX
9
10 class Graph {
11 private:
12     vector<vector<int>> adj;
13
14     void dfs(int v, vector<bool>& visited, vector<int>& path, stack<int>& sorting) {
15         visited[v] = true;
16         for (auto w : adj[v]) {
17             if (!visited[w]) {
18                 path[w] = v;
19                 dfs(w, visited, path, sorting);
20             }
21         }
22         sorting.push(v);
23     }
24
25 public:
26     Graph(int vertices = 0) : adj(vertices) { }
27
28     void addEdge(int u, int v);
29     void addDirectedEdge(int u, int v);
30     void dfs(int v, vector<int>& path);
31     void bfs(int s, vector<int>& path, vector<int>& dist);
32     void topsort(int v);
33     int components();
34     void print();
35 };
36
37 void Graph::dfs(int vertex, vector<int>& path) {
38     stack<int> sorting;
39     vector<bool> visited(adj.size());
40
41     for (int v = 0; v < adj.size(); ++v) {
42         visited[v] = false;
43     }
44     dfs(vertex, visited, path, sorting);
45 }
```

# DEPTH-FIRST SEARCH

DFS explores the **neighborhood** before finishing a vertex. Let us see the behavior starting from  $t$ .

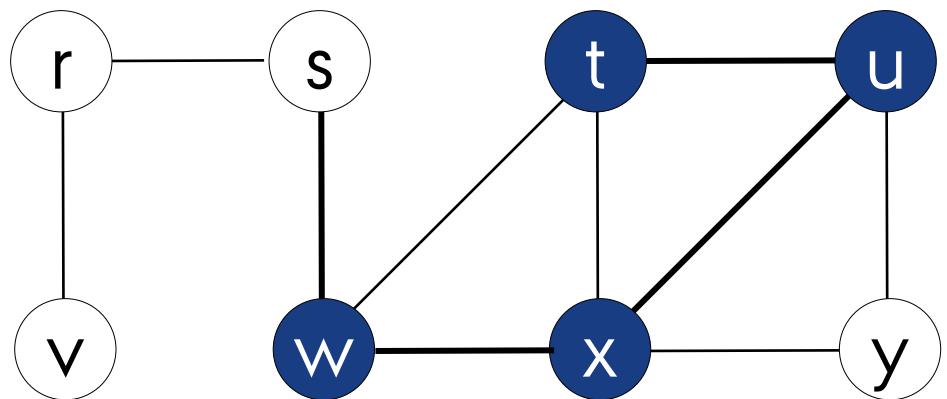


```

1  using namespace std;
2  #include <vector>
3  #include <queue>
4  #include <stack>
5  #include <iostream>
6  #include <climits>
7
8  #define INFINITY INT_MAX
9
10 class Graph {
11 private:
12     vector<vector<int>> adj;
13
14     void dfs(int v, vector<bool>& visited, vector<int>& path, stack<int>& sorting) {
15         visited[v] = true;
16         for (auto w : adj[v]) {
17             if (!visited[w]) {
18                 path[w] = v;
19                 dfs(w, visited, path, sorting);
20             }
21         }
22         sorting.push(v);
23     }
24
25 public:
26     Graph(int vertices = 0) : adj(vertices) { }
27
28     void addEdge(int u, int v);
29     void addDirectedEdge(int u, int v);
30     void dfs(int v, vector<int>& path);
31     void bfs(int s, vector<int>& path, vector<int>& dist);
32     void topsort(int v);
33     int components();
34     void print();
35 };
36
37 void Graph::dfs(int vertex, vector<int>& path) {
38     stack<int> sorting;
39     vector<bool> visited(adj.size());
40
41     for (int v = 0; v < adj.size(); ++v) {
42         visited[v] = false;
43     }
44     dfs(vertex, visited, path, sorting);
45 }
```

# DEPTH-FIRST SEARCH

DFS explores the **neighborhood** before finishing a vertex. Let us see the behavior starting from  $t$ .

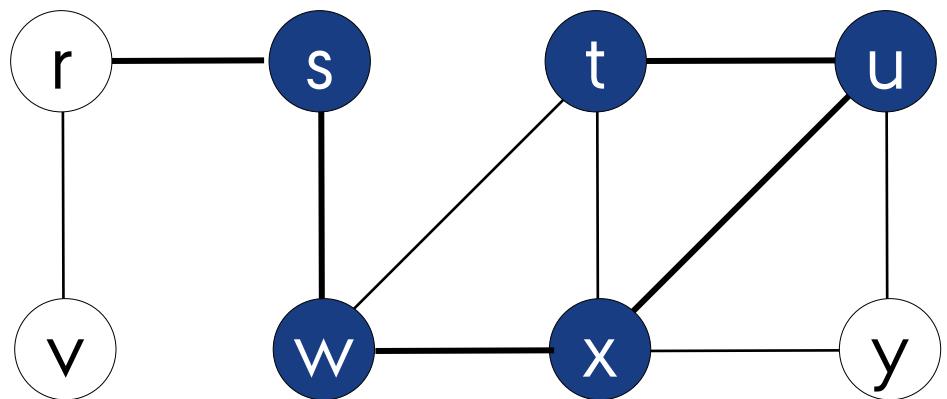


```

1  using namespace std;
2  #include <vector>
3  #include <queue>
4  #include <stack>
5  #include <iostream>
6  #include <climits>
7
8  #define INFINITY INT_MAX
9
10 class Graph {
11 private:
12     vector<vector<int>> adj;
13
14     void dfs(int v, vector<bool>& visited, vector<int>& path, stack<int>& sorting) {
15         visited[v] = true;
16         for (auto w : adj[v]) {
17             if (!visited[w]) {
18                 path[w] = v;
19                 dfs(w, visited, path, sorting);
20             }
21         }
22         sorting.push(v);
23     }
24
25 public:
26     Graph(int vertices = 0) : adj(vertices) { }
27
28     void addEdge(int u, int v);
29     void addDirectedEdge(int u, int v);
30     void dfs(int v, vector<int>& path);
31     void bfs(int s, vector<int>& path, vector<int>& dist);
32     void topsort(int v);
33     int components();
34     void print();
35 };
36
37 void Graph::dfs(int vertex, vector<int>& path) {
38     stack<int> sorting;
39     vector<bool> visited(adj.size());
40
41     for (int v = 0; v < adj.size(); ++v) {
42         visited[v] = false;
43     }
44     dfs(vertex, visited, path, sorting);
45 }
```

# DEPTH-FIRST SEARCH

DFS explores the **neighborhood** before finishing a vertex. Let us see the behavior starting from  $t$ .

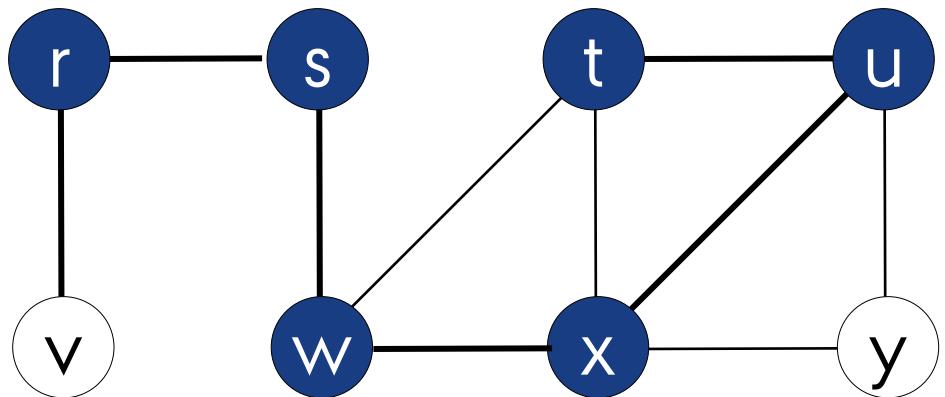


```

1  using namespace std;
2  #include <vector>
3  #include <queue>
4  #include <stack>
5  #include <iostream>
6  #include <climits>
7
8  #define INFINITY INT_MAX
9
10 class Graph {
11 private:
12     vector<vector<int>> adj;
13
14     void dfs(int v, vector<bool>& visited, vector<int>& path, stack<int>& sorting) {
15         visited[v] = true;
16         for (auto w : adj[v]) {
17             if (!visited[w]) {
18                 path[w] = v;
19                 dfs(w, visited, path, sorting);
20             }
21         }
22         sorting.push(v);
23     }
24
25 public:
26     Graph(int vertices = 0) : adj(vertices) { }
27
28     void addEdge(int u, int v);
29     void addDirectedEdge(int u, int v);
30     void dfs(int v, vector<int>& path);
31     void bfs(int s, vector<int>& path, vector<int>& dist);
32     void topsort(int v);
33     int components();
34     void print();
35 };
36
37 void Graph::dfs(int vertex, vector<int>& path) {
38     stack<int> sorting;
39     vector<bool> visited(adj.size());
40
41     for (int v = 0; v < adj.size(); ++v) {
42         visited[v] = false;
43     }
44     dfs(vertex, visited, path, sorting);
45 }
  
```

# DEPTH-FIRST SEARCH

DFS explores the **neighborhood** before finishing a vertex. Let us see the behavior starting from  $t$ .



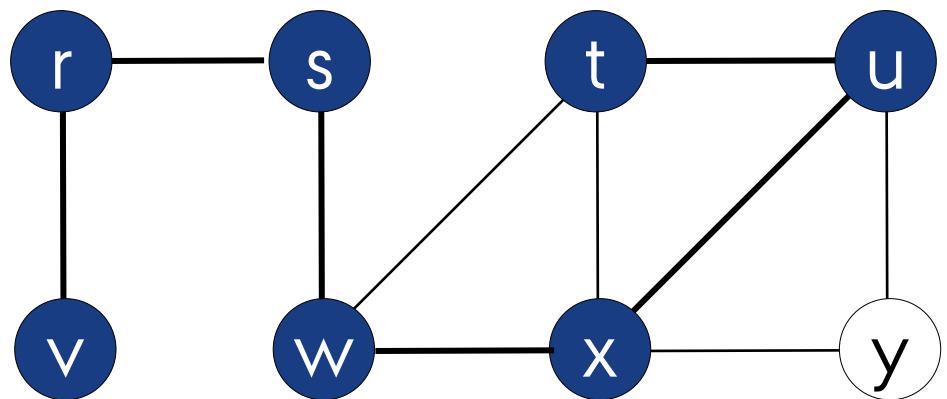
```

1  using namespace std;
2  #include <vector>
3  #include <queue>
4  #include <stack>
5  #include <iostream>
6  #include <climits>
7
8  #define INFINITY INT_MAX
9
10 class Graph {
11     private:
12         vector<vector<int>> adj;
13
14     void dfs(int v, vector<bool>& visited, vector<int>& path, stack<int>& sorting) {
15         visited[v] = true;
16         for (auto w : adj[v]) {
17             if (!visited[w]) {
18                 path[w] = v;
19                 dfs(w, visited, path, sorting);
20             }
21         }
22         sorting.push(v);
23     }
24
25     public:
26     Graph(int vertices = 0) : adj(vertices) { }
27
28     void addEdge(int u, int v);
29     void addDirectedEdge(int u, int v);
30     void dfs(int v, vector<int>& path);
31     void bfs(int s, vector<int>& path, vector<int>& dist);
32     void topsort(int v);
33     int components();
34     void print();
35 };
15  void Graph::dfs(int vertex, vector<int>& path) {
16      stack<int> sorting;
17      vector<bool> visited(adj.size());
18
19      for (int v = 0; v < adj.size(); ++v) {
20          visited[v] = false;
21      }
22      dfs(vertex, visited, path, sorting);
23  }

```

# DEPTH-FIRST SEARCH

DFS explores the **neighborhood** before finishing a vertex. Let us see the behavior starting from  $t$ .

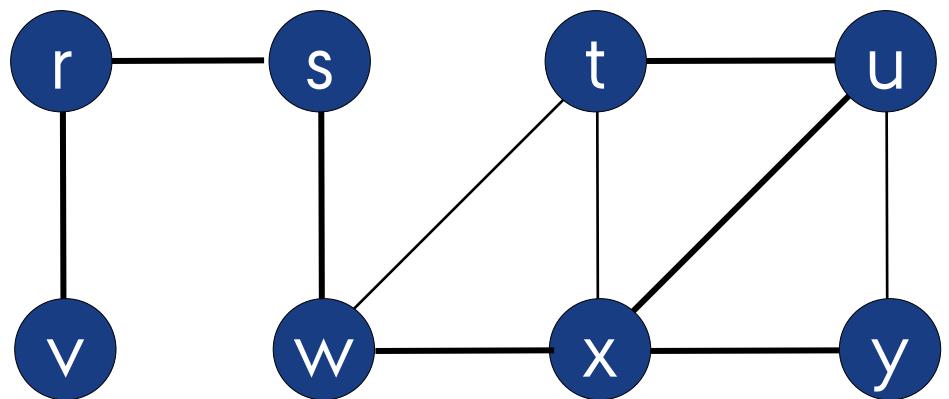


```

1  using namespace std;
2  #include <vector>
3  #include <queue>
4  #include <stack>
5  #include <iostream>
6  #include <climits>
7
8  #define INFINITY INT_MAX
9
10 class Graph {
11 private:
12     vector<vector<int>> adj;
13
14     void dfs(int v, vector<bool>& visited, vector<int>& path, stack<int>& sorting) {
15         visited[v] = true;
16         for (auto w : adj[v]) {
17             if (!visited[w]) {
18                 path[w] = v;
19                 dfs(w, visited, path, sorting);
20             }
21         }
22         sorting.push(v);
23     }
24
25 public:
26     Graph(int vertices = 0) : adj(vertices) { }
27
28     void addEdge(int u, int v);
29     void addDirectedEdge(int u, int v);
30     void dfs(int v, vector<int>& path);
31     void bfs(int s, vector<int>& path, vector<int>& dist);
32     void topsort(int v);
33     int components();
34     void print();
35 };
36
37 void Graph::dfs(int vertex, vector<int>& path) {
38     stack<int> sorting;
39     vector<bool> visited(adj.size());
40
41     for (int v = 0; v < adj.size(); ++v) {
42         visited[v] = false;
43     }
44     dfs(vertex, visited, path, sorting);
45 }
  
```

# DEPTH-FIRST SEARCH

DFS explores the **neighborhood** before finishing a vertex. Let us see the behavior starting from  $t$ .



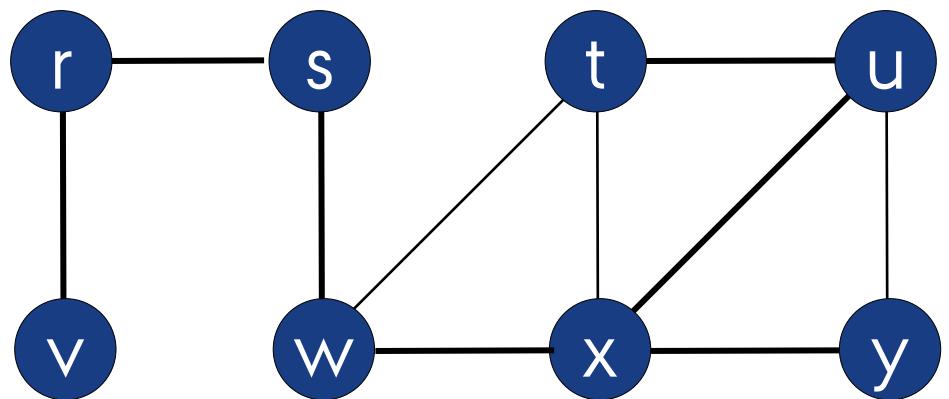
```

1  using namespace std;
2  #include <vector>
3  #include <queue>
4  #include <stack>
5  #include <iostream>
6  #include <climits>
7
8  #define INFINITY INT_MAX
9
10 class Graph {
11     private:
12         vector<vector<int>> adj;
13
14     void dfs(int v, vector<bool>& visited, vector<int>& path, stack<int>& sorting) {
15         visited[v] = true;
16         for (auto w : adj[v]) {
17             if (!visited[w]) {
18                 path[w] = v;
19                 dfs(w, visited, path, sorting);
20             }
21         }
22         sorting.push(v);
23     }
24
25     public:
26     Graph(int vertices = 0) : adj(vertices) { }
27
28     void addEdge(int u, int v);
29     void addDirectedEdge(int u, int v);
30     void dfs(int v, vector<int>& path);
31     void bfs(int s, vector<int>& path, vector<int>& dist);
32     void topsort(int v);
33     int components();
34     void print();
35 };
36
37 void Graph::dfs(int vertex, vector<int>& path) {
38     stack<int> sorting;
39     vector<bool> visited(adj.size());
40
41     for (int v = 0; v < adj.size(); ++v) {
42         visited[v] = false;
43     }
44     dfs(vertex, visited, path, sorting);
45 }

```

# DEPTH-FIRST SEARCH

DFS explores the **neighborhood** before finishing a vertex. Let us see the behavior starting from  $t$ .



```

1  using namespace std;
2  #include <vector>
3  #include <queue>
4  #include <stack>
5  #include <iostream>
6  #include <climits>
7
8  #define INFINITY INT_MAX
9
10 class Graph {
11     private:
12         vector<vector<int>> adj;
13
14     void dfs(int v, vector<bool>& visited, vector<int>& path, stack<int>& sorting) {
15         visited[v] = true;
16         for (auto w : adj[v]) {
17             if (!visited[w]) {
18                 path[w] = v;
19                 dfs(w, visited, path, sorting);
20             }
21         }
22         sorting.push(v);
23     }
24
25     public:
26     Graph(int vertices = 0) : adj(vertices) { }
27
28     void addEdge(int u, int v);
29     void addDirectedEdge(int u, int v);
30     void dfs(int v, vector<int>& path);
31     void bfs(int s, vector<int>& path, vector<int>& dist);
32     void topsort(int v);
33     int components();
34     void print();
35 };
15  void Graph::dfs(int vertex, vector<int>& path) {
16      stack<int> sorting;
17      vector<bool> visited(adj.size());
18
19      for (int v = 0; v < adj.size(); ++v) {
20          visited[v] = false;
21      }
22      dfs(vertex, visited, path, sorting);
23 }

```

# REFLECTION

---

1. In the dfs implementation what does the parameters path and sorting represent?
2. What is the time complexity of dfs?
3. The given dfs implementation is recursive but can be made iteratively as well using a stack. Suggest by drawing example or by pseudo-code how this can be done?

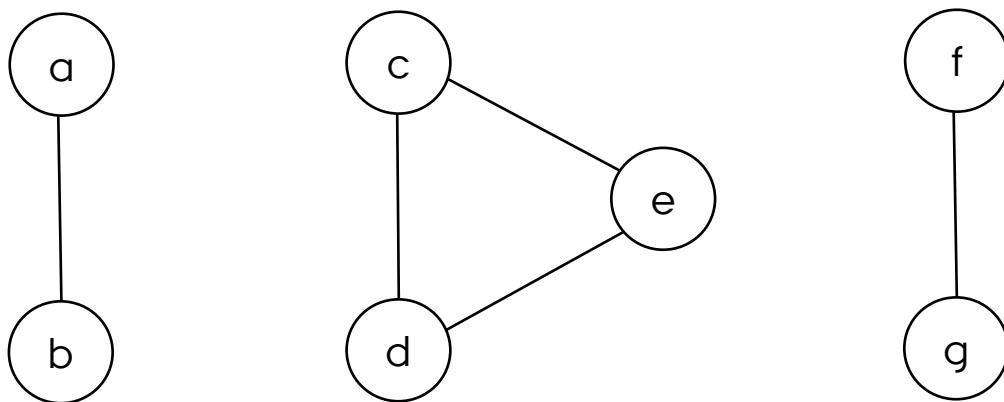


Photo by [Anthony Tran](#) on [Unsplash](#)

# APPLICATIONS OF DFS

There are two simple applications of the DFS algorithm.

The first is **counting the connected components** (number of DFS calls).

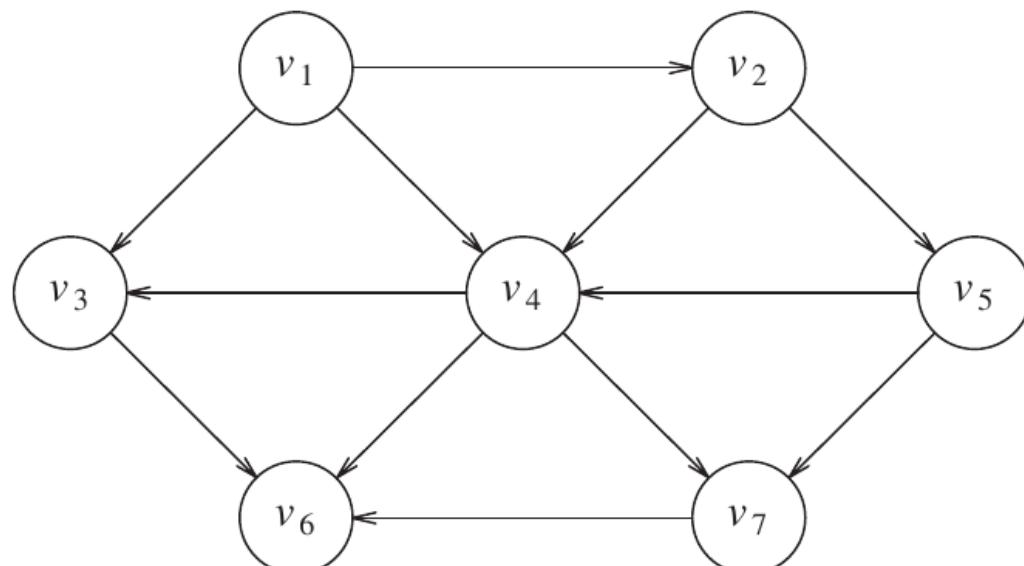


graph\_class.cpp

```
52 int Graph::components() {
53     int components = 0;
54     stack<int> sorting;
55     vector<int> path(adj.size());
56     vector<bool> visited(adj.size());
57     for (int v = 0; v < adj.size(); ++v) {
58         visited[v] = false;
59     }
60
61     for (int v = 0; v < adj.size(); ++v) {
62         if (!visited[v]) {
63             ++components;
64             dfs(v, visited, path, sorting);
65         }
66     }
67
68     return components;
69 }
70
71 void Graph::topsort(int v) {
72     stack<int> sorting;
73     vector<int> path(adj.size());
74     vector<bool> visited(adj.size());
75
76     for (int v = 0; v < adj.size(); ++v) {
77         visited[v] = false;
78     }
79     dfs(v, visited, path, sorting);
80     while (sorting.empty() == false) {
81         cout << sorting.top() << " ";
82         sorting.pop();
83     }
84 }
```

# APPLICATIONS OF DFS

There are two simple applications of the DFS algorithm. The second is **topological sorting** of a direct acyclic graph (DAG): if there is a path from  $v_i$  to  $v_j$ ,  $v_j$  appears *after*  $v_i$  in the ordering  $(v_1, v_2, v_5, v_4, v_7, v_3, v_6)$ .



graph\_class.cpp

```
52 int Graph::components() {
53     int components = 0;
54     stack<int> sorting;
55     vector<int> path(adj.size());
56     vector<bool> visited(adj.size());
57     for (int v = 0; v < adj.size(); ++v) {
58         visited[v] = false;
59     }
60
61     for (int v = 0; v < adj.size(); ++v) {
62         if (!visited[v]) {
63             ++components;
64             dfs(v, visited, path, sorting);
65         }
66     }
67
68     return components;
69 }
70
71 void Graph::topsort(int v) {
72     stack<int> sorting;
73     vector<int> path(adj.size());
74     vector<bool> visited(adj.size());
75
76     for (int v = 0; v < adj.size(); ++v) {
77         visited[v] = false;
78     }
79     dfs(v, visited, path, sorting);
80     while (sorting.empty() == false) {
81         cout << sorting.top() << " ";
82         sorting.pop();
83     }
84 }
```



# AGENDA

---

- ✓ Definitions and representations
- ✓ Depth-First Search (DFS) and applications
- Breadth-First Search (BFS) and applications
- Single-source Shortest-Path Algorithms

# BREADTH-FIRST SEARCH

---

Consists in an approach that explores the graph in **breadth**, one level at a time by **increasing distance** from the source.

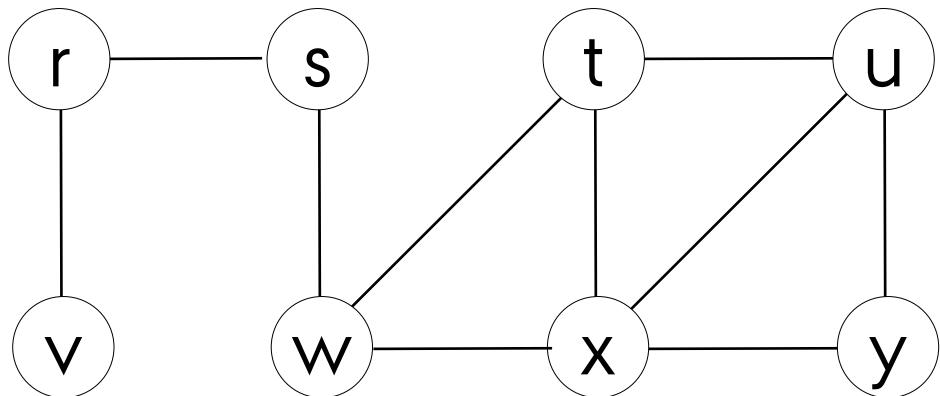
The algorithm also computes other **useful** information while it explores the graph:

- A breadth-first tree that stores the **predecessor** in the search
- The distance in **number of edges** from the **source to the vertex**

# BREADTH-FIRST SEARCH

BFS finishes a vertex before exploring the **neighborhood**. Let us see the behavior starting from  $t$ .

Simulate by hand – see paper



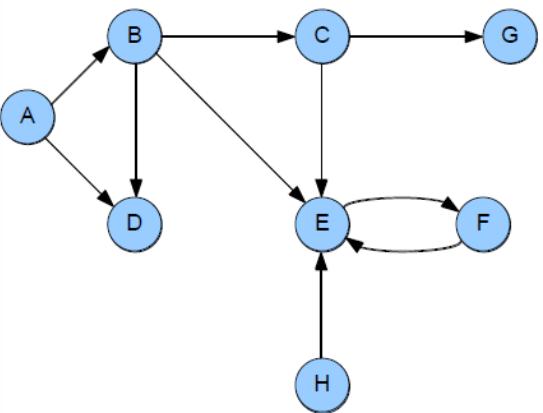
```
30 void Graph::bfs(int s, vector<int>& path, vector<int>& dist) {  
31     queue<int> q;  
32     vector<bool> visited(adj.size());  
33  
34     for (int v = 0; v < adj.size(); ++v) {  
35         dist[v] = INT_MAX;  
36         visited[v] = false;  
37     }  
38     dist[s] = 0;  
39     visited[s] = true;  
40     q.push(s);  
41  
42     while (!q.empty()) {  
43         s = q.front();  
44         q.pop();  
45         for (auto w : adj[s]) {  
46             if (!visited[w]) {  
47                 dist[w] = dist[s] + 1;  
48                 visited[w] = true;  
49                 path[w] = s;  
50                 q.push(w);  
51             }  
52         }  
53     }  
54 }
```

graph\_class.cpp

# REFLECTION

---

1. Perform BFS on the below graph (directed) starting in node A. For each step make note of the queue state.



2. What is the time complexity of bfs?
3. BFS finds the shortest path in an unweighted graph from the start node to all other nodes. In terms of number of edges. Argue for why this is the case?



# BREADTH-FIRST SEARCH

---

The **main** application of the BFS algorithm is to compute the **distance in number of edges** from the source vertex to all other **vertices**.

This intuition will help to build a **single-source shortest path** algorithm due to Dijkstra.

```
30 void Graph::bfs(int s, vector<int>& path, vector<int>& dist) {  
31     queue<int> q;  
32     vector<bool> visited(adj.size());  
33  
34     for (int v = 0; v < adj.size(); ++v) {  
35         dist[v] = INT_MAX;  
36         visited[v] = false;  
37     }  
38     dist[s] = 0;  
39     visited[s] = true;  
40     q.push(s);  
41  
42     while (!q.empty()) {  
43         s = q.front();  
44         q.pop();  
45         for (auto w : adj[s]) {  
46             if (!visited[w]) {  
47                 dist[w] = dist[s] + 1;  
48                 visited[w] = true;  
49                 path[w] = s;  
50                 q.push(w);  
51             }  
52         }  
53     }  
54 }
```

graph\_class.cpp

# AGENDA

---

- ✓ Definitions and representations
- ✓ Depth-First Search (DFS) and applications
- ✓ Breadth-First Search (BFS) and applications
- Single-source Shortest-Path Algorithms

# DIJKSTRA'S ALGORITHM

---

Let  $G$  a directed graph and suppose that every edge  $(u, v)$  is associated to a **weight** (cost, distance)  $W(u, v)$ .

## Single-destination shortest path problem:

Given two vertices  $s$  and  $t$  in  $(G, W)$ , find a path of minimum weight between  $s$  and  $t$ .

Apparently, it is not harder than the **single-source shortest path** problem to find the shortest path from  $s$  to **every other vertex**.

We will reuse a variant of the BFS algorithm to compute the shortest path (both length and path itself).

# DIJKSTRA'S ALGORITHM

---

The intuition behind the algorithm is the following:

1. We start with a **bad estimate** of the distance between the source and all other nodes ( $\infty$ )
2. We find the **unvisited** node that is closest to the visited nodes
3. We check if that node can **improve** the estimate (*relaxation*)



# DIJKSTRA'S ALG

---

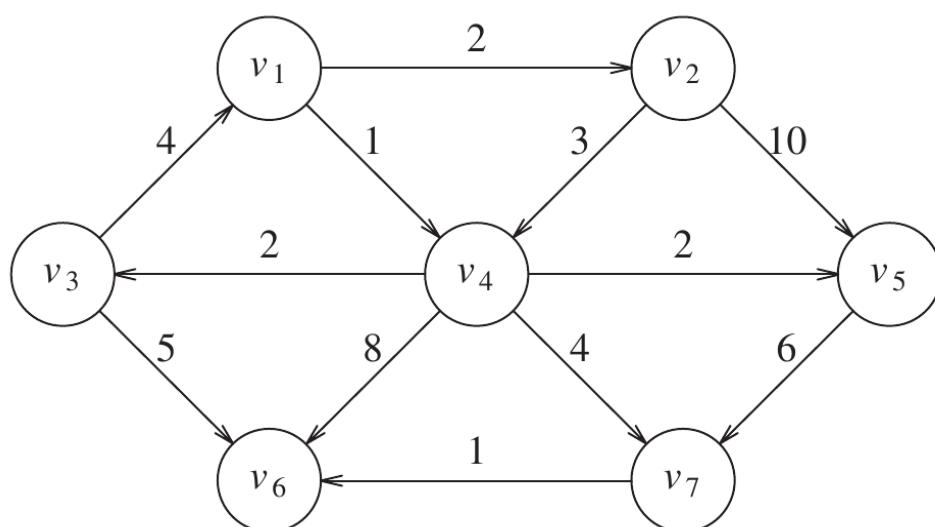
We first augment the graph implementation to store a **matrix of weights** between all vertices.

We also need a way to add a **weighted edge** with the given weight.

```
1 using namespace std;
2 #include <vector>
3 #include <queue>
4 #include <stack>
5 #include <iostream>
6 #include <climits>
7
8 #define INFINITY 1000000
9
10 class Graph {
11     private:
12         vector<vector<int>> adj;
13         vector<vector<int>> weight;
14
15     void dfs(int v, vector<bool>& visited, vector<int>& path, stack<int>& sorting) {
16         visited[v] = true;
17         for (auto w : adj[v]) {
18             if (!visited[w]) {
19                 path[w] = v;
20                 dfs(w, visited, path, sorting);
21             }
22         }
23         sorting.push(v);
24     }
25
26     public:
27     Graph(int vertices = 1) : adj(vertices), weight(vertices) {
28         for (int i = 0; i < vertices; i++) {
29             weight[i].resize(vertices, INFINITY);
30         }
31     }
32
33     void addEdge(int u, int v);
34     void addDirectedEdge(int u, int v);
35     void addWeightedEdge(int u, int v, int w);
36     void dfs(int v, vector<int>& path);
37     void bfs(int s, vector<int>& path, vector<int>& dist);
38     void topsort(int v);
39     int components();
40     void dijkstra(int s, vector<int>& path, vector<int>& dist);
41     void print();
42 };
```

# DIJKSTRA'S ALG

Let us execute the algorithm from  $v_1$ .



$v$	$known$	$d_v$	$p_v$
$v_1$	F	0	0
$v_2$	F	$\infty$	0
$v_3$	F	$\infty$	0
$v_4$	F	$\infty$	0
$v_5$	F	$\infty$	0
$v_6$	F	$\infty$	0
$v_7$	F	$\infty$	0

```
void Graph::dijkstra_pq(int s, vector < int >&path,
                        vector < int >&dist) {
    struct pair_comp {
        constexpr bool operator()( pair<int, int> const& a,
                                    pair<int, int> const& b) const noexcept {
            return a.first > b.first;
        }
    };
    priority_queue < pair<int, int>, v
                    vector<pair<int, int>>, pair_comp>q;
    vector < bool >visited(adj.size());

    for (int v = 0; v < adj.size(); ++v) {
        dist[v] = INFINITY;
        visited[v] = false;
        path[v] = -1;
    }
    dist[s] = 0;
    q.push(make_pair(dist[s], s));

    while (!q.empty()) {
        int u = q.top().second;
        q.pop();
        visited[u] = true;
        for (auto v:adj[u]) {
            if (!visited[v] && dist[u] != INT_MAX
                && dist[u] + weight[u][v] < dist[v]) {
                dist[v] = dist[u] + weight[u][v];
                path[v] = u;
                q.push(make_pair(dist[v], v));
            }
        }
    }
}
```

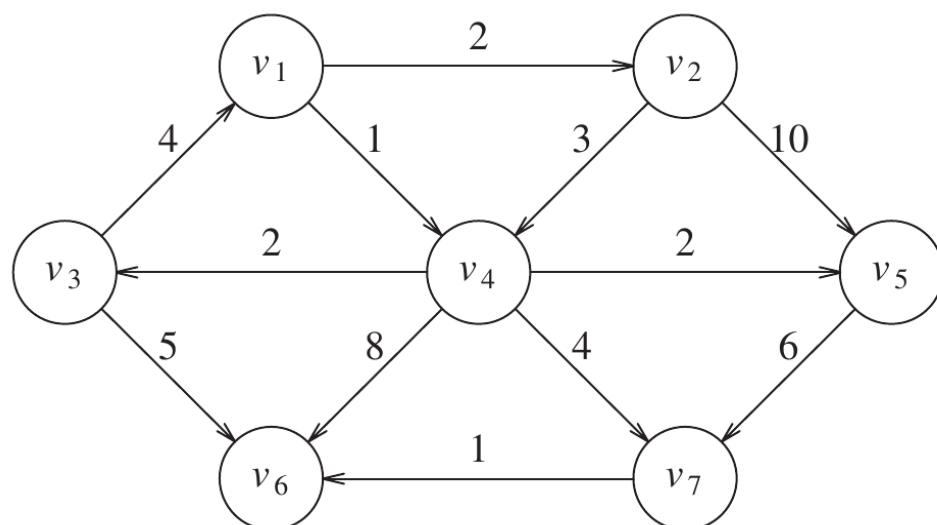
# DIJKSTRA'S ALG

Investigate V1's neighbours: V2, V4

Calculate the cost of progressing:  
V2 = 2 , V4 = 1

Then remove V1 and prioritize again:  
V4, V2

$v$	$known$	$d_v$	$p_v$
$v_1$	T	0	0
$v_2$	F	2	$v_1$
$v_3$	F	$\infty$	0
$v_4$	F	1	$v_1$
$v_5$	F	$\infty$	0
$v_6$	F	$\infty$	0
$v_7$	F	$\infty$	0



```
void Graph::dijkstra_pq(int s, vector < int >&path,
                        vector < int >&dist) {
    struct pair_comp {
        constexpr bool operator()( pair<int, int> const& a,
                                    pair<int, int> const& b) const noexcept {
            return a.first > b.first;
        }
    };
    priority_queue < pair<int, int>, v
                    vector<pair<int, int>>, pair_comp>q;
    vector < bool >visited(adj.size());
    for (int v = 0; v < adj.size(); ++v) {
        dist[v] = INFINITY;
        visited[v] = false;
        path[v] = -1;
    }
    dist[s] = 0;
    q.push(make_pair(dist[s], s));
    while (!q.empty()) {
        int u = q.top().second;
        q.pop();
        visited[u] = true;
        for (auto v:adj[u]) {
            if (!visited[v] && dist[u] != INT_MAX
                && dist[u] + weight[u][v] < dist[v]) {
                dist[v] = dist[u] + weight[u][v];
                path[v] = u;
                q.push(make_pair(dist[v], v));
            }
        }
    }
}
```

# DIJKSTRA'S ALG

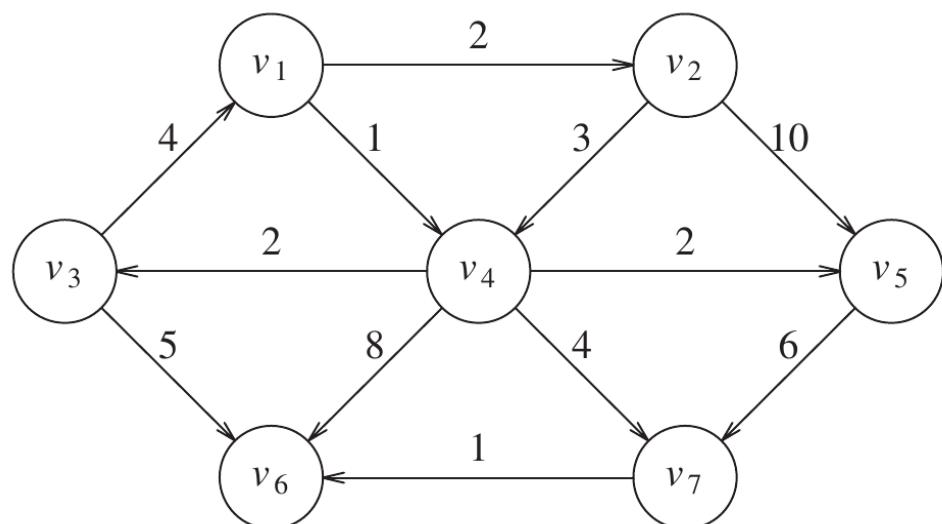
Investigate V4's neighbours: V3, V5, V7, V6

Calculate the cost of progressing:

$$V6: 1 + 8 = 9$$

Then remove V4 and prioritize again:  
V2, V3, V5, V7, V6

$v$	$known$	$d_v$	$p_v$
$v_1$	T	0	0
$v_2$	F	2	$v_1$
$v_3$	F	3	$v_4$
$v_4$	T	1	$v_1$
$v_5$	F	3	$v_4$
$v_6$	F	9	$v_4$
$v_7$	F	5	$v_4$



```
void Graph::dijkstra_pq(int s, vector < int >&path,
                        vector < int >&dist) {
    struct pair_comp {
        constexpr bool operator()( pair<int, int> const& a,
                                    pair<int, int> const& b) const noexcept {
            return a.first > b.first;
        }
    };
    priority_queue < pair<int, int>, v
                    vector<pair<int, int>>, pair_comp>q;
    vector < bool >visited(adj.size());

    for (int v = 0; v < adj.size(); ++v)
        dist[v] = INFINITY;
    visited[v] = false;
    path[v] = -1;
}
dist[s] = 0;
q.push(make_pair(dist[s], s));

while (!q.empty()) {
    int u = q.top().second;
    q.pop();
    visited[u] = true;
    for (auto v:adj[u]) {
        if (!visited[v] && dist[u] != INT_MAX
            && dist[u] + weight[u][v] < dist[v]) {
            dist[v] = dist[u] + weight[u][v];
            path[v] = u;
            q.push(make_pair(dist[v], v));
        }
    }
}
```

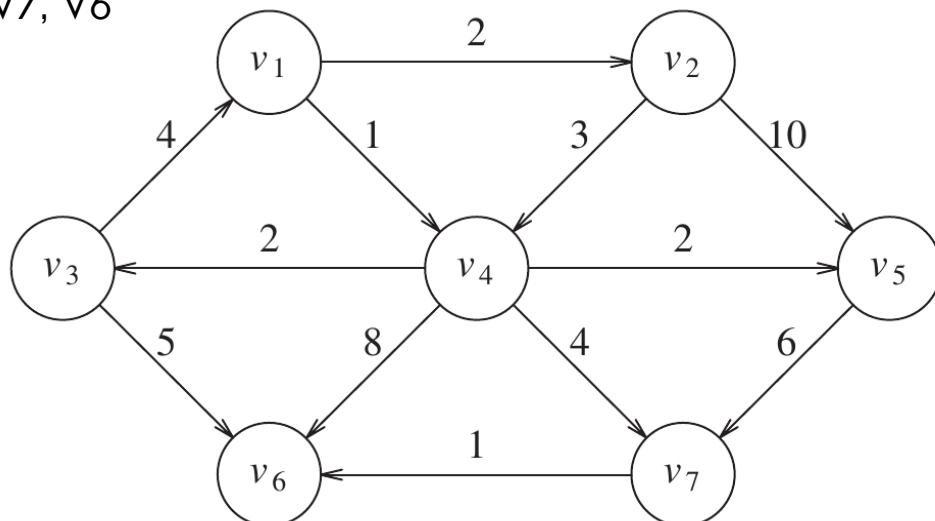
# DIJKSTRA'S ALG

Investigate V2's neighbours: V4, V5

Calculate the cost of progressing:  
V4 is already visited => must have lower  
cost than going through V2, so omit  
V5=12 => higher than already assigned  
value,  
so don't update

Then remove V2 and prioritize again:

V3, V5, V7, V6



v	known	$d_v$	$p_v$
v1	T	0	0
v2	T	2	v1
v3	F	3	v4
v4	T	1	v1
v5	F	3	v4
v6	F	9	v4
v7	F	5	v4

```
void Graph::dijkstra_pq(int s, vector < int >&path,
                        vector < int >&dist) {
    struct pair_comp {
        constexpr bool operator()( pair<int, int> const& a,
                                    pair<int, int> const& b) const noexcept {
            return a.first > b.first;
        }
    };
    priority_queue < pair<int, int>, v
                    vector<pair<int, int>>, pair_comp>q;
    vector < bool >visited(adj.size());
    for (int v = 0; v < adj.size(); ++v) {
        dist[v] = INFINITY;
        visited[v] = false;
        path[v] = -1;
    }
    dist[s] = 0;
    q.push(make_pair(dist[s], s));
    while (!q.empty()) {
        int u = q.top().second;
        q.pop();
        visited[u] = true;
        for (auto v:adj[u]) {
            if (!visited[v] && dist[u] != INT_MAX
                && dist[u] + weight[u][v] < dist[v]) {
                dist[v] = dist[u] + weight[u][v];
                path[v] = u;
                q.push(make_pair(dist[v], v));
            }
        }
    }
}
```

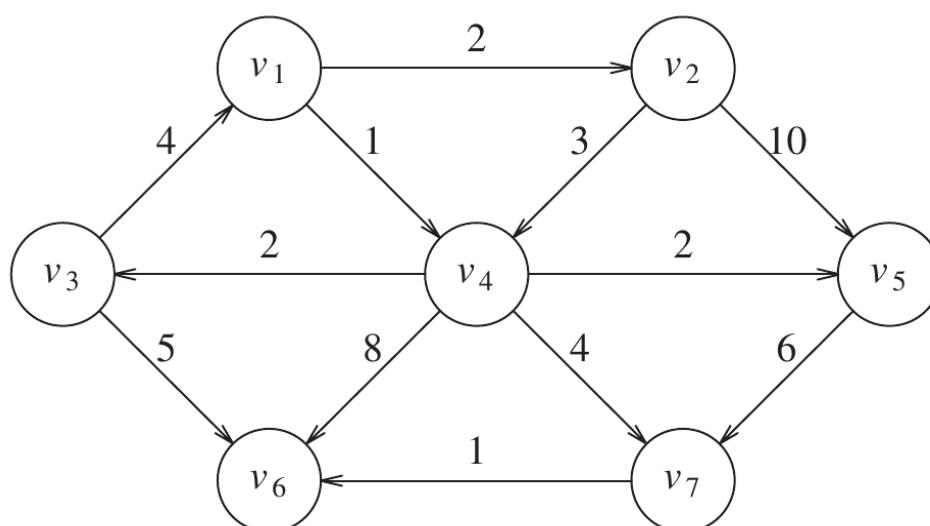
# DIJKSTRA'S ALG

Investigate V3's neighbours: V1, V6

Calculate the cost of progressing:  
V1 is already visited => omit  
 $V_6 = 3 + 5 = 8 \Rightarrow$  Update  $V_6 = 8$

Then remove V3 and prioritize again:  
V5, V7, V6

And likewise for V5..



$v$	known	$d_v$	$p_v$
$v_1$	T	0	0
$v_2$	T	2	$v_1$
$v_3$	T	3	$v_4$
$v_4$	T	1	$v_1$
$v_5$	T	3	$v_4$
$v_6$	F	8	$v_3$
$v_7$	F	5	$v_4$

```
void Graph::dijkstra_pq(int s, vector < int >&path,
                        vector < int >&dist) {
    struct pair_comp {
        constexpr bool operator()( pair<int, int> const& a,
                                    pair<int, int> const& b) const noexcept {
            return a.first > b.first;
        }
    };
    priority_queue < pair<int, int>, vector<pair<int, int>>, pair_comp>q;
    vector < bool >visited(adj.size());

    for (int v = 0; v < adj.size(); ++v)
        dist[v] = INFINITY;
    visited[v] = false;
    path[v] = -1;
}
dist[s] = 0;
q.push(make_pair(dist[s], s));

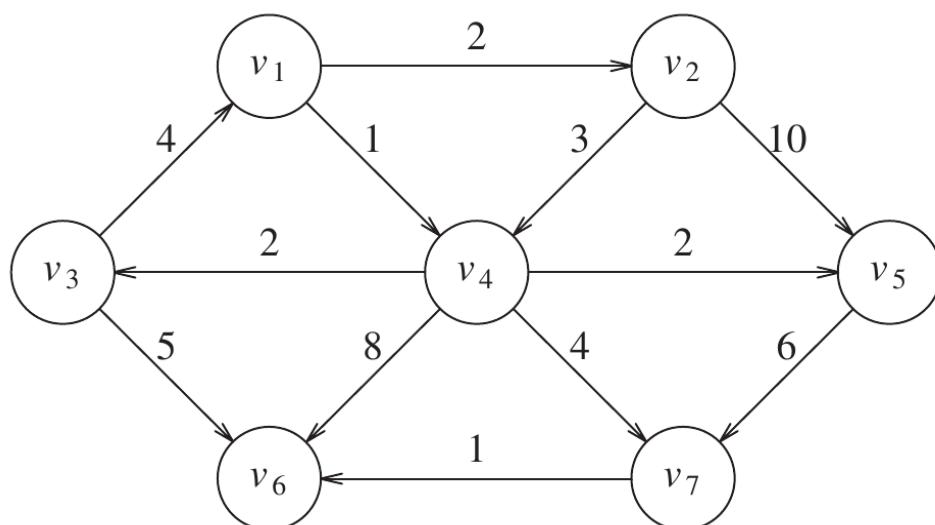
while (!q.empty()) {
    int u = q.top().second;
    q.pop();
    visited[u] = true;
    for (auto v:adj[u]) {
        if (!visited[v] && dist[u] != INT_MAX
            && dist[u] + weight[u][v] < dist[v]) {
            dist[v] = dist[u] + weight[u][v];
            path[v] = u;
            q.push(make_pair(dist[v], v));
        }
    }
}
```

# DIJKSTRA'S ALG

Investigate V7's neighbour: V6

Calculate the cost of progressing:  
 $V6=1+4+1=6 \Rightarrow$  Update  $V6=6$

Then remove V7 and prioritize again:  
V6



$v$	known	$d_v$	$p_v$
$v_1$	T	0	0
$v_2$	T	2	$v_1$
$v_3$	T	3	$v_4$
$v_4$	T	1	$v_1$
$v_5$	T	3	$v_4$
$v_6$	F	6	$v_7$
$v_7$	T	5	$v_4$

```
void Graph::dijkstra_pq(int s, vector < int >&path,
                        vector < int >&dist) {
    struct pair_comp {
        constexpr bool operator()( pair<int, int> const& a,
                                    pair<int, int> const& b) const noexcept {
            return a.first > b.first;
        }
    };
    priority_queue < pair<int, int>, v
                    vector<pair<int, int>>, pair_comp>q;
    vector < bool >visited(adj.size());

    for (int v = 0; v < adj.size(); ++v)
        dist[v] = INFINITY;
    visited[v] = false;
    path[v] = -1;
}
dist[s] = 0;
q.push(make_pair(dist[s], s));

while (!q.empty()) {
    int u = q.top().second;
    q.pop();
    visited[u] = true;
    for (auto v:adj[u]) {
        if (!visited[v] && dist[u] != INT_MAX
            && dist[u] + weight[u][v] < dist[v]) {
            dist[v] = dist[u] + weight[u][v];
            path[v] = u;
            q.push(make_pair(dist[v], v));
        }
    }
}
```

# DIJKSTRA'S ALG

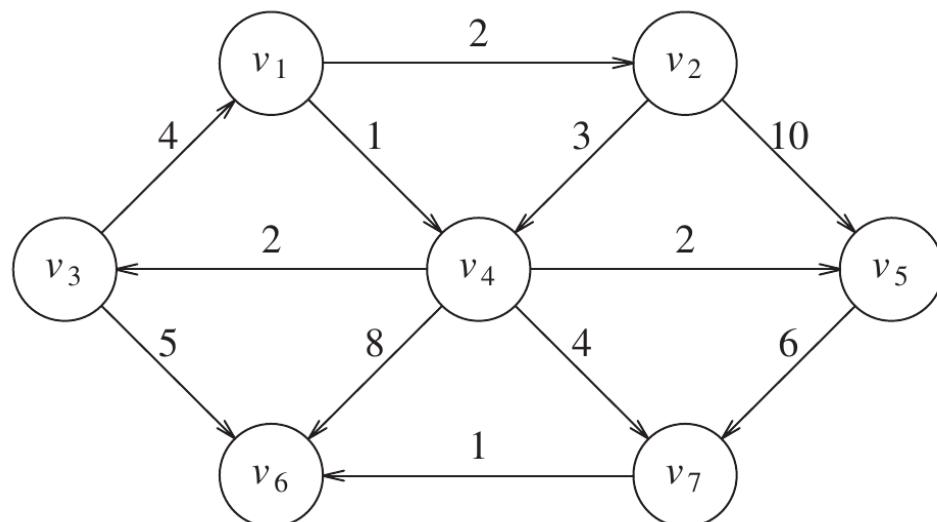
Until all nodes are visited.

Investigate V6's neighbour: <none>

Calculate the cost of progressing:  
<NA>

Then remove V6 and prioritize again:  
<Empty>

$v$	$known$	$d_v$	$p_v$
$v_1$	T	0	0
$v_2$	T	2	$v_1$
$v_3$	T	3	$v_4$
$v_4$	T	1	$v_1$
$v_5$	T	3	$v_4$
$v_6$	T	6	$v_7$
$v_7$	T	5	$v_4$



```
void Graph::dijkstra_pq(int s, vector < int >&path,
                        vector < int >&dist) {
    struct pair_comp {
        constexpr bool operator()( pair<int, int> const& a,
                                    pair<int, int> const& b) const noexcept {
            return a.first > b.first;
        }
    };
    priority_queue < pair<int, int>, v
                    vector<pair<int, int>>, pair_comp>q;
    vector < bool >visited(adj.size());

    for (int v = 0; v < adj.size(); ++v)
        dist[v] = INFINITY;
    visited[v] = false;
    path[v] = -1;
}
dist[s] = 0;
q.push(make_pair(dist[s], s));

while (!q.empty()) {
    int u = q.top().second;
    q.pop();
    visited[u] = true;
    for (auto v:adj[u]) {
        if (!visited[v] && dist[u] != INT_MAX
            && dist[u] + weight[u][v] < dist[v]) {
            dist[v] = dist[u] + weight[u][v];
            path[v] = u;
            q.push(make_pair(dist[v], v));
        }
    }
}
```

# DIJKSTRA'S ALG

---

The algorithm **extracts** the minimum  $/V/$  times and **decreases** distances at most  $/E/$  times.

```
void Graph::dijkstra_pq(int s, vector < int >&path,
                        vector < int >&dist) {
    struct pair_comp {
        constexpr bool operator()( pair<int, int> const& a,
                                   pair<int, int> const& b) const noexcept {
            return a.first > b.first;
        }
    };
    priority_queue < pair<int, int>, v
                    vector<pair<int, int>>, pair_comp>q;
    vector < bool >visited(adj.size());

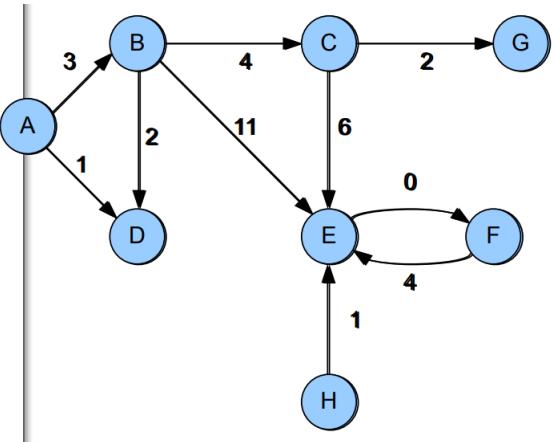
    for (int v = 0; v < adj.size(); ++v) {
        dist[v] = INFINITY;
        visited[v] = false;
        path[v] = -1;
    }
    dist[s] = 0;
    q.push(make_pair(dist[s], s));

    while (!q.empty()) {
        int u = q.top().second;
        q.pop();
        visited[u] = true;
        for (auto v:adj[u]) {
            if (!visited[v] && dist[u] != INT_MAX
                && dist[u] + weight[u][v] < dist[v]) {
                dist[v] = dist[u] + weight[u][v];
                path[v] = u;
                q.push(make_pair(dist[v], v));
            }
        }
    }
}
```

# REFLECTION

---

1. Perform Dijkstra's algorithm on the below graph starting in node A.



2. Can we use Dijkstra's algorithm if the input graph has negative weight edges? Argue for your answer e.g. providing an example graph.
3. If all edges in a graph has weight 1, how does Dijkstra's algorithm then compare to another graph algorithm you have seen?



# PATHFINDING WITH A\*

# AGENDA

---

- Recap BFS + Dijkstra's algorithm
- A\* Search Algorithm

# BREADTH-FIRST SEARCH (BFS)

---

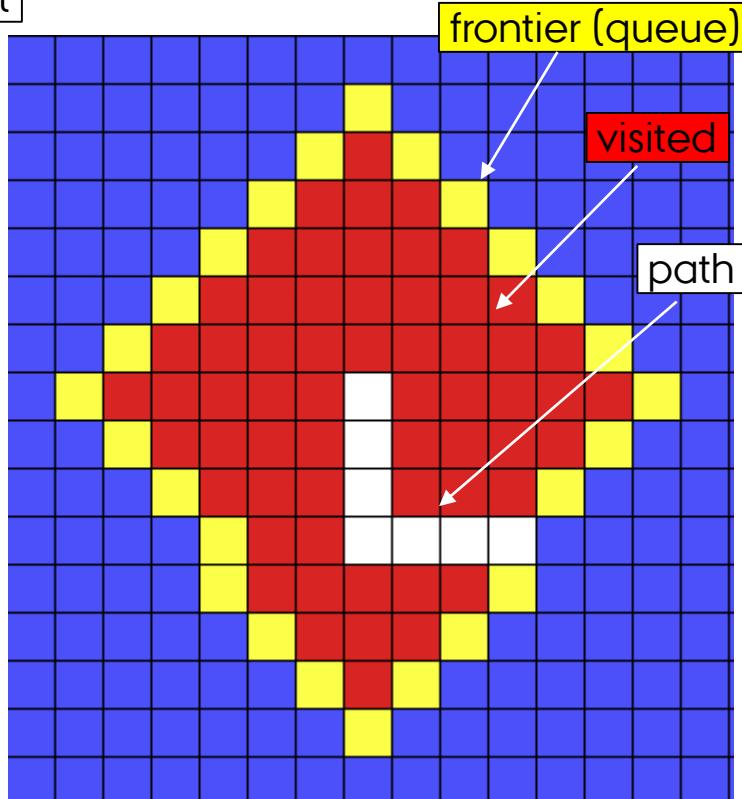
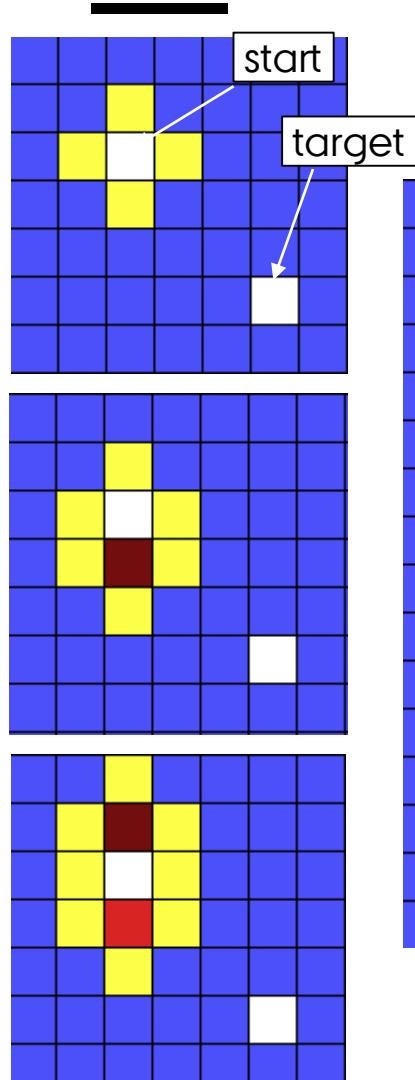


**BFS** explores in all directions regardless of cost or proximity to the target.

- BFS finds the shortest path from a start node to all other nodes in unweighted graph.
- BFS does not consider edge costs – it only considers the number of edges in the path
- BFS search complexity:  $O(|V| + |E|)$

```
30 void Graph::bfs(int s, vector<int>& path, vector<int>& dist) {  
31     queue<int> q;  
32     vector<bool> visited(adj.size());  
33  
34     for (int v = 0; v < adj.size(); ++v) {  
35         dist[v] = INT_MAX;  
36         visited[v] = false;  
37     }  
38     dist[s] = 0;  
39     visited[s] = true;  
40     q.push(s);  
41  
42     while (!q.empty()) {  
43         s = q.front();  
44         q.pop();  
45         for (auto w : adj[s]) {  
46             if (!visited[w]) {  
47                 dist[w] = dist[s] + 1;  
48                 visited[w] = true;  
49                 path[w] = s;  
50                 q.push(w);  
51             }  
52         }  
53     }  
54 }
```

# BREADTH-FIRST SEARCH (BFS)



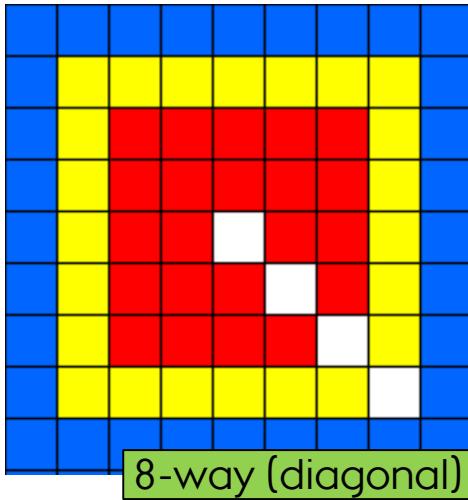
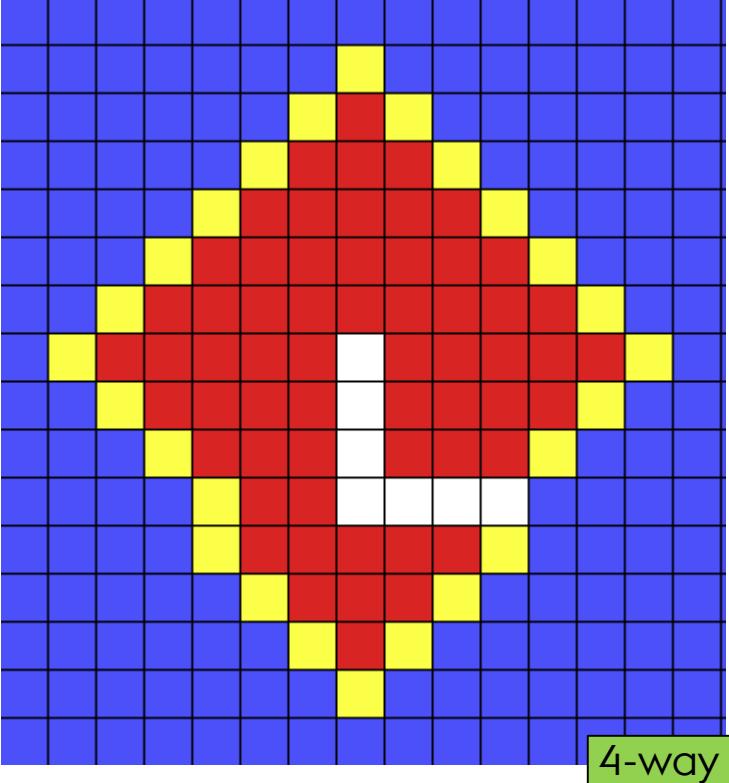
```
BFS(Graph g, Node s, Node t)
{
    // Initializing
    frontier = new Queue();
    s.prev = None;
    frontier.push(s);

    // Explore nodes
    while(!frontier.isEmpty())
    {
        current = frontier.get();
        if(current == t) return; // early exit

        foreach(Node n in current.neighbors)
        {
            if(n.prev == null)
            {
                n.prev = current;
                frontier.push(n);
            }
        }
    }
}
```

pseudo code

# BREADTH-FIRST SEARCH (BFS)



```
BFS(Graph g, Node s, Node t)
{
    // Initializing
    frontier = new Queue();
    s.prev = None;
    frontier.push(s);

    // Explore nodes
    while(!frontier.isEmpty())
    {
        current = frontier.get();
        if(current == t) return; // early exit

        foreach(Node n in current.neighbors)
        {
            if(n.prev == null)
            {
                n.prev = current;
                frontier.push(n);
            }
        }
    }
}
```

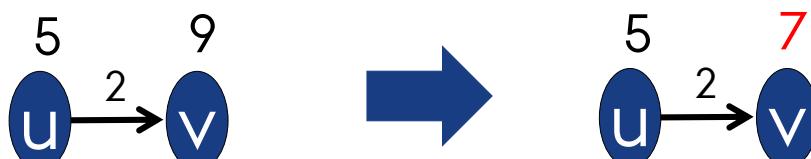
pseudo code

# DIJKSTRA'S ALGORITHM



Dijkstra's selects the next node to explore based on **lowest cost from source**. Finds the shortest path in a weighted graph.

- Shortest path = path for which the sum of edge weights is minimal.
- Same algorithm structure as BFS.
- Search complexity:  $O(|E| * \log(|V|))$
- **Relax** if cheaper path:



```
FindPathDijkstra(Graph g, Node s, Node t)
{
    // Initializing
    frontier = new PriorityQueue();
    s.prev = None;
    s.cost = 0
    frontier.push(s, s.cost);

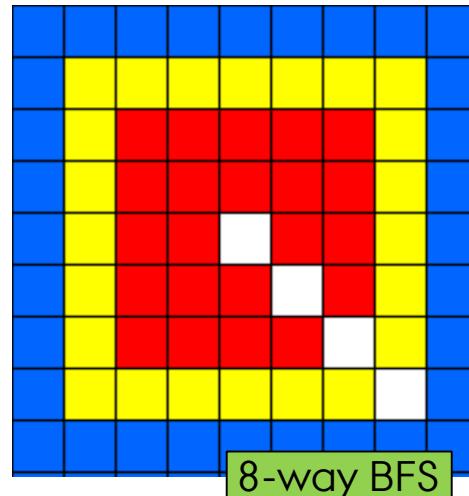
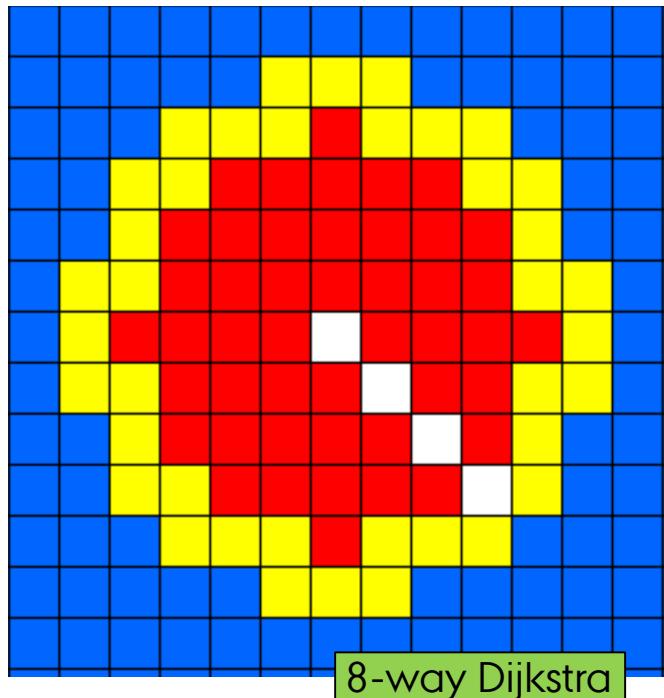
    // Explore nodes
    while(!frontier.isEmpty())
    {
        current = frontier.get();
        if(current == t) return; // early exit

        foreach(Node n in current.neighbors)
        {
            if(current.cost + edge(current, n).weight < n.cost)
            {
                n.cost = current.cost + edge(current, n).weight;
                n.prev = current;
                frontier.push(n, n.cost);
            }
        }
    }
}
```



# DIJKSTRA'S ALGORITHM

- In 8-way grid Dijkstra will form circle whereas BFS will form square. Diagonal higher cost.
- In 4-way grid Dijkstra will be the same as BFS.



```
FindPathDijkstra(Graph g, Node s, Node t)
{
    // Initializing
    frontier = new PriorityQueue();
    s.prev = None;
    s.cost = 0
    frontier.push(s, s.cost);

    // Explore nodes
    while(!frontier.isEmpty())
    {
        current = frontier.get();
        if(current == t) return; // early exit

        foreach(Node n in current.neighbors)
        {
            if(current.cost + edge(current, n).weight < n.cost)
            {
                n.cost = current.cost + edge(current, n).weight;
                n.prev = current;
                frontier.push(n, n.cost);
            }
        }
    }
}
```

# A\* ALGORITHM

---



**A\*** selects next node to explore based on **lowest sum of cost from source and cost to target**

- A\* prioritizes paths that seem to lead closer to the goal (“targeting algorithm”).
- The A\* algorithm (Hart, Nilsson & Raphael 1968) uses a **heuristic** to guide the search, i.e. select the next node to investigate based on a best guess of the actual distance to the target.
- Heuristic is an **estimated cost of optimal path from start node to target node**. It is a best guess and may be wrong/imprecise as it **cannot see graph (e.g. “walls”)**.  
Example heuristics: physical distance (Euclidean) or distance in coordinate system.
- Search complexity depends on heuristic.

# A\* ALGORITHM

---



**A\*** selects next node to explore based on **lowest sum of cost from source and cost to target**

- A\* selects the node from the frontier which minimizes the function:

$$f(n) = g(n) + h(n)$$

- n is the current node
  - g(n) is the cost from the source (s) to n
  - h(n) is the heuristic that estimates the cost of the optimal path from n to the target (t).
- 
- Notice: if  $h(n) = 0$  then A\* reduces to Dijkstra's algorithm.

# A\* ALGORITHM

```
FindPathDijkstra(Graph g, Node s, Node t)
{
    // Initializing
    frontier = new PriorityQueue();
    s.prev = None;
    s.cost = 0
    frontier.push(s, 0);

    // Explore nodes
    while(!frontier.isEmpty())
    {
        current = frontier.get();
        if(current == t) return; // early exit

        foreach(Node n in current.neighbors)
        {
            if(current.cost + edge(current, n).weight < n.cost)
            {
                n.cost = current.cost + edge(current, n).weight;
                priority = n.cost;
                n.prev = current;
                frontier.push(n, priority);
            }
        }
    }
}
```

```
FindPathAStar(Graph g, Node s, Node t)
{
    // Initializing
    frontier = new PriorityQueue();
    s.prev = None;
    s.cost = 0
    frontier.push(s, 0);

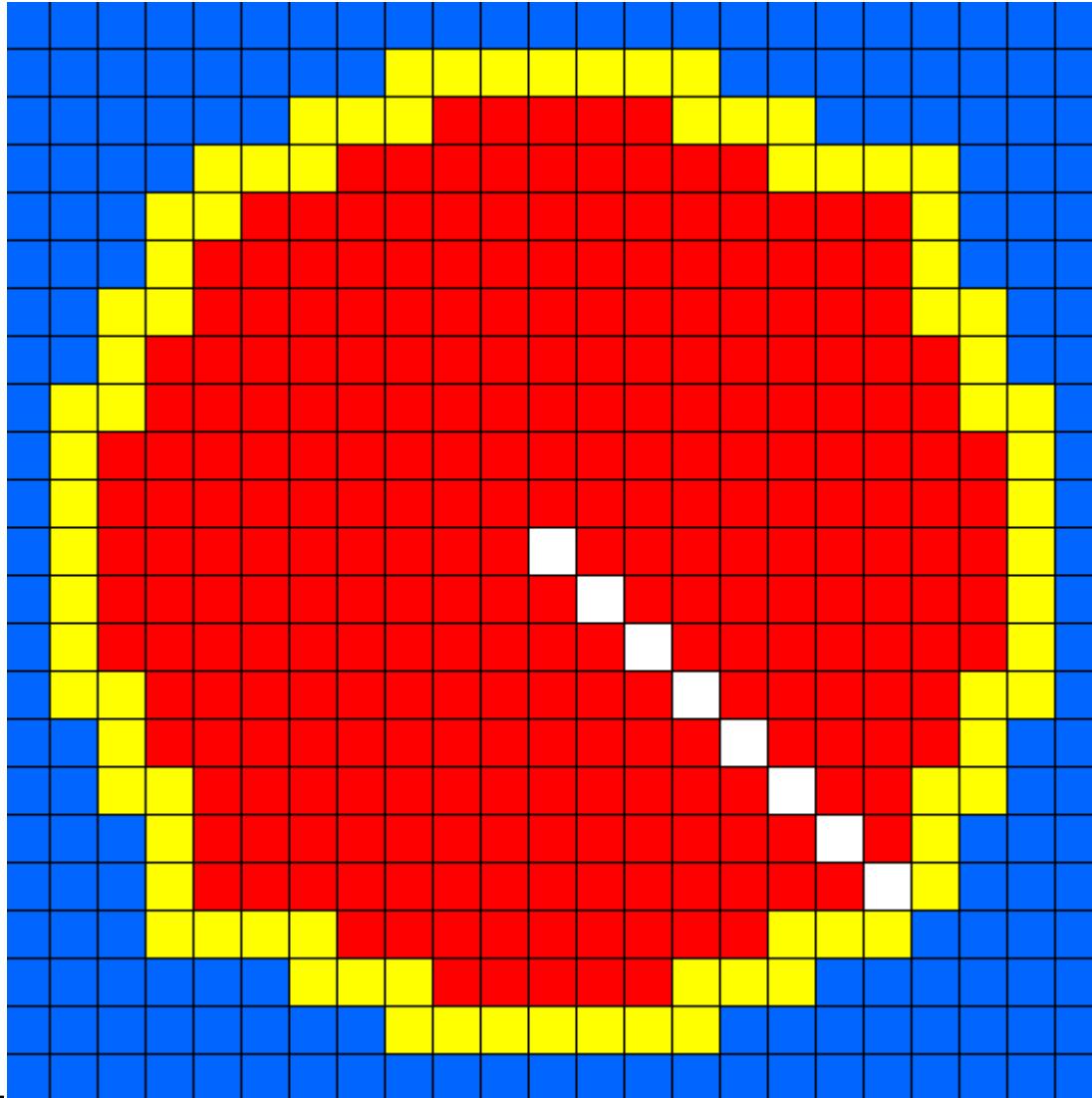
    // Explore nodes
    while(!frontier.isEmpty())
    {
        current = frontier.get();
        if(current == t) return; // early exit

        foreach(Node n in current.neighbors)
        {
            if(current.cost + edge(current, n).weight < n.cost)
            {
                n.cost = current.cost + edge(current, n).weight;
                priority = n.cost + h(n,t);
                n.prev = current;
                frontier.push(n, priority);
            }
        }
    }
}
```

# DIJKSTRA'S VS A\*

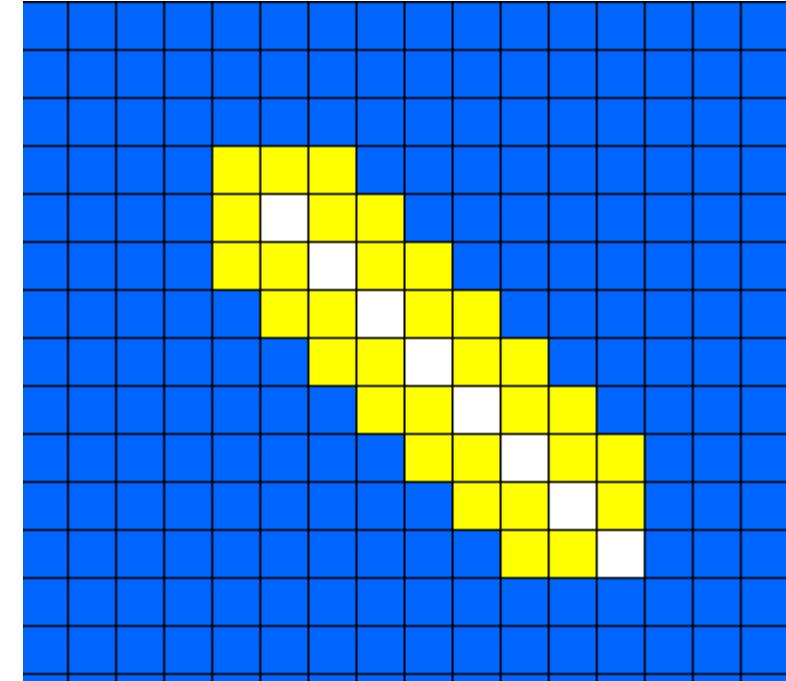


UNIVERSITY  
DEPARTMENT OF ENGINEERING



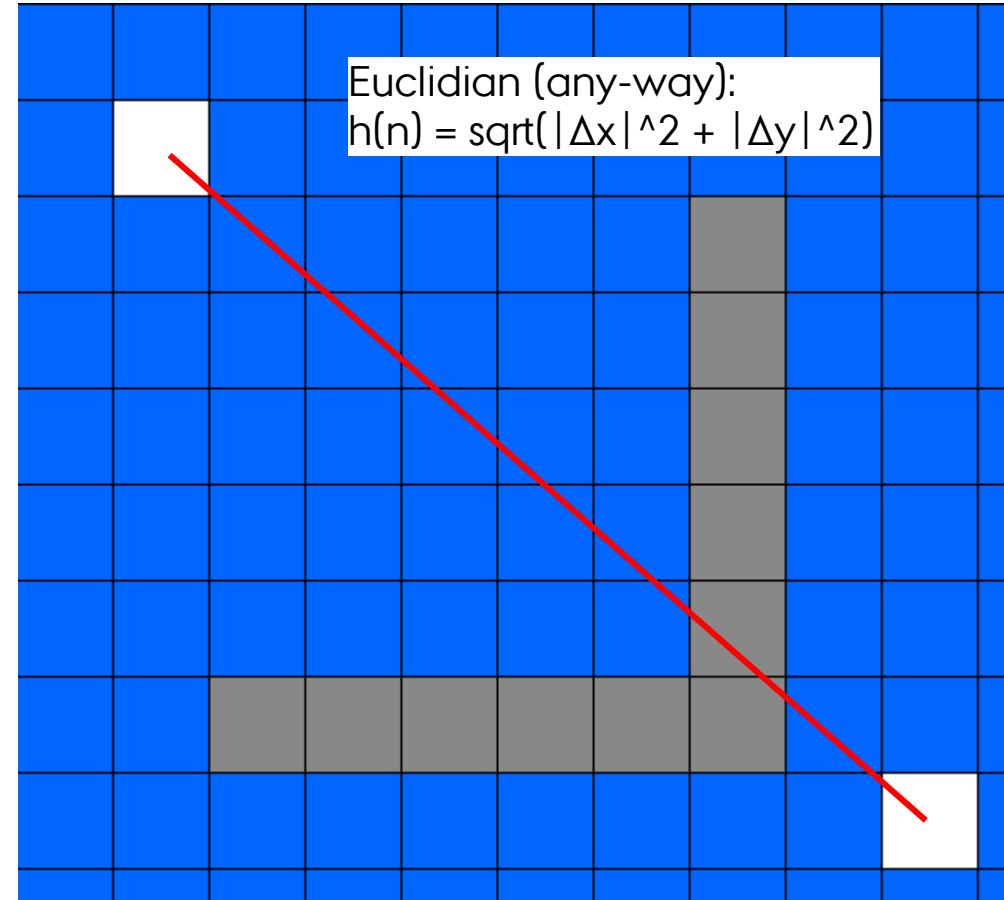
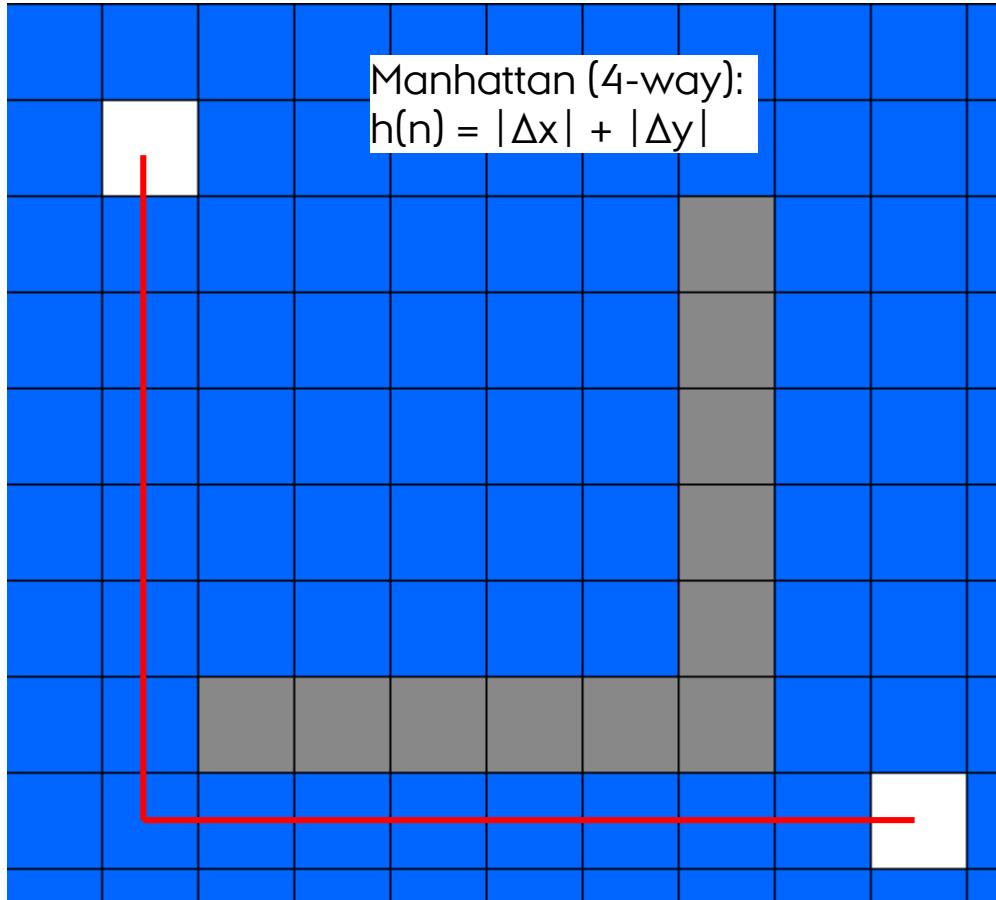
OCTOBER 2022

ALGORITHMS & DATA STRUCTURES

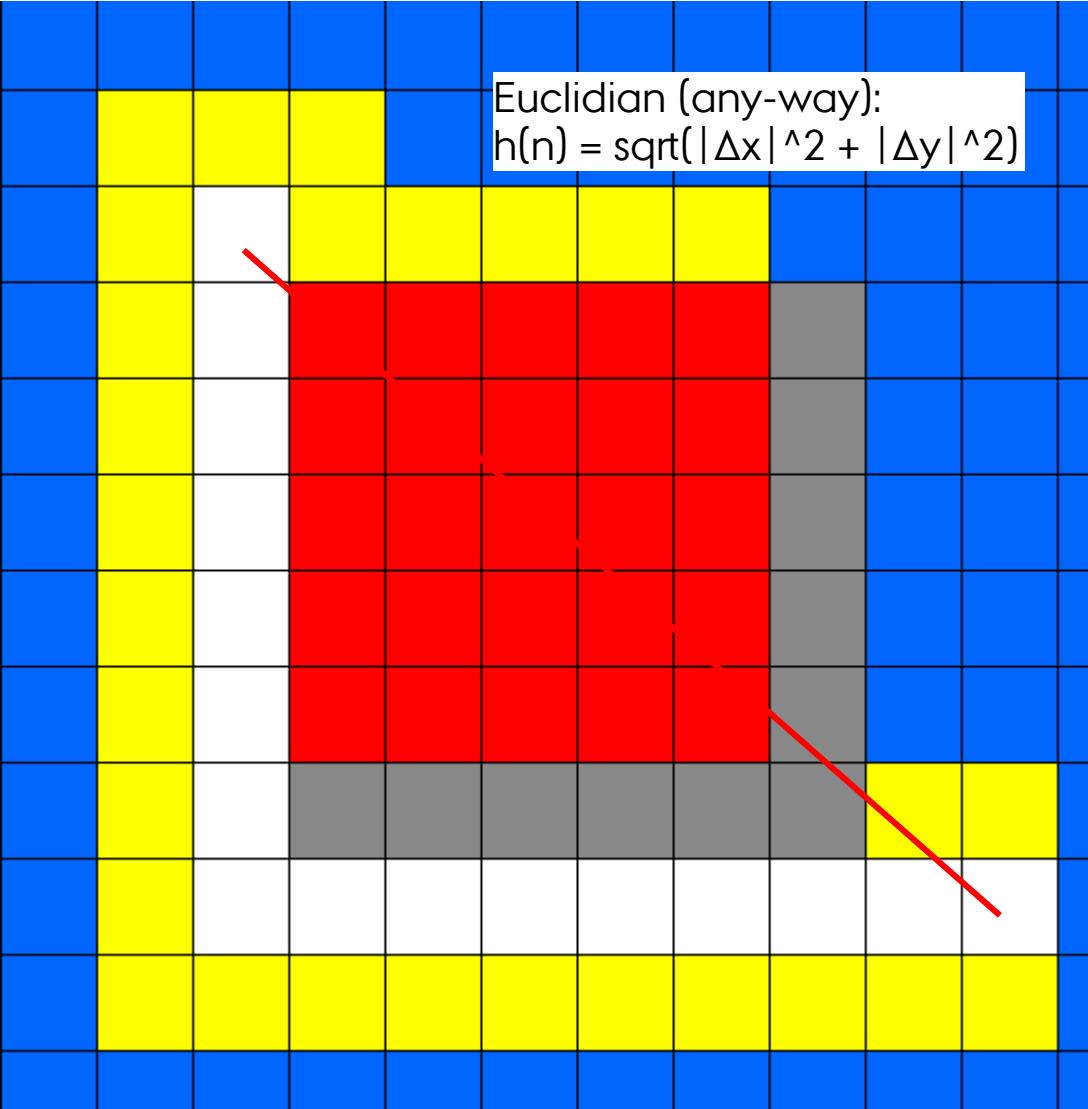
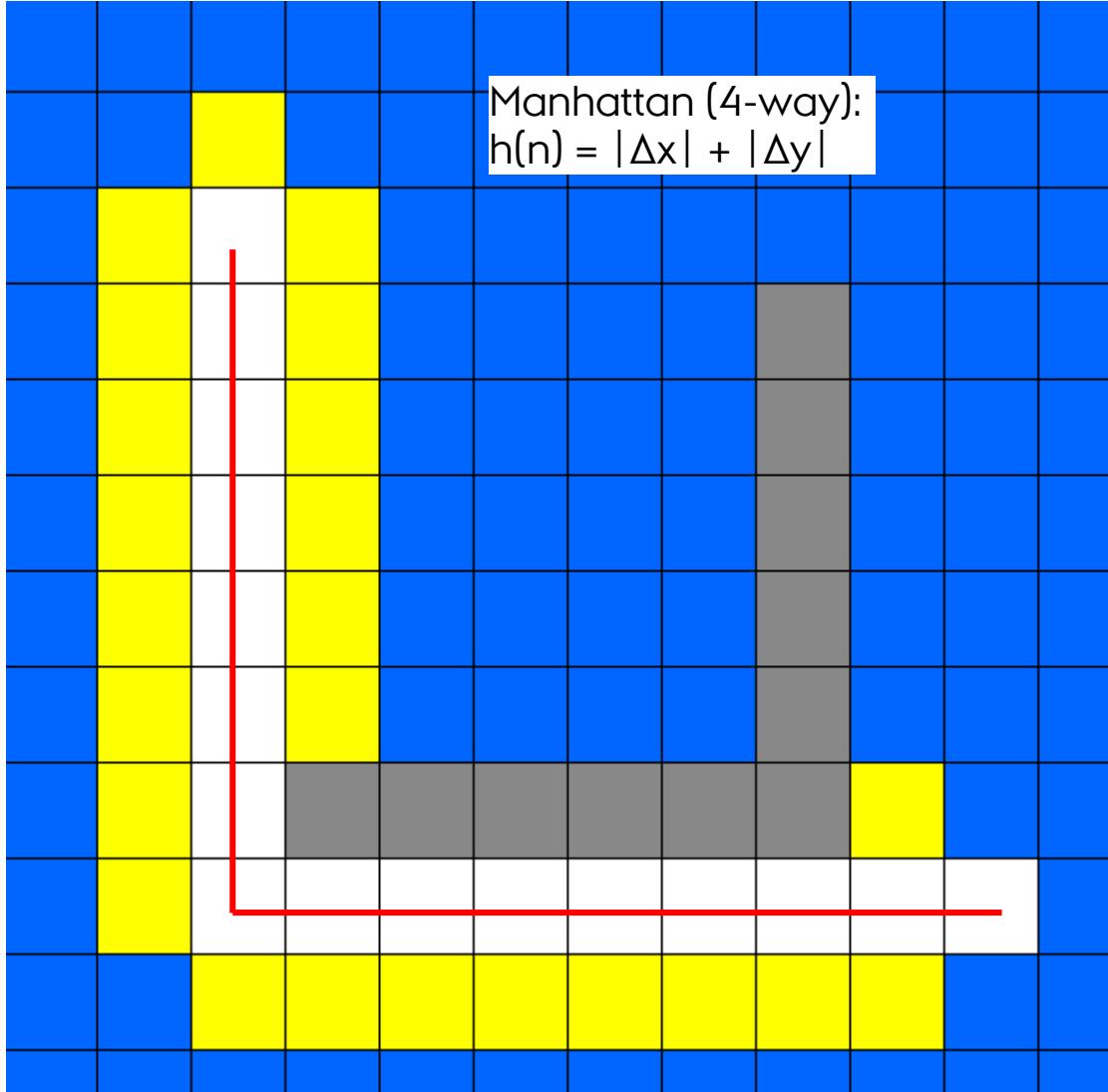


# HEURISTIC FUNCTION

---



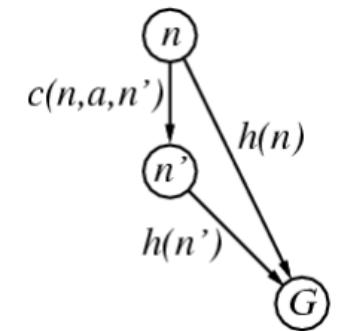
# HEURISTIC FUNCTION



# HEURISTIC FUNCTION

---

- If  $h(n)$  is always lower than or equal to the actual optimal cost of moving from  $n$  to  $T$ , then  $A^*$  is guaranteed to find a shortest path ( $h$  is **admissible**).
- May additionally require  $h(n)$  is **consistent** for  $A^*$  to be optimal. Depends on whether the algorithm allows re-queuing or uses visited set. A heuristic is **consistent** if for every node  $n$ , every successor  $n'$  of  $n$  generated by any action  $a$ ,  $h(n) \leq c(n,a,n') + h(n')$ . If  $h$  is consistent, we have that  $f(n)$  is **non-decreasing** along any path.
- The heuristics Manhattan and Euclidian are consistent and admissible.
- If  $h(n)$  is always equal to the actual cost from  $n$  to  $T$ , then  $A^*$  will only follow the best path and never expand anything else (unlikely case).
- If  $h(n)$  can be higher than the actual cost from  $n$  to  $T$ , then  $A^*$  may not find the shortest path (but may run even faster).

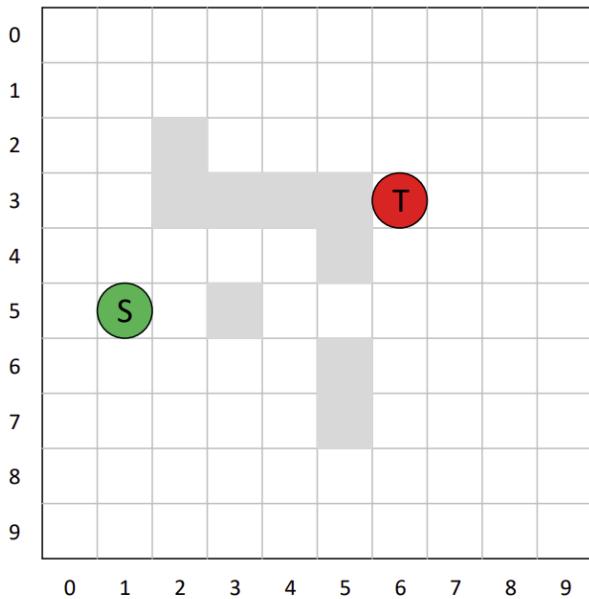


# REFLECTION

---

1. Assume that  $h(n) = |\Delta x| + |\Delta y|$  (Manhattan heuristic, 4-way grid). Use pen and paper to execute A\* search on the following graph.

Insert neighbors in the frontier in clockwise order and make note of  $g(n)$ ,  $h(n)$ , and  $f(n)$  for each step (table or inline)



# REFLECTION

---

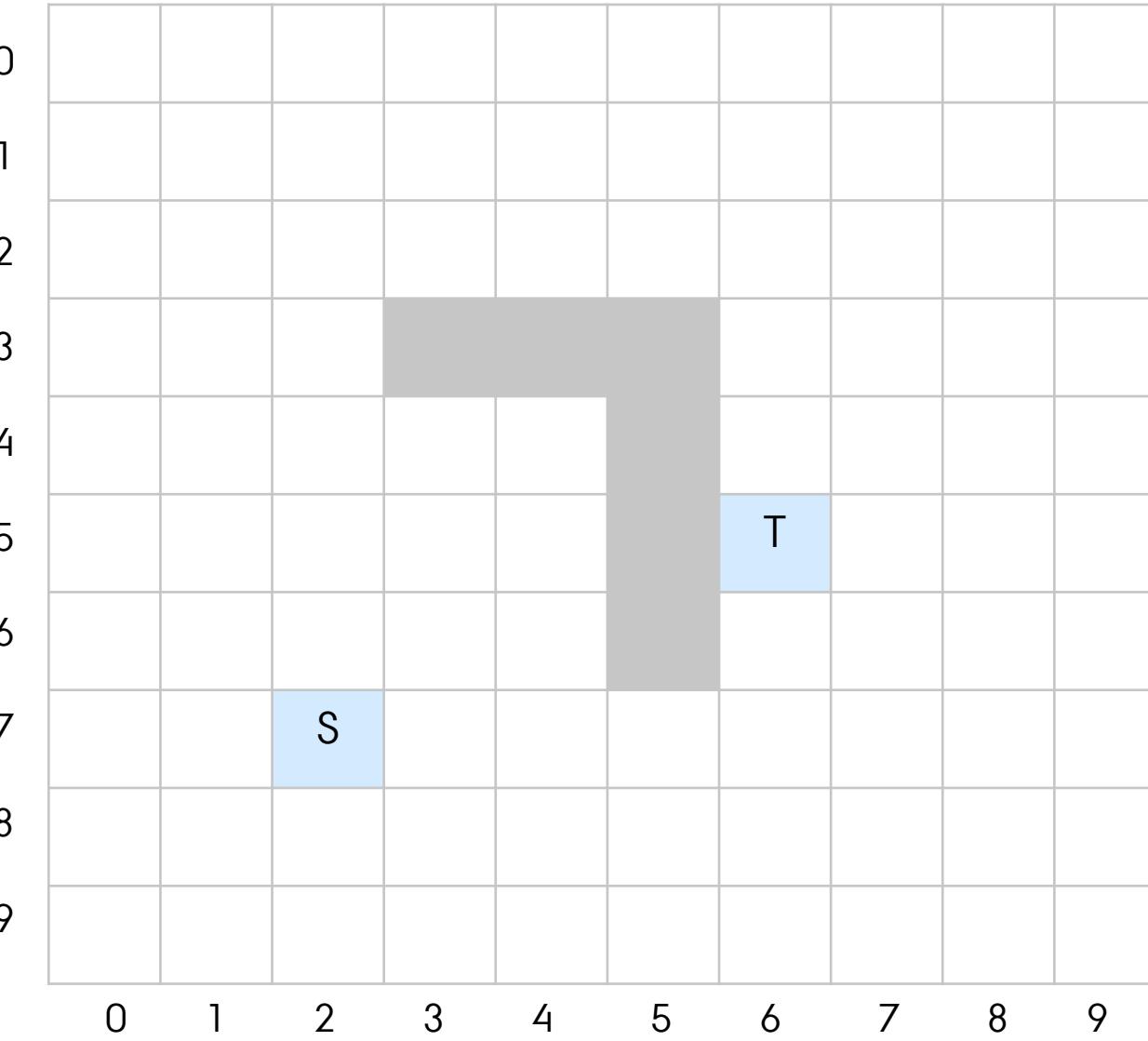
2. Give an example graph showing, that if  $h(n)$  overestimates the actual cost from  $n$  to target, then A\* may find a path that is **not** the shortest one.



Photo by [Anthony Tran](#) on [Unsplash](#)

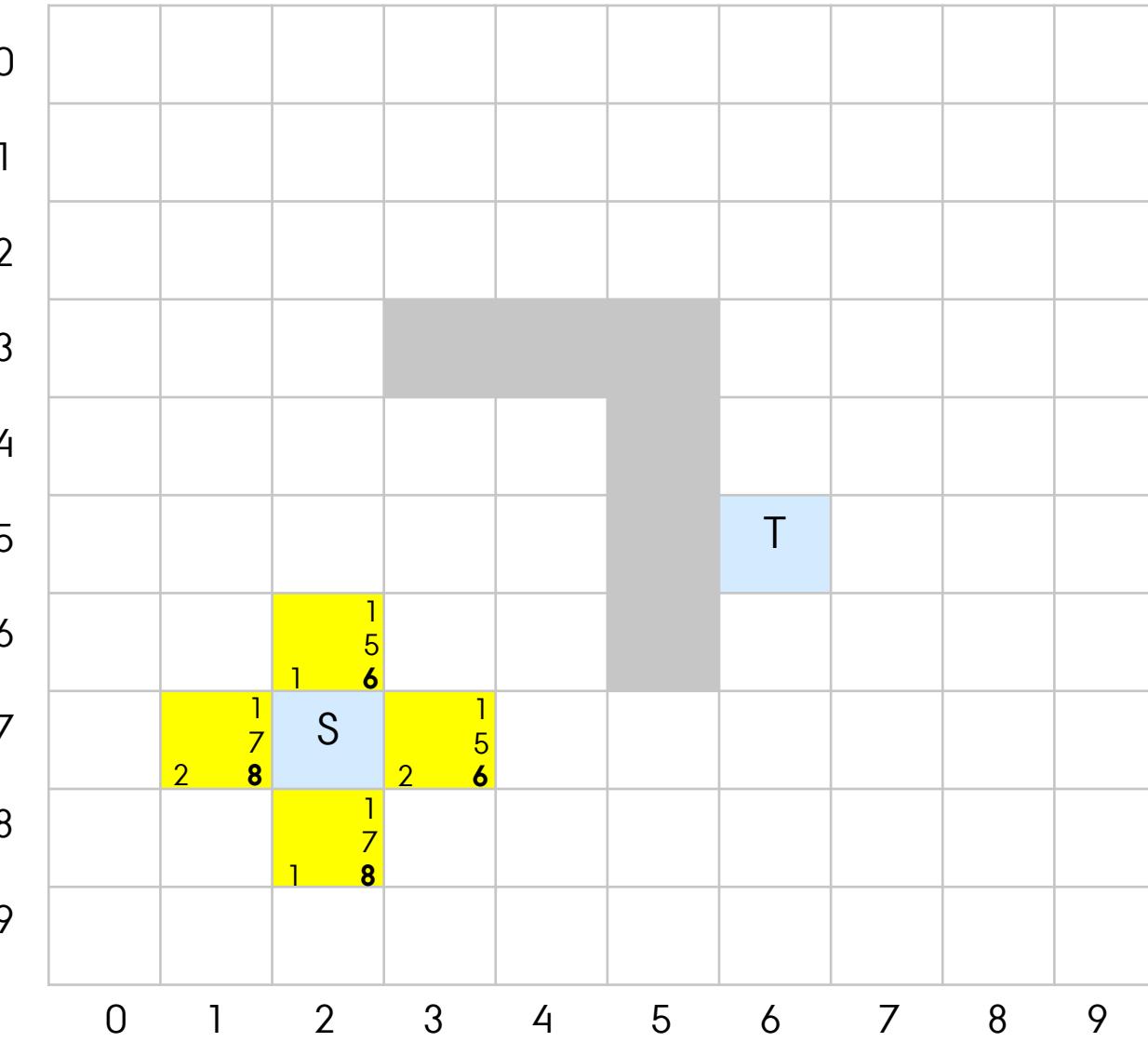
# EXAMPLE

---



# EXAMPLE

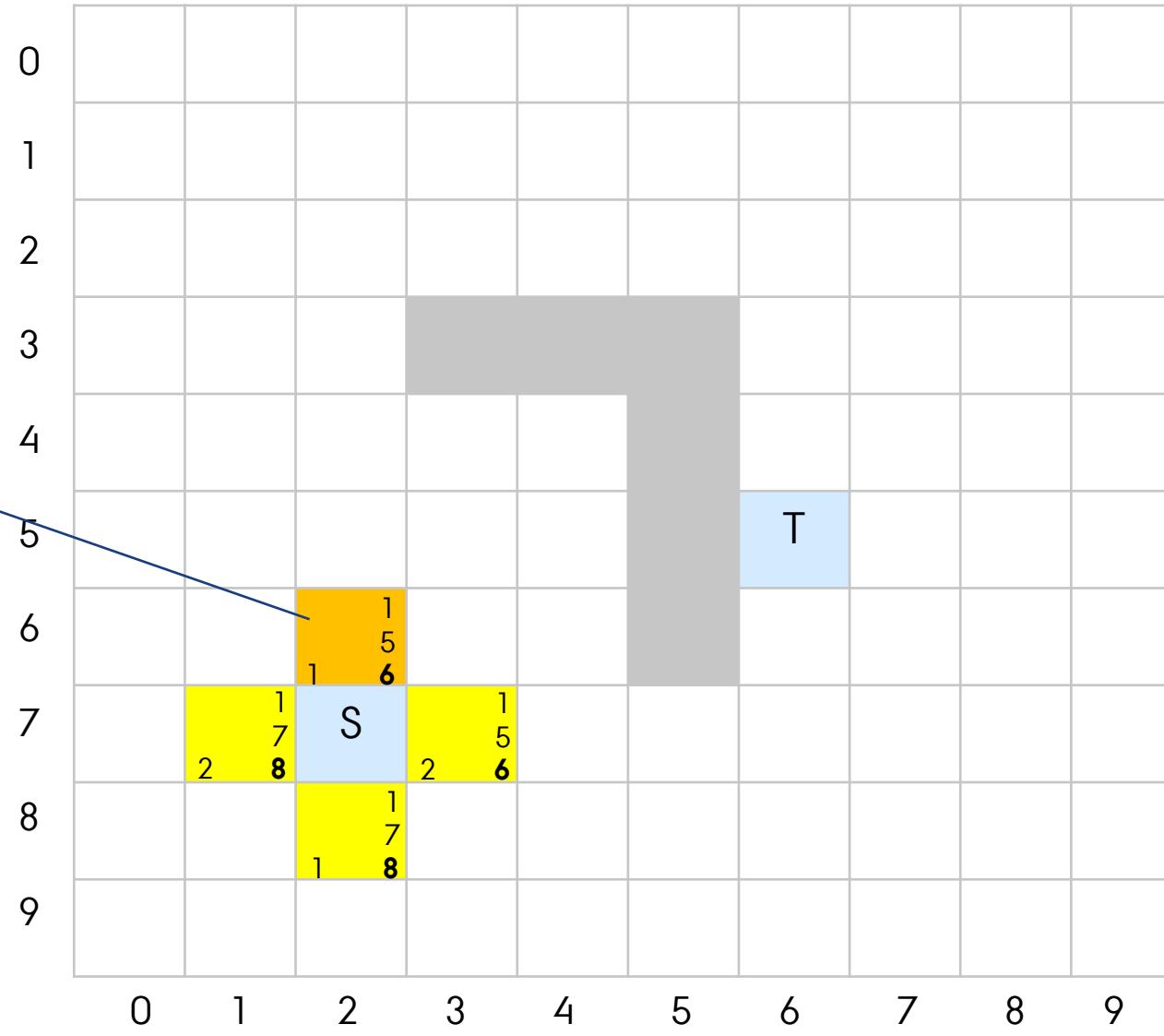
---



# EXAMPLE

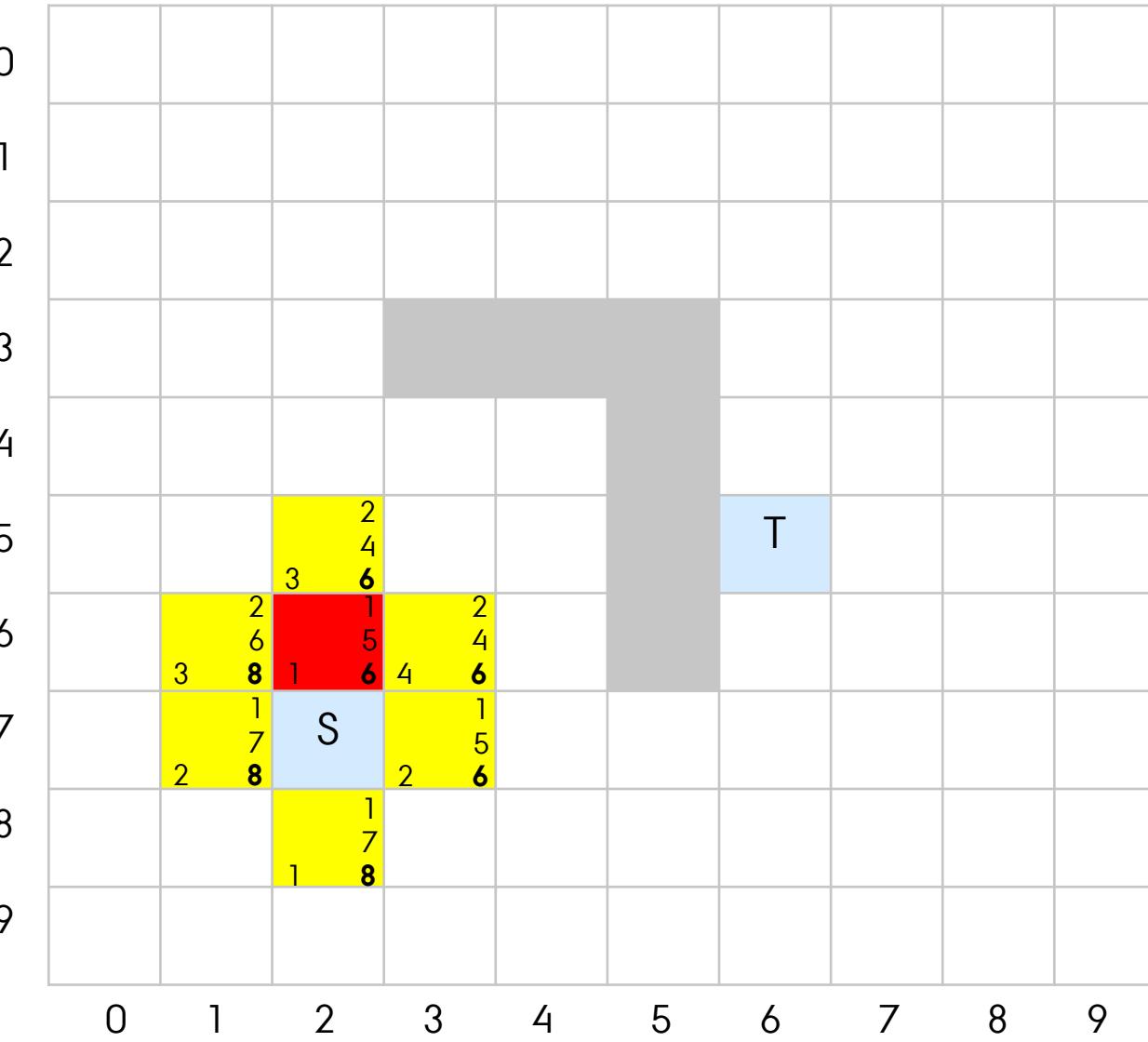
---

$g(n)=1$   
 $h(n)=5$   
 $p=1 \quad f(n)=\mathbf{6}$



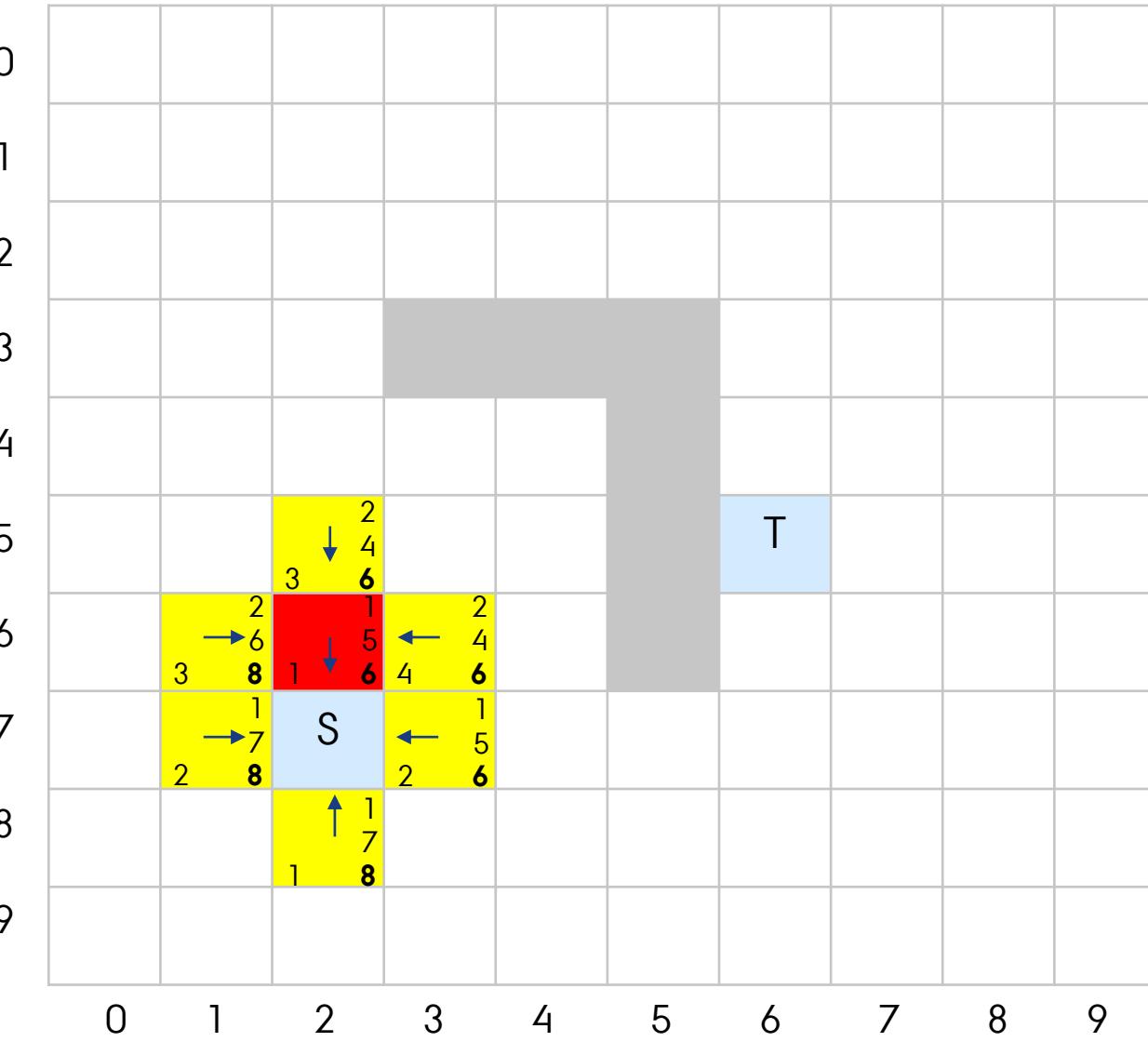
# EXAMPLE

---



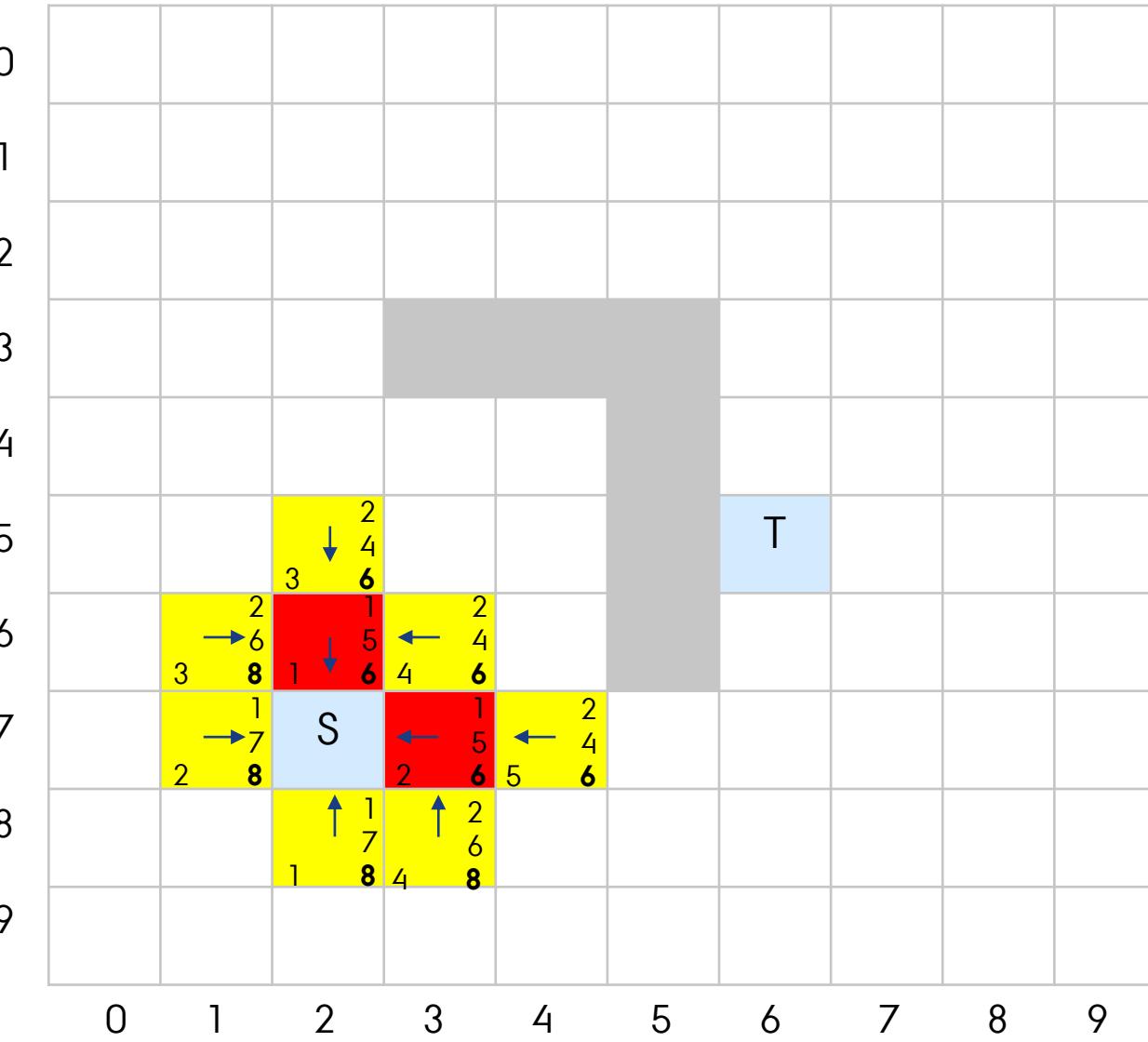
# EXAMPLE

---



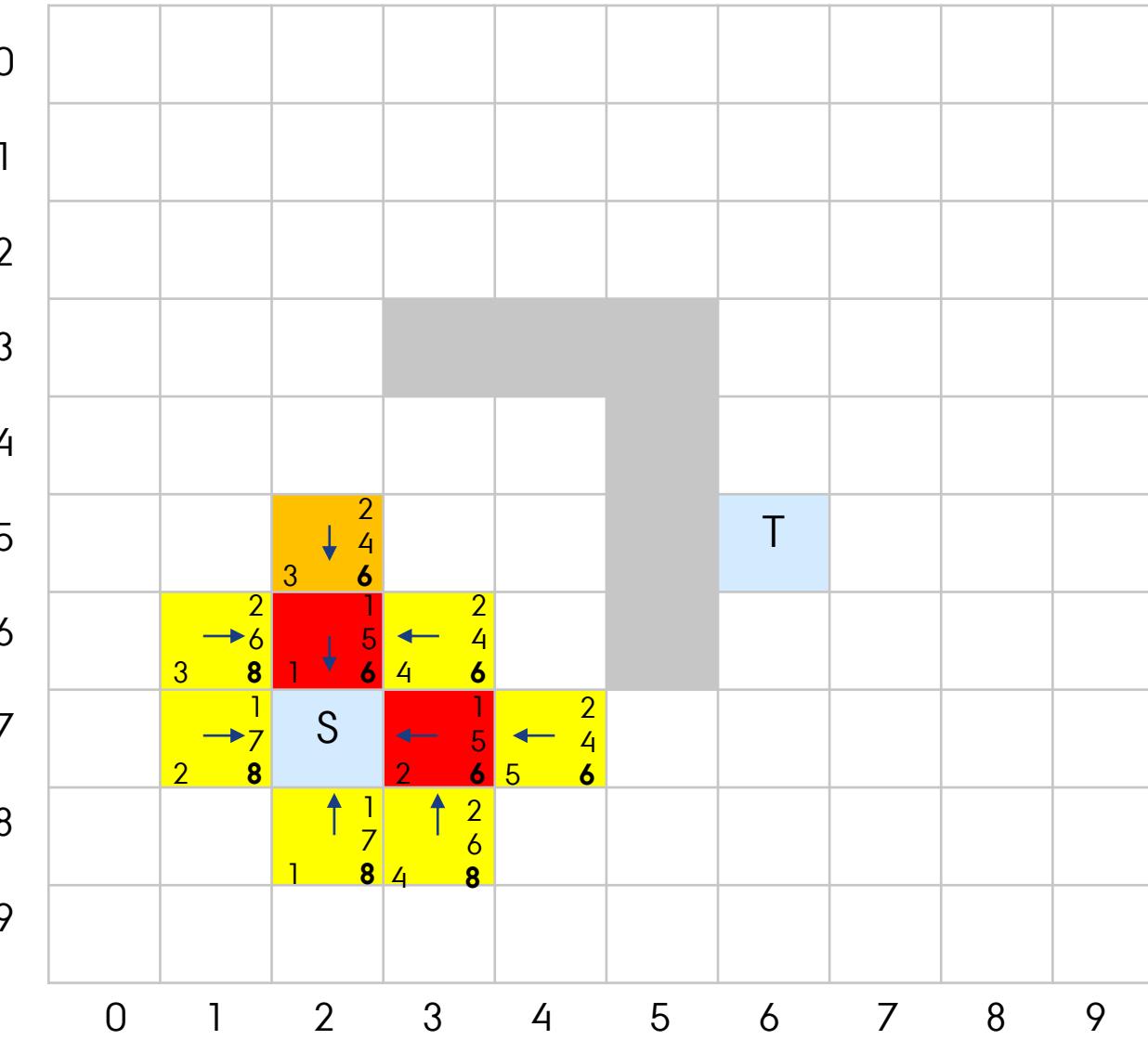
# EXAMPLE

---



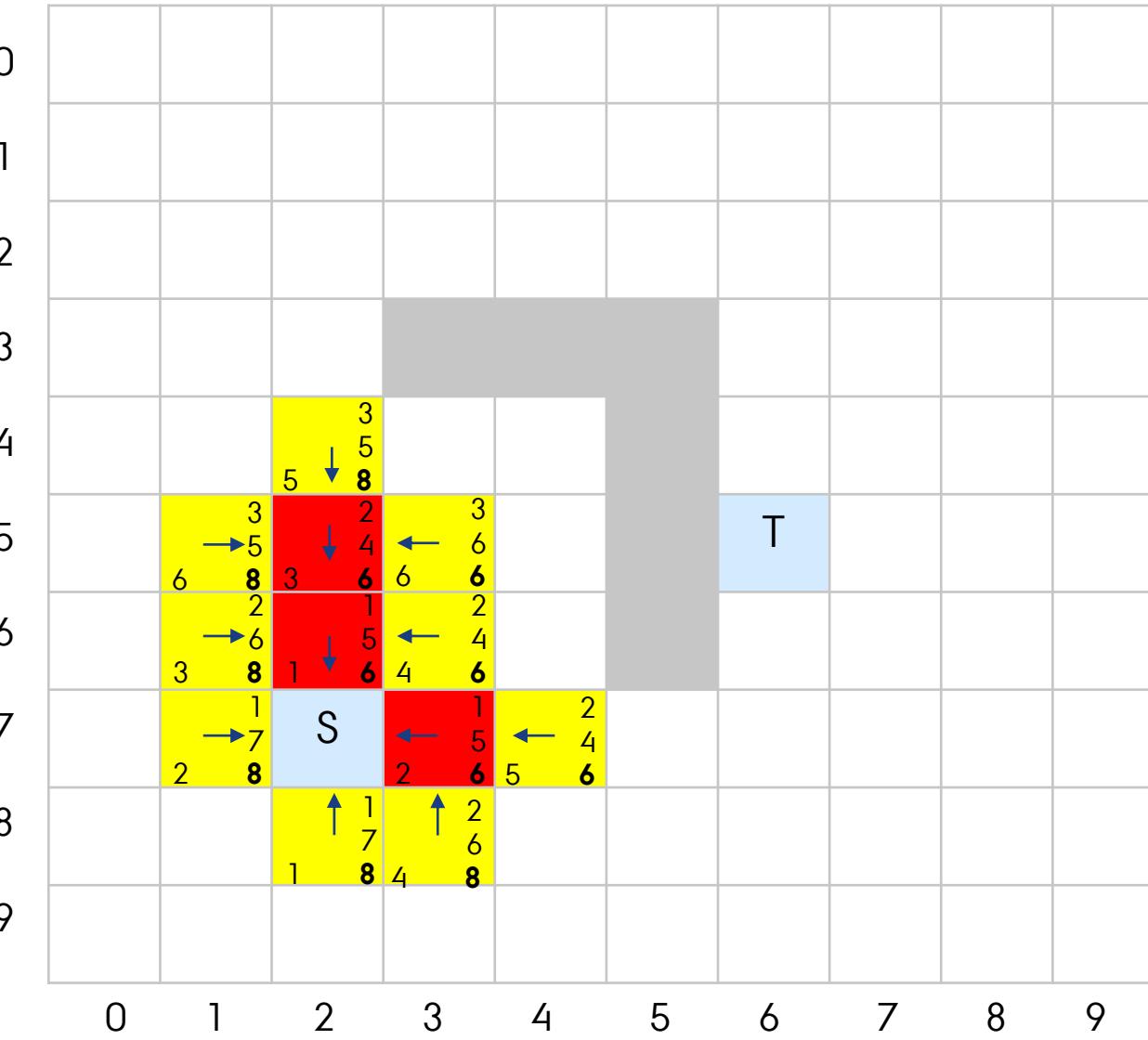
# EXAMPLE

---



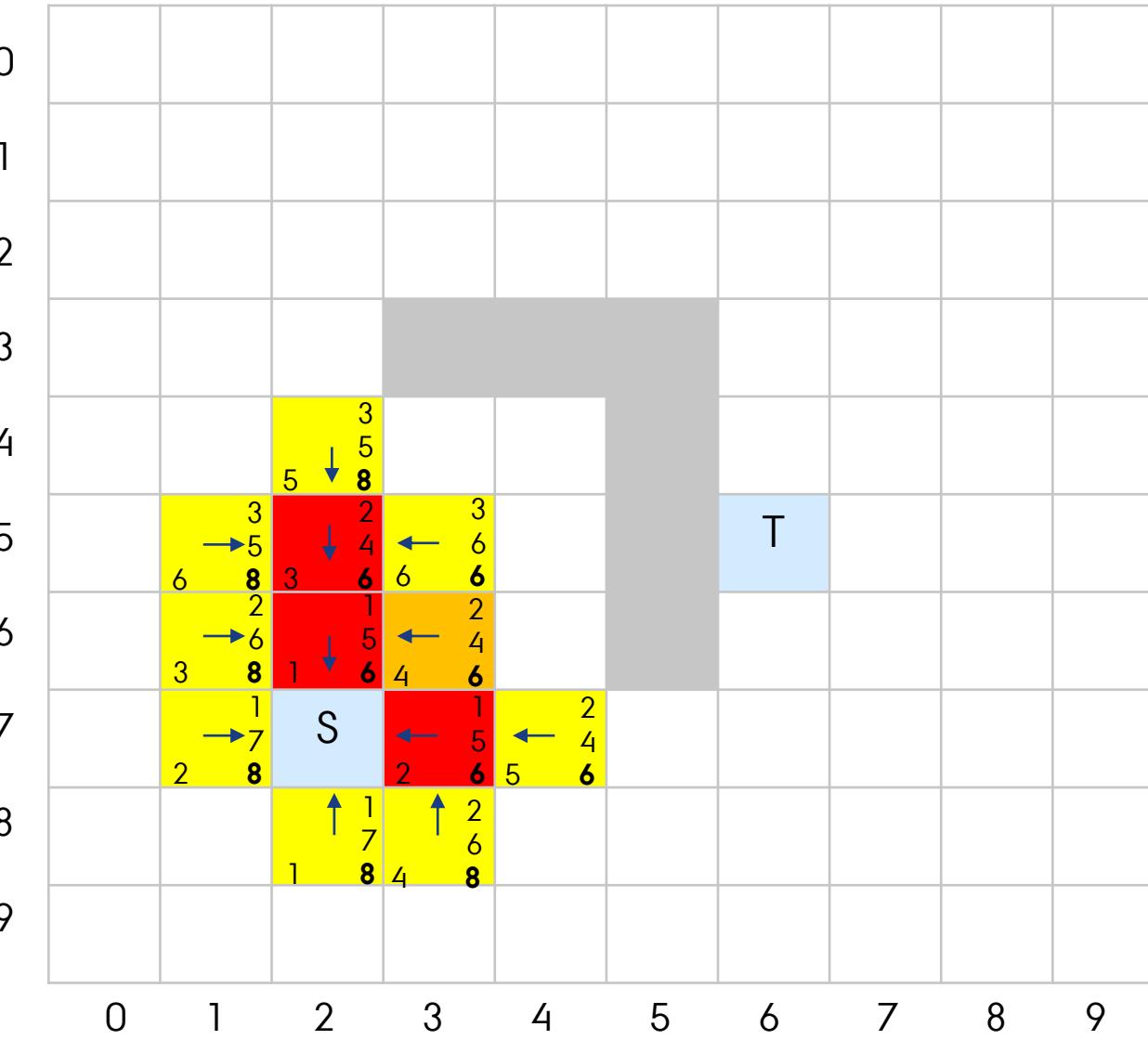
# EXAMPLE

---



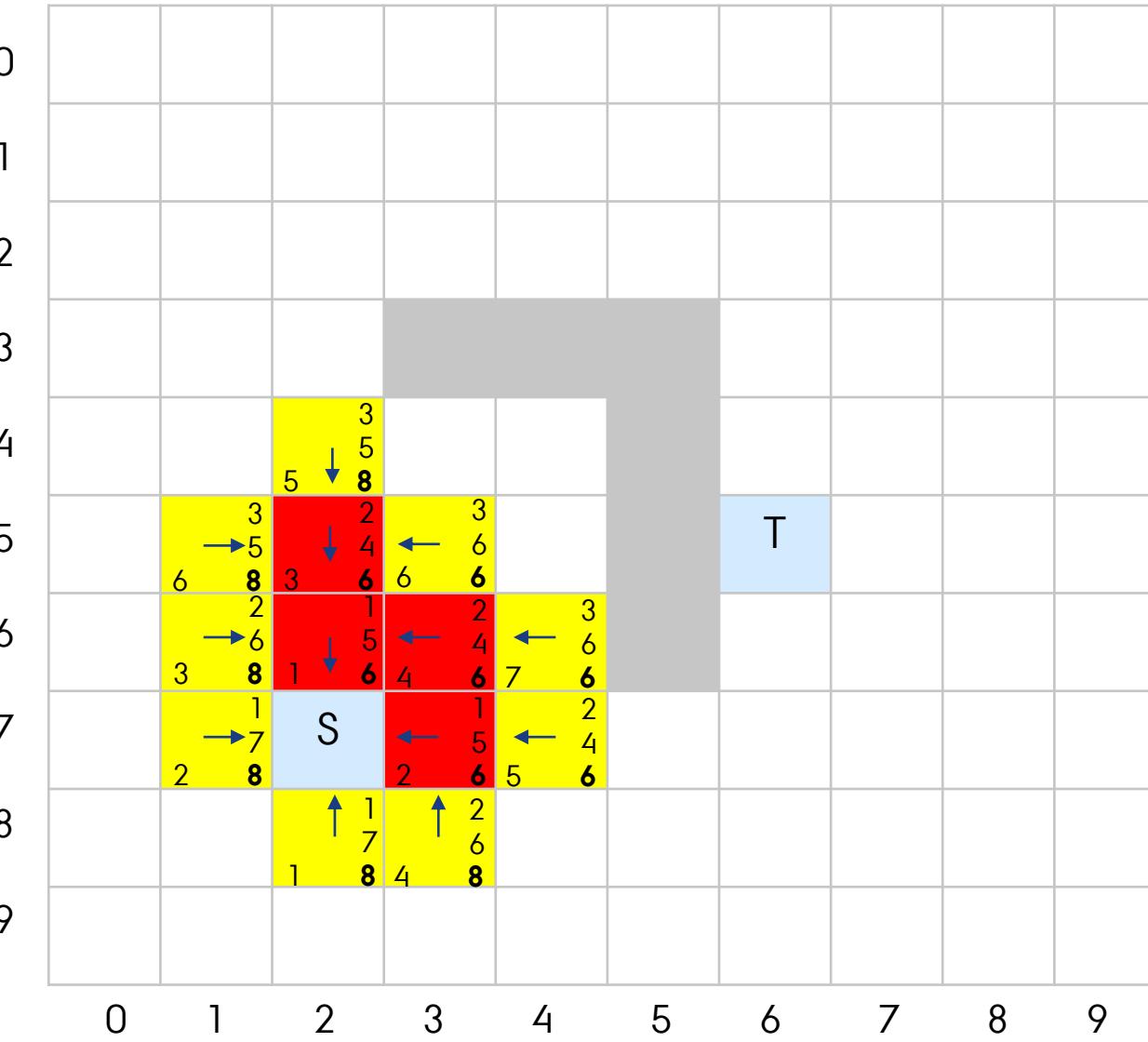
# EXAMPLE

---



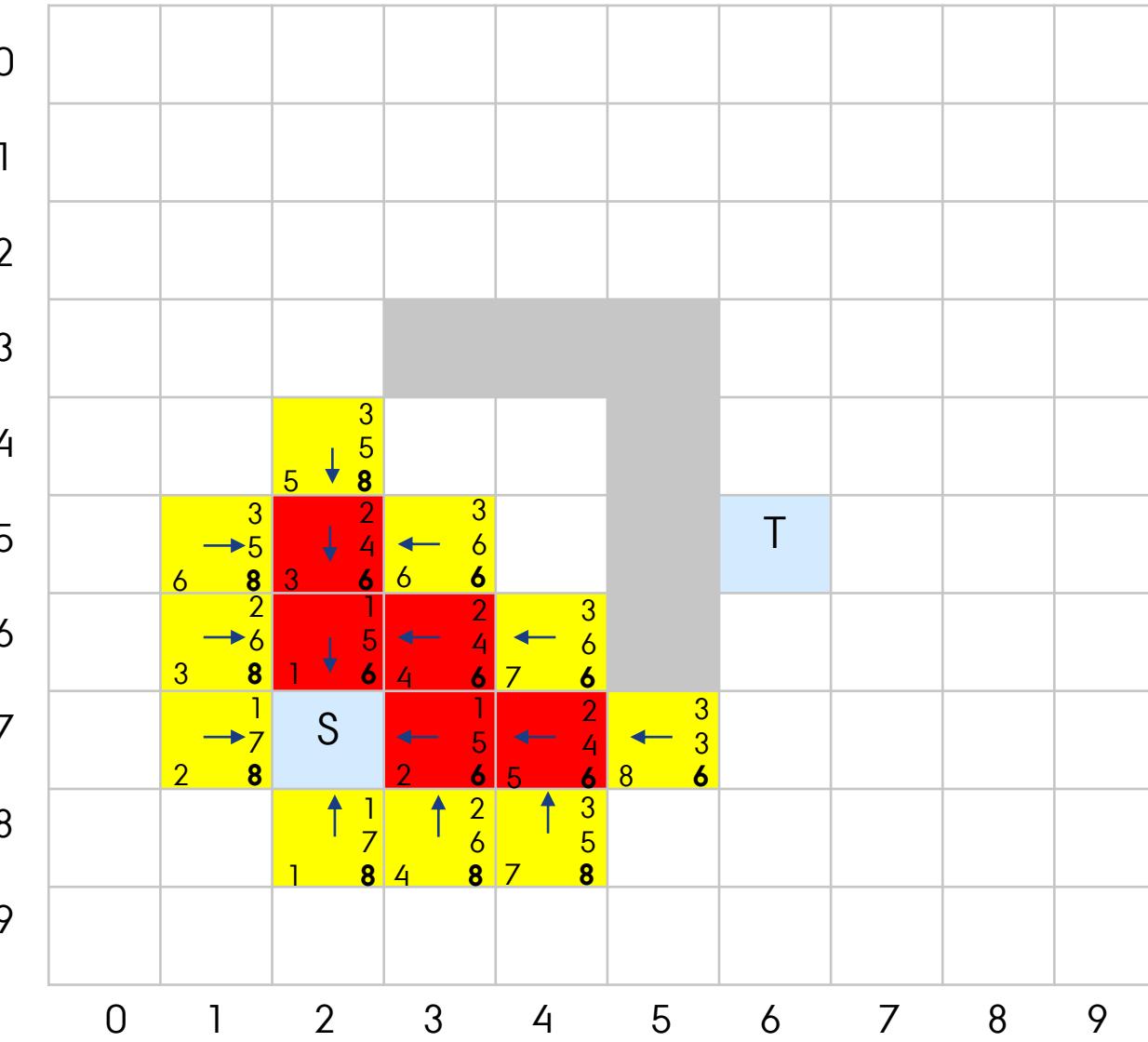
# EXAMPLE

---



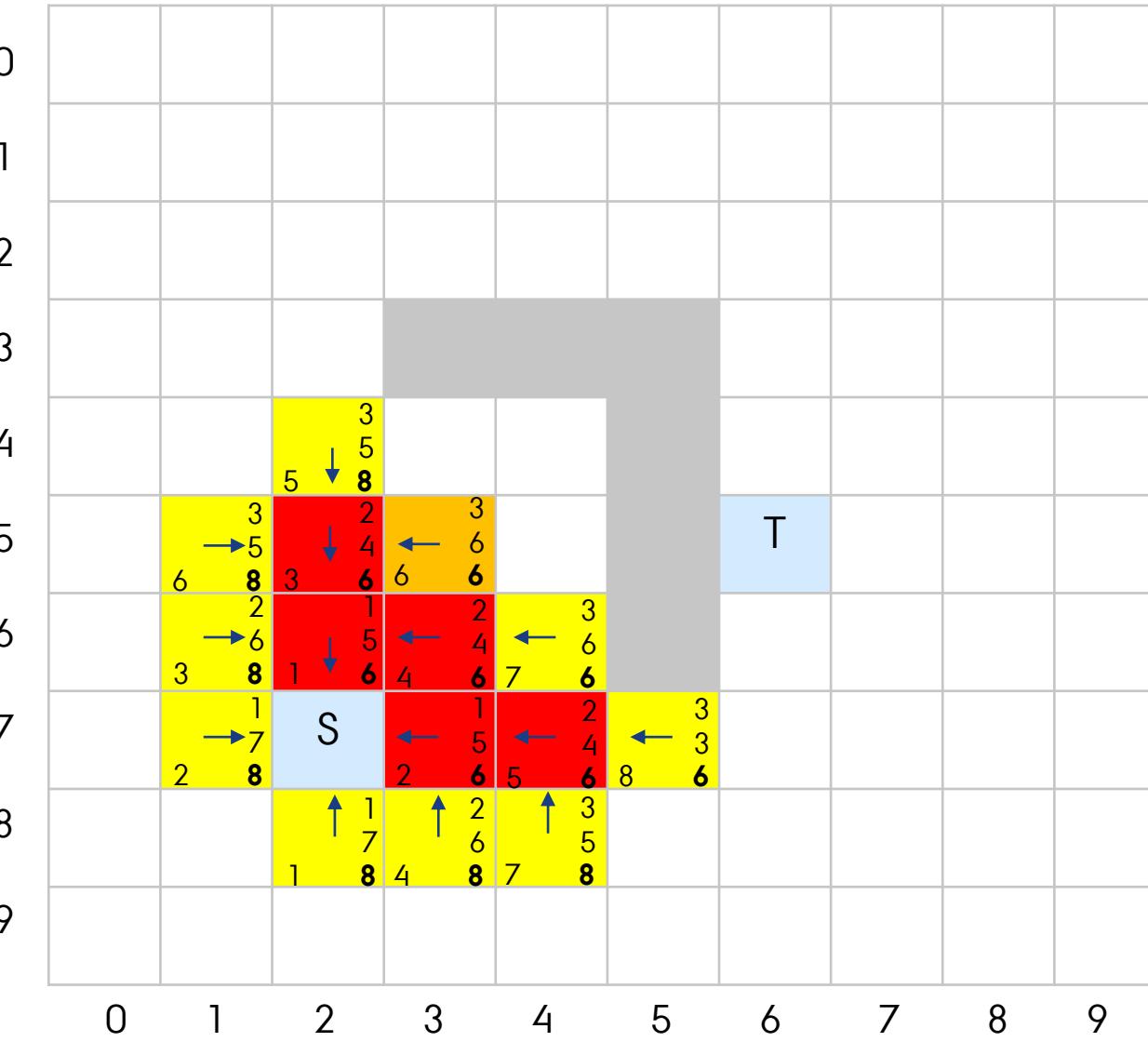
# EXAMPLE

---



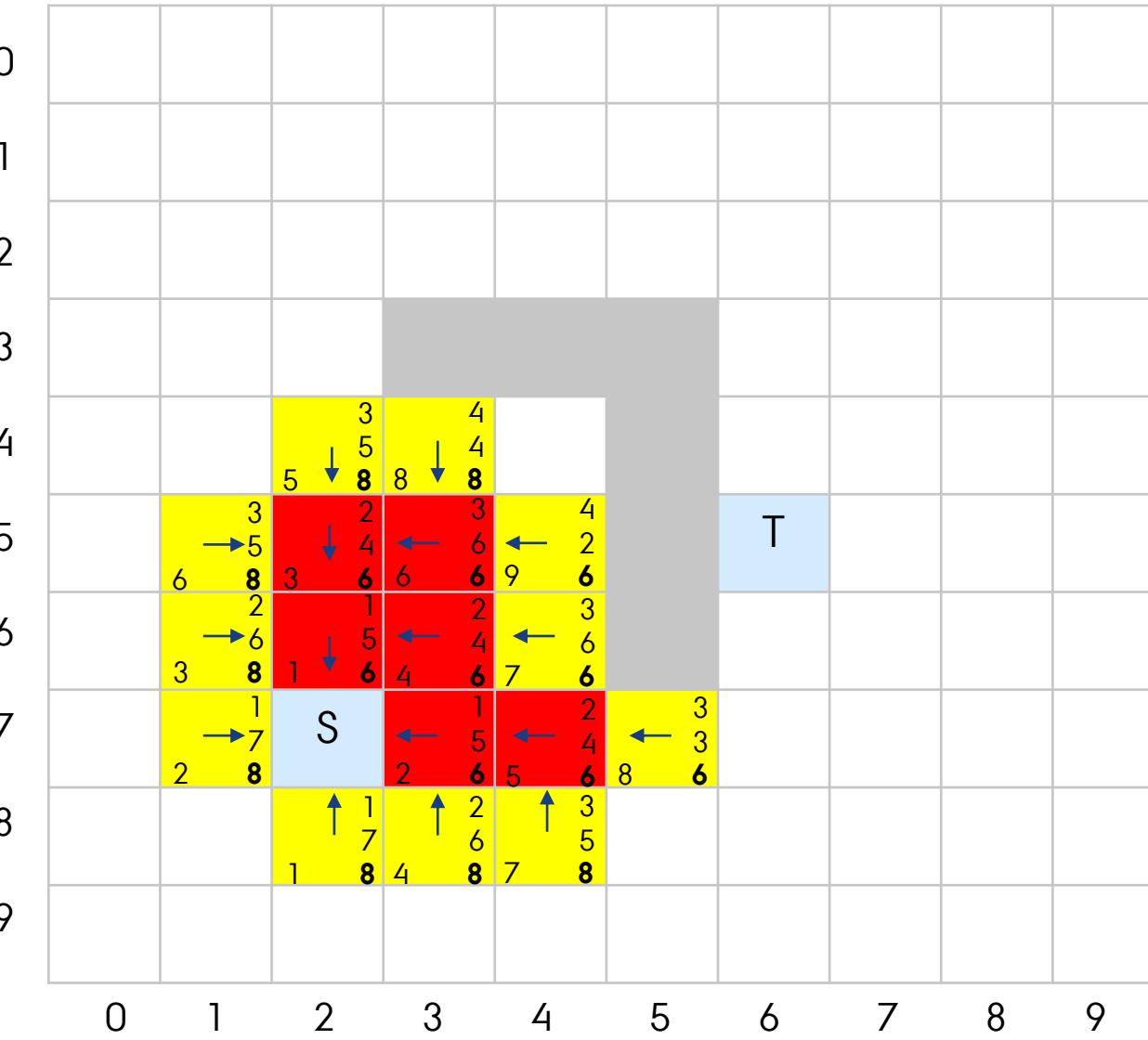
# EXAMPLE

---



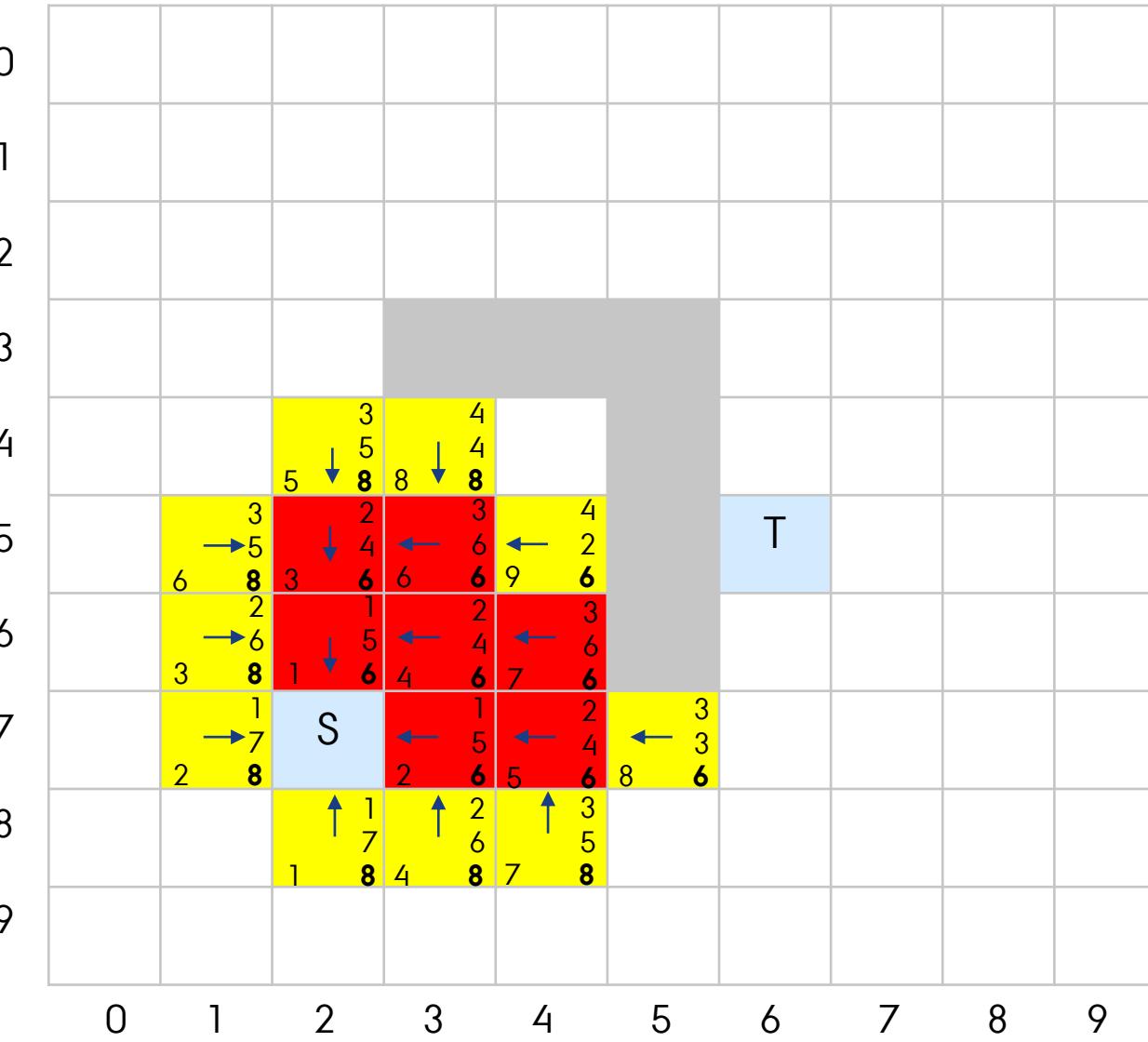
# EXAMPLE

---



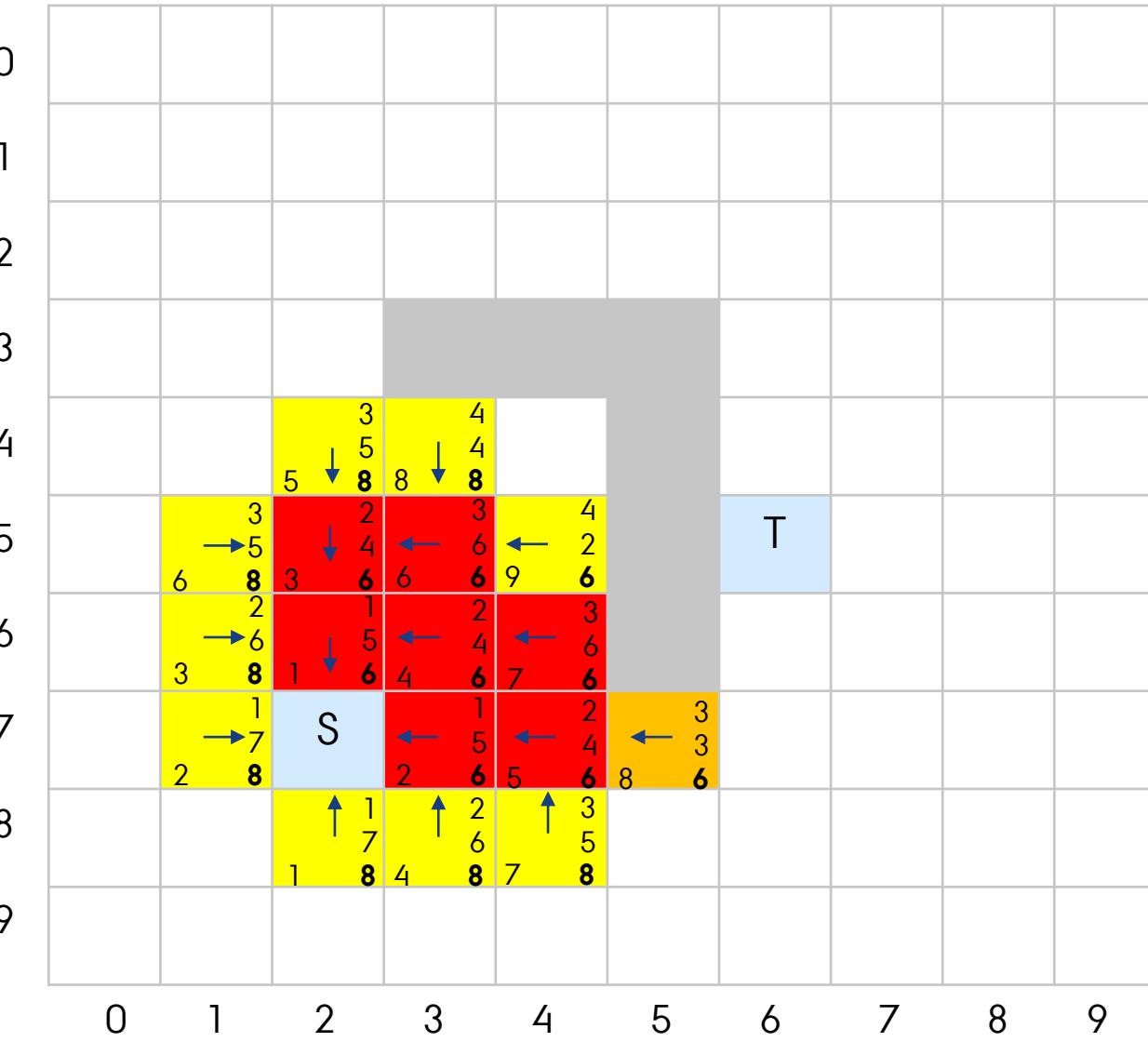
# EXAMPLE

---



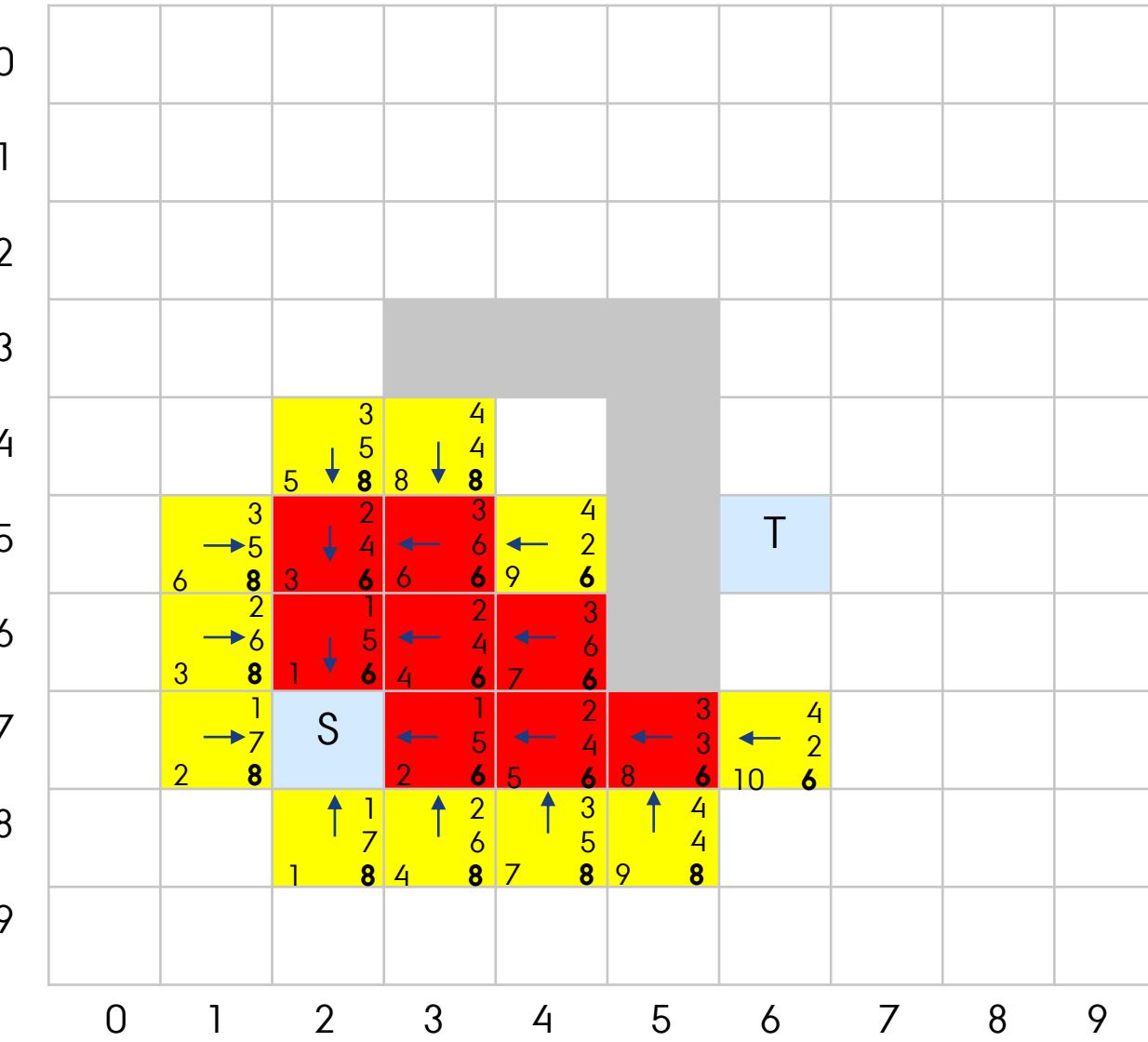
# EXAMPLE

---



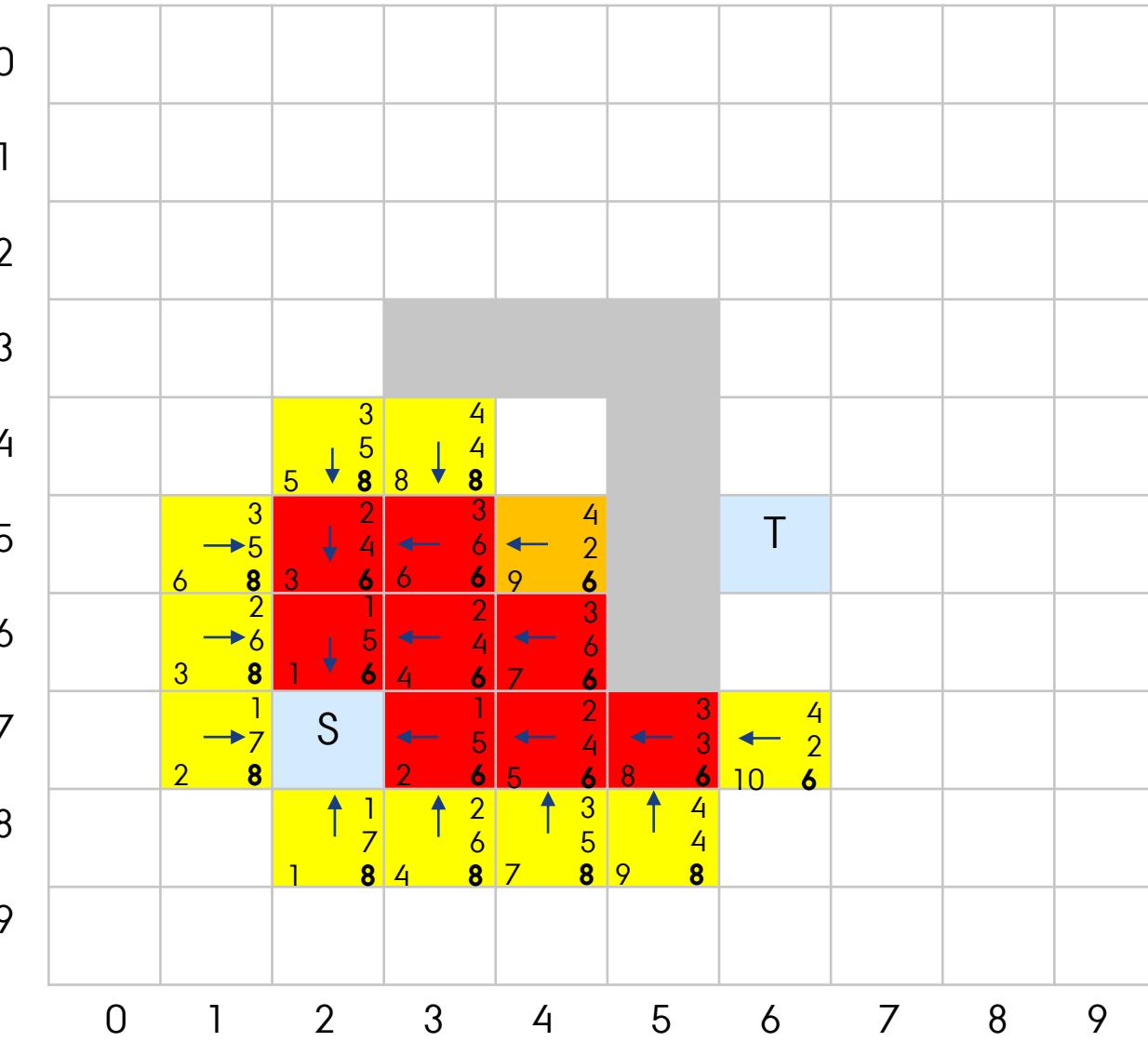
# EXAMPLE

---



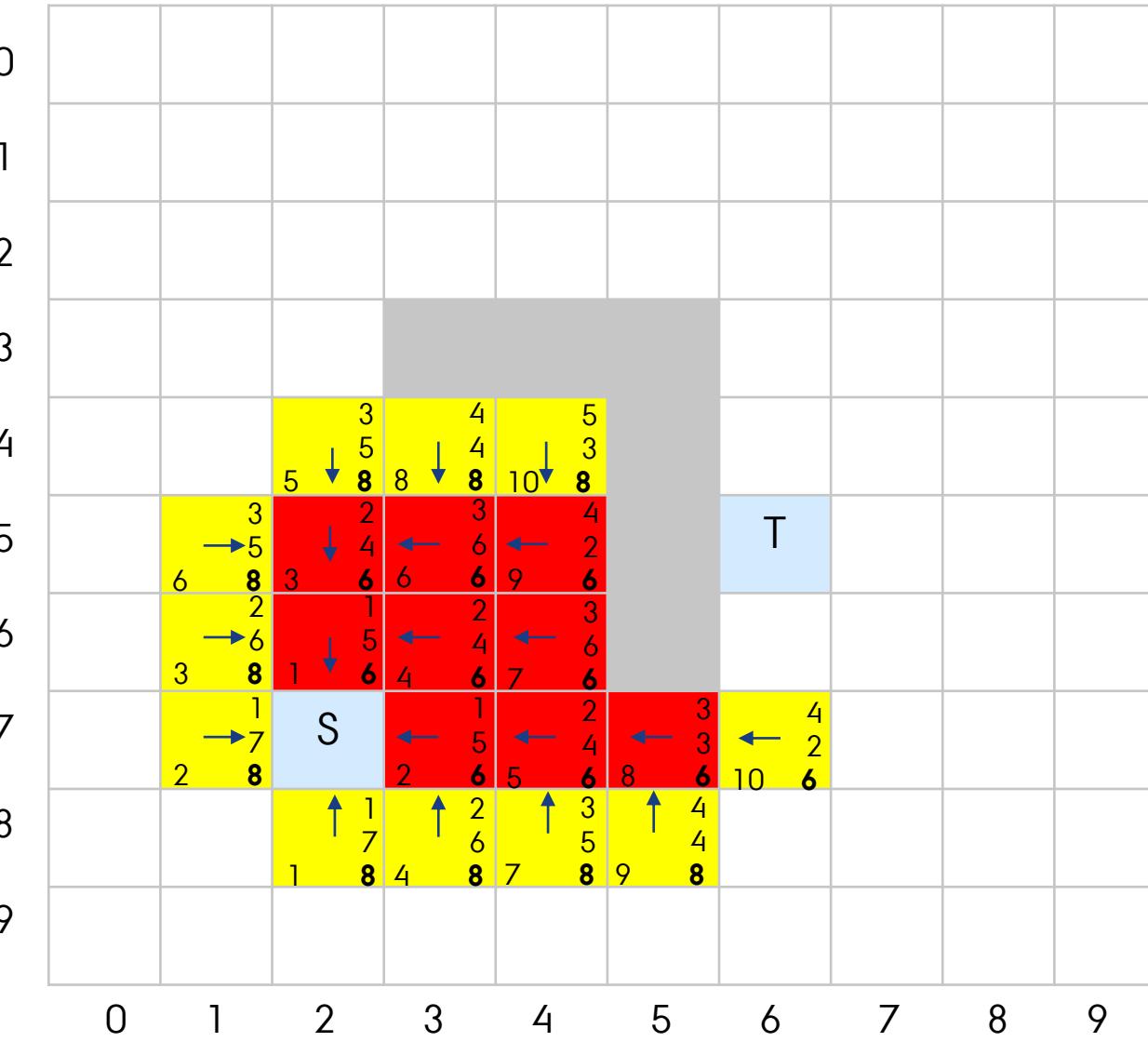
# EXAMPLE

---



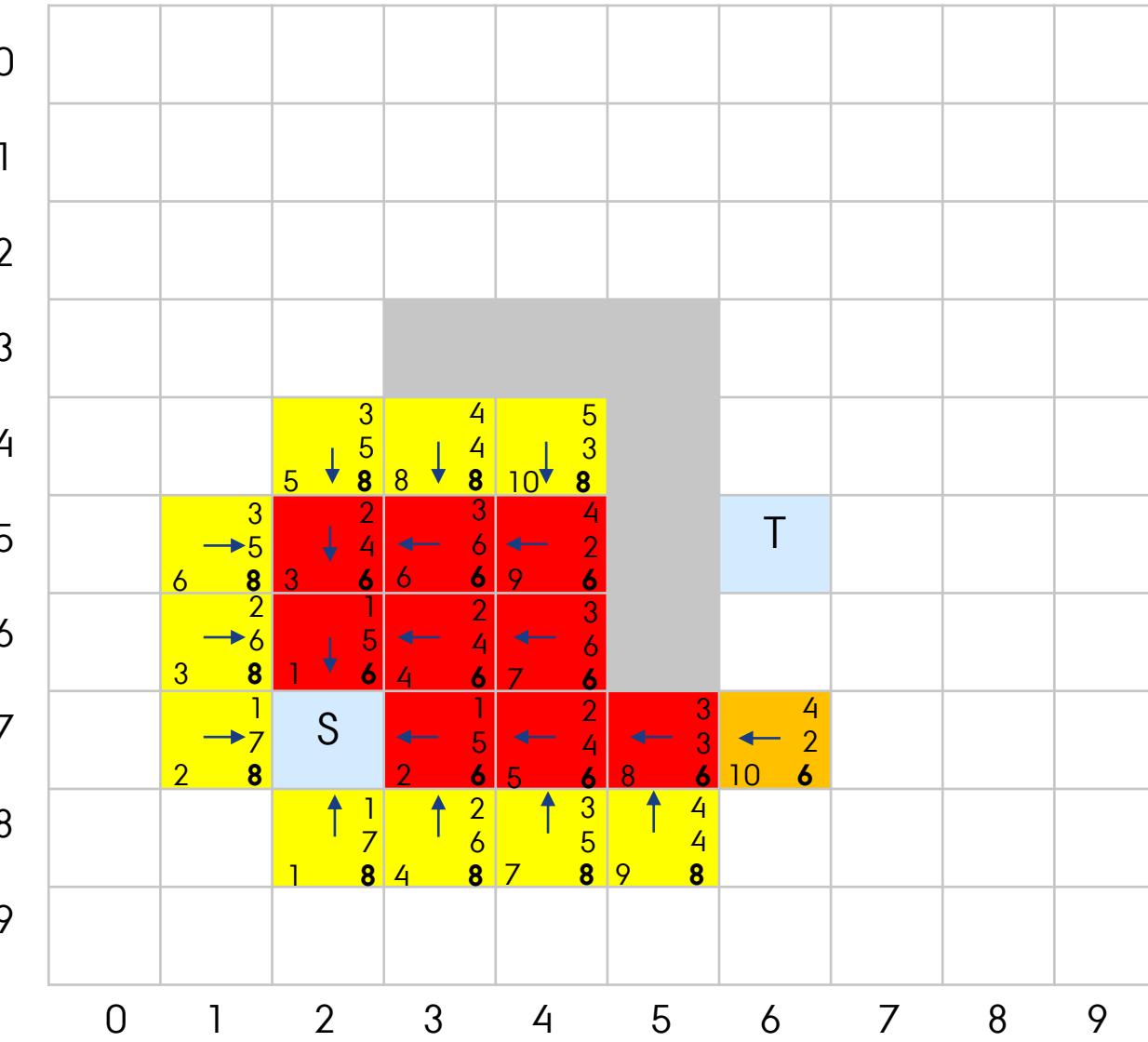
# EXAMPLE

---



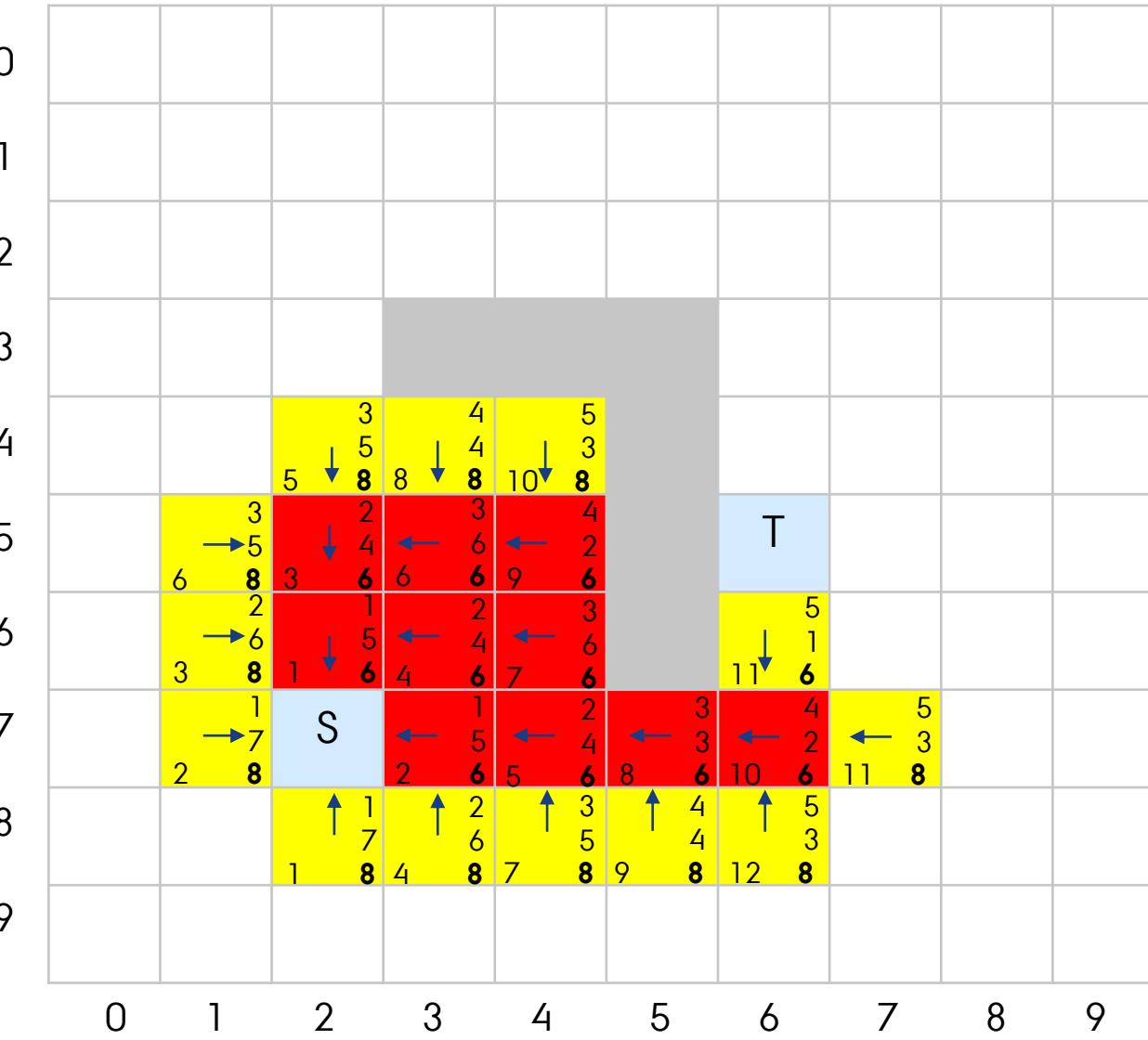
# EXAMPLE

---



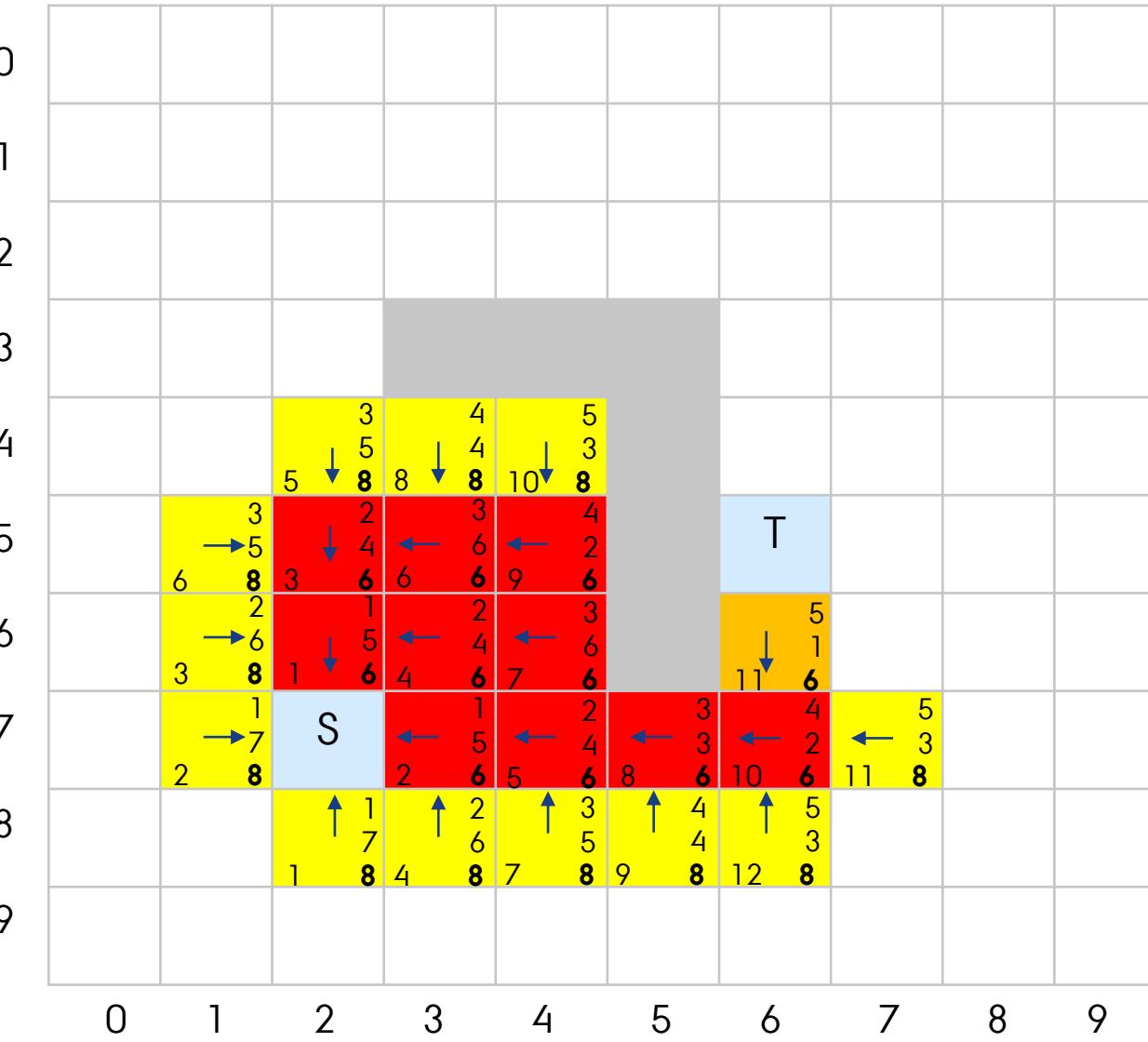
# EXAMPLE

---



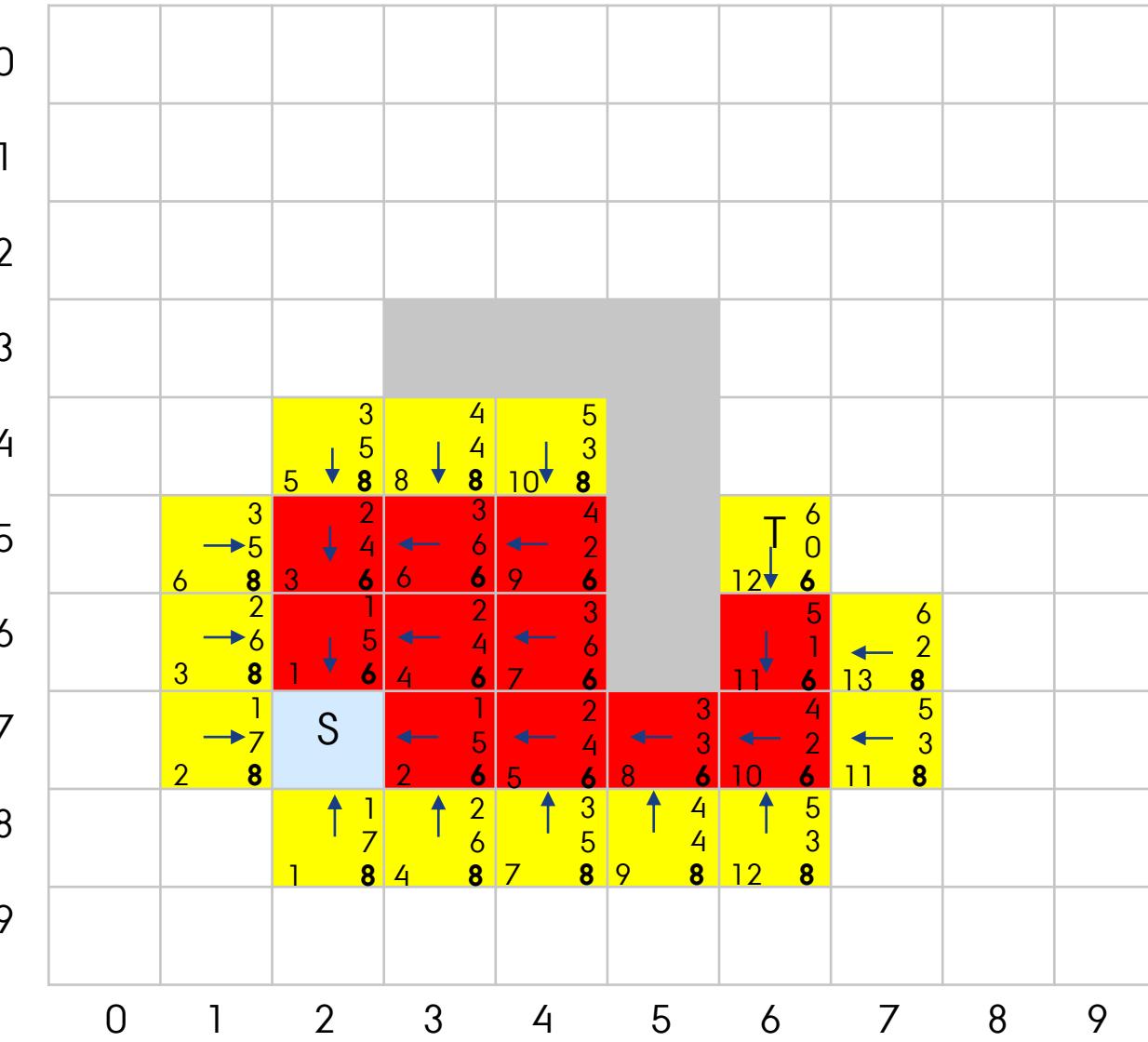
# EXAMPLE

---



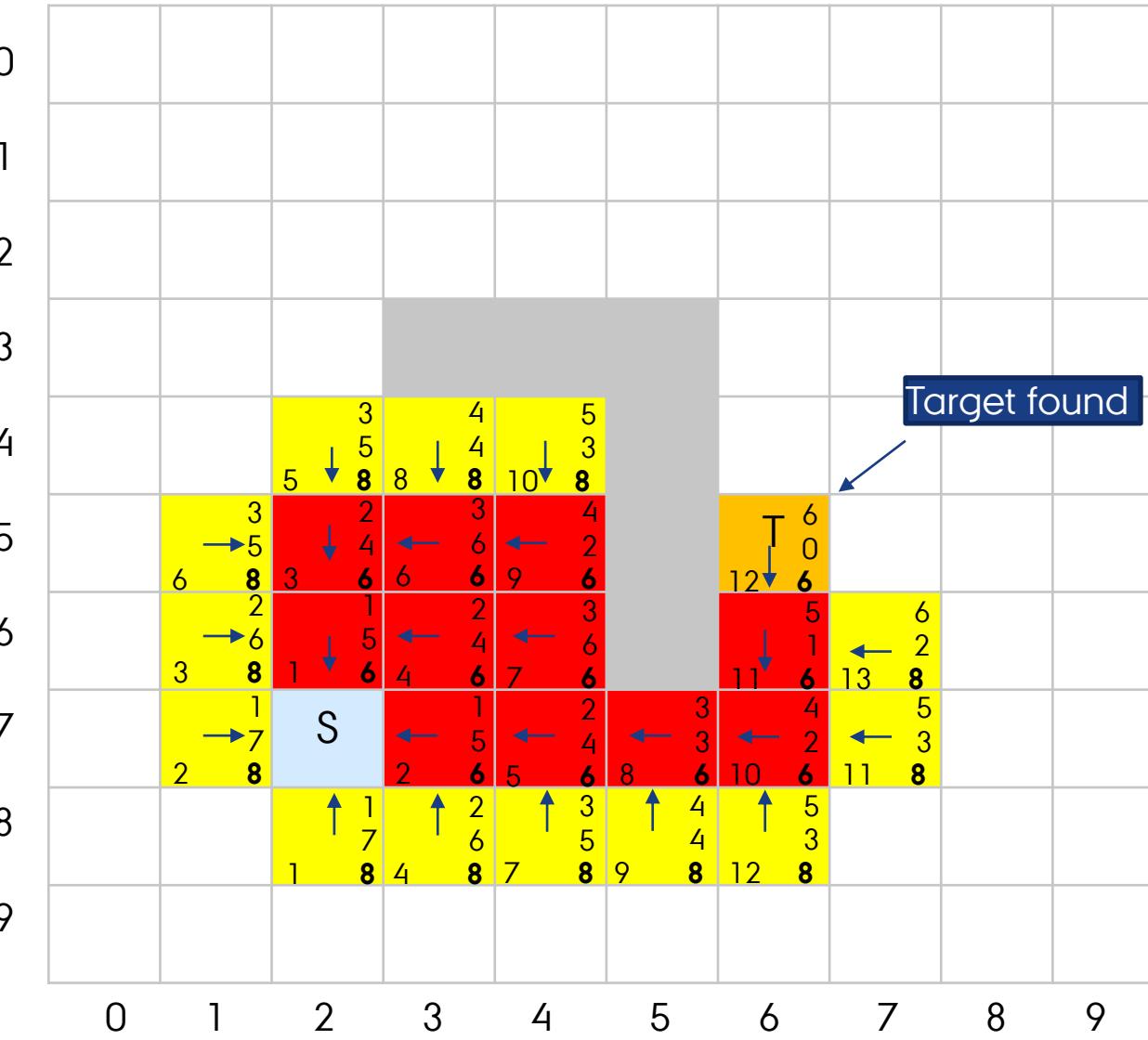
# EXAMPLE

---



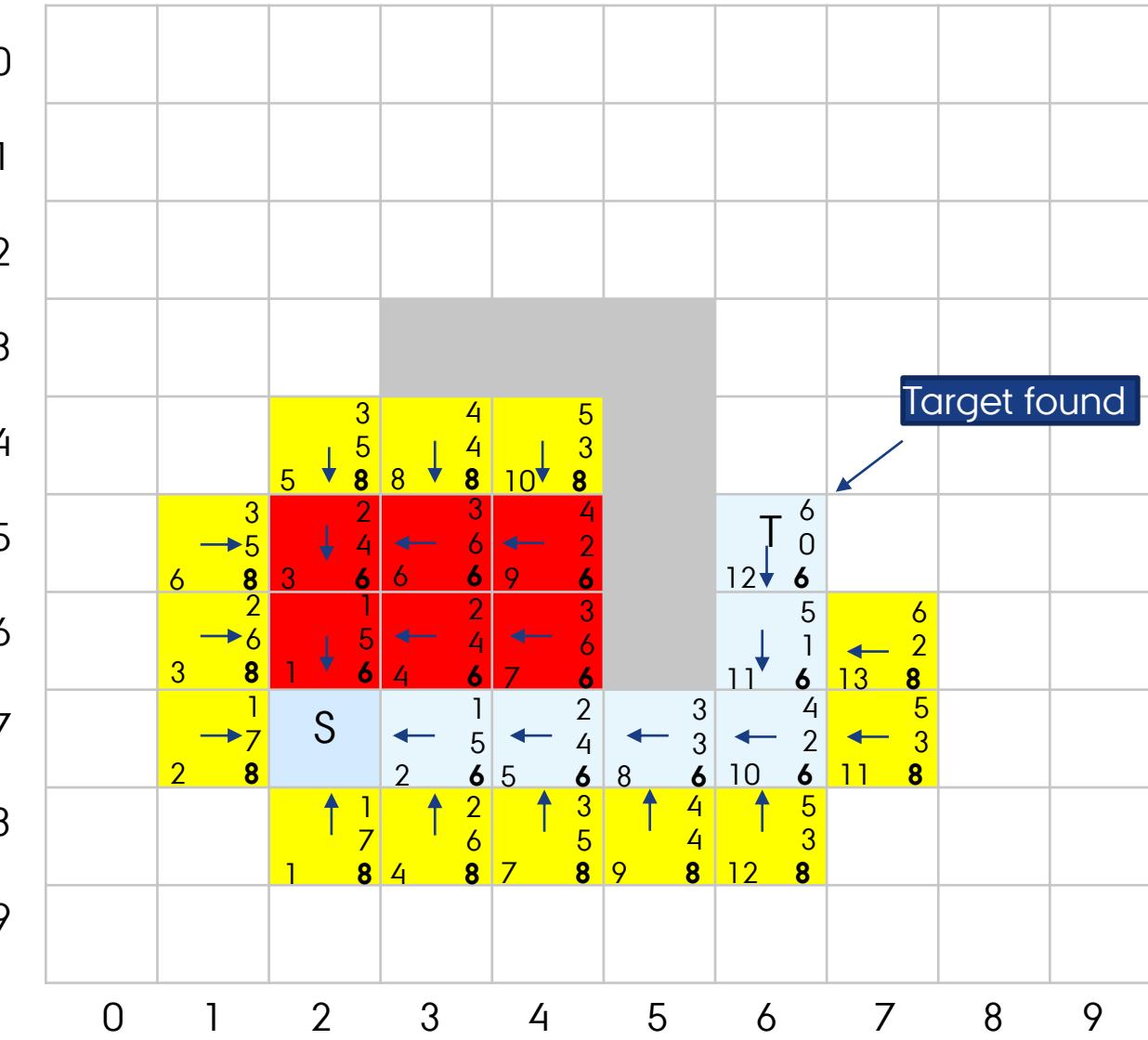
# EXAMPLE

---



# EXAMPLE

---



# GRAPH ALGORITHMS SHORTEST PATH + MST

# AGENDA

---

- All-pairs shortest-paths
- Minimum Spanning Tree (MST)
  - Prim's MST algorithm
  - Kruskal's MST algorithm

# ALL-PAIR SHORTEST PATHS

---

Let  $G$  a directed graph and suppose that every edge  $(u, v)$  is associated to a **weight** (cost, distance)  $W(u, v)$ .

## All-pairs shortest path problem:

Given a graph  $(G, E, W)$ , find a path of minimum weight for all pairs of vertices  $(u, v)$ .

If  $G$  does not have **negative-weight edges**, we can reuse Dijkstra's single-source shortest-path algorithm for all possible source vertices. *But what would be the complexity?*

# ALL-PAIR SHORTEST PATHS

---

We need to execute Dijkstra's algorithm  $|V|$  times.

To recap, the algorithm **extracts** the minimum  $|V|$  times and **decreases** distances at most  $|E|$  times. The complexity would be different with different data structures:

1. **Vector**:  $|V|O(|V|^2 + |E|) = O(|V|^3)$
2. **Min-heap**:  $|V|O(|E| \log |V|) = O(|V||E| \log |V|)$

If  $G$  has negative-weight edges, we can use **Bellman-Ford** ( $O(|V| \cdot |E|)$ ) with for an overall complexity of  $O(|V|^2 |E|)$ . If  $G$  is **dense**  $|E| = O(|V|^2)$  so overall  $O(|V|^4)$ .

# FLOYD-WARSHALL ALGORITHM

---

Let's study a method to solve the problem in an asymptotically faster way when  $G$  is a **dense** graph. The graph may have negative-weight edges, but not **negative cycles**.

The Floyd-Warshall algorithm is based on **dynamic programming** and solves the problem in time  $O(V^3)$ .

We shall adopt the convention that if  $(i, j)$  is not an edge in  $G$ , then  $W(i, j) = \infty$ .

# IDEA

---

Start with all the known vertices' cost and put that in a  $N \times N$  matrix ( $N = |V|$ )

Go through all vertices from 1 to  $N$  ( $v_k$ ):

For all  $i, j$  in  $1..N$ : If the cost of going from  $i$  to  $j$  via  $k$  is less than the original cost  
update the cost to that

In pseudocode:

```
let V = number of vertices in graph
let dist = V × V array of minimum distances initialized to ∞
for each vertex v
    dist [v][v] ← 0
for each edge (u,v)
    dist [u][v] ← weight(u,v)
for k from 1 to V
    for i from 1 to V
        for j from 1 to V
            if dist [i][j] > dist [i][k] + dist [k][j]
                dist [i][j] ← dist [i][k] + dist [k][j]
            end if
```

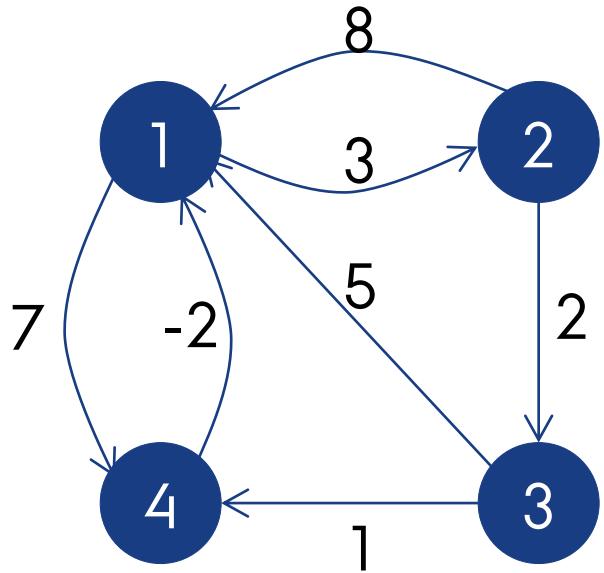
# REFLECTION

---

1. In the following five slides, ensure that you relate the steps to the algorithm
2. Convince yourselves that the C++ code for the Floyd-Warshall algorithm implements the algorithm described on the previous slide
3. Do the Floyd-Warshall warm-up exercise



Photo by [Anthony Tran](#) on [Unsplash](#)



$$A^0 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & \infty \\ 3 & 5 & \infty & 0 & 1 \\ 4 & -2 & \infty & \infty & 0 \end{pmatrix}$$

---

Check if going via 1 makes it “cheaper”:

- 1<sup>st</sup> row, 1<sup>st</sup> column and the diagonal can be copied (going from 1 to  $x$  via 1 will not make it “cheaper”)

$$\mathbf{A}^1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ -2 & & & 0 \end{pmatrix} \end{matrix}$$

- Is it less to go from 2 to 3 via 1?  $A^0[2,3] = 2$ ;  $A^0[2,1] = 8$ ;  $A^0[1,3] = \infty$ , i.e. going via 1 will cost  $8 + \infty$ , so there is no better path than previous, i.e.  $A^1[2,3] = 2$
- Is it less to go from 2 to 4 via 1?  $A^0[2,4] = \infty$ ;  $A^0[2,1] = 8$ ;  $A^0[1,4] = 7$ , i.e. going via 1 will cost  $8 + 7$ , so there is a better path than previous, i.e.  $A^1[2,4] = 15$

# VIA “1”

---

Continuing gives

$$\mathbf{A}^1 = \begin{pmatrix} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & -2 & 1 & \infty & 0 \end{pmatrix}$$

# VIA “2”

---

Copy row and column 2 + diagonal

$$A^2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \\ 2 & 8 & 0 & 2 & 15 \\ 3 & & 8 & 0 & \\ 4 & & & 1 & 0 \end{pmatrix}$$

- Is it “cheaper” to go from 1 to 3 via 2?  $A^1[1,3] = \infty$ ;  $A_1[1,2] = 3$ ;  $A_1[2,3] = 2$ , i.e. going via 2 will cost  $3+2$ , so there is a better path than previous, i.e.  $A_2[1,3] = 5$

# VIA “2” (2)

---

Continuing gives

$$\mathbf{A}^2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ -2 & 1 & 3 & 0 \end{pmatrix}$$

Then via 3 and via 4 giving:

$$\mathbf{A}^3 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ -2 & 1 & 3 & 0 \end{pmatrix}$$

$$\mathbf{A}^4 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & 5 & 6 \\ 1 & 0 & 2 & 3 \\ -1 & 2 & 0 & 1 \\ -2 & 1 & 3 & 0 \end{pmatrix}$$

# FLOYD-WARSHALL ALGORITHM

The algorithm starts by initializing the matrix using the weights and proceeds by **optimizing** the distance matrix in  $O(V^3)$ .

```
118 void Graph::allPairs(Matrix<int>& path, Matrix<int>& dist) {  
119     int n = adj.size();  
120     // Initialize d and path  
121     for(int i = 0; i < n; ++i) {  
122         for(int j = 0; j < n; ++j) {  
123             dist[i][j] = weight[i][j];  
124             path[i][j] = -1;  
125         }  
126     }  
127  
128     for(int k = 0; k < n; ++k) {  
129         // Consider each vertex as an intermediate  
130         for(int i = 0; i < n; ++i) {  
131             for(int j = 0; j < n; ++j) {  
132                 if(dist[i][k] + dist[k][j] < dist[i][j]) {  
133                     // Update shortest path  
134                     dist[i][j] = dist[i][k] + dist[k][j];  
135                     path[i][j] = k;  
136                 }  
137             }  
138         }  
139     }  
140 }
```

graph\_class.cpp

# AGENDA

---

- ✓ All-pairs shortest-paths
- Minimum Spanning Tree (MST)
  - Prim's MST algorithm
  - Kruskal's MST algorithm

# MINIMUM SPANNING TREE

---

Suppose we want to solve the following problem. Given a set of computers, where each pair of computers can be connected using fiber, find an *interconnecting network* using the **least amount of fiber** possible.

This problem can be modeled as a graph problem, where the graphs are **undirected** and **weighted**, and vertices represent the computers. Edges represent the **connections** that can be built and the weights of edges represent the amount of fiber needed.

# MINIMUM SPANNING TREE

---

A **spanning graph** of  $G$  is a subgraph  $H$  with  $V' = V$  and  $E' \subseteq E$ .

In this model, the problem we want to solve is to find the **connected spanning graph** (which contains all vertices in the original graph) with minimum sum of weights in its edges.

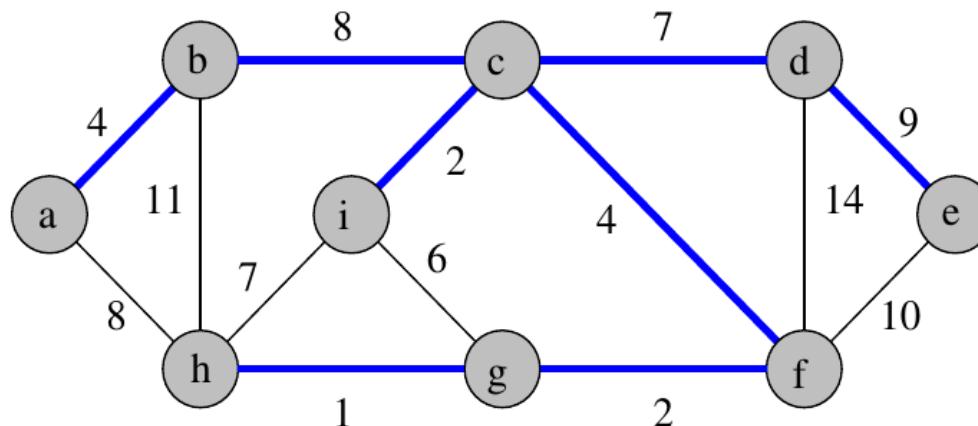
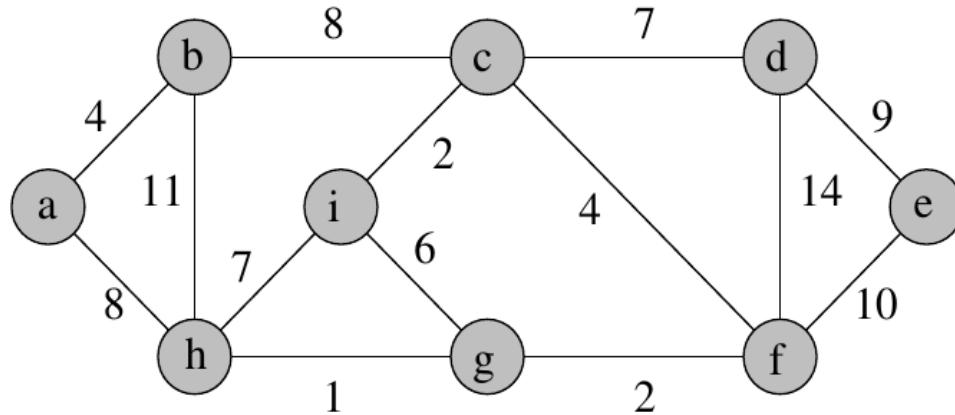
## Minimum Spanning Tree problem:

Given a graph  $(V, E, W)$ , compute the connected spanning subgraph  $T$  of  $G$  with minimum total weight.

Naturally, the problem can only be solved if the graph is **connected**.

# MINIMUM SPANNING TREE

---



# MINIMUM SPANNING TREE

---

We will see two classical **greedy** algorithms to solve the problem. A **generic** greedy approach solves the problem by building the MST **incrementally**:

1. The algorithm maintains a set of edges  $A$  such that, at the beginning of each iteration,  $A$  is **contained** in an MST.
2. At each iteration, we determine an edge  $(u, v)$  of minimal weight such that  $A' = A \cup (u, v)$  also satisfies the property.
3. This edge is called a **light edge** for  $A$ .

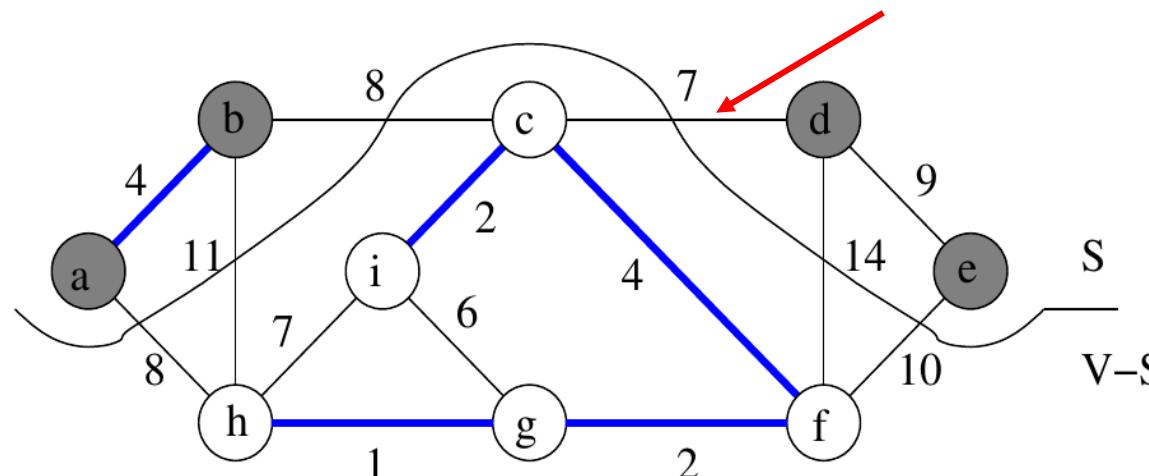
→ *How to find a light edge?*

# MINIMUM SPANNING TREE

---

Consider a graph  $G = (V, E)$  and let  $S \subset V$  (vertices in **gray**).

Consider  $A$  to be a subset of an MST and the set of edges in  $G$  with an end in  $S$  and another in  $V - S$ . We call this set a **cut**.



An edge **crossing** the cut is **light** if it has minimum weight. The two algorithms will specialize the generic approach using this property.

# REFLECTION

---

1. Give another example of the use of a MST
2. On the slide with the MST example, is the weight on the edge between l and h was 1 instead of 7, two different MST exist – which ones?
3. Do the MST warm-up exercise



Photo by [Anthony Tran](#) on [Unsplash](#)

# AGENDA

---

- ✓ All-pairs shortest-paths
- ✓ Minimum Spanning Tree (MST)
- Prim's MST algorithm
- Kruskal's MST algorithm

# PRIM'S ALGORITHM

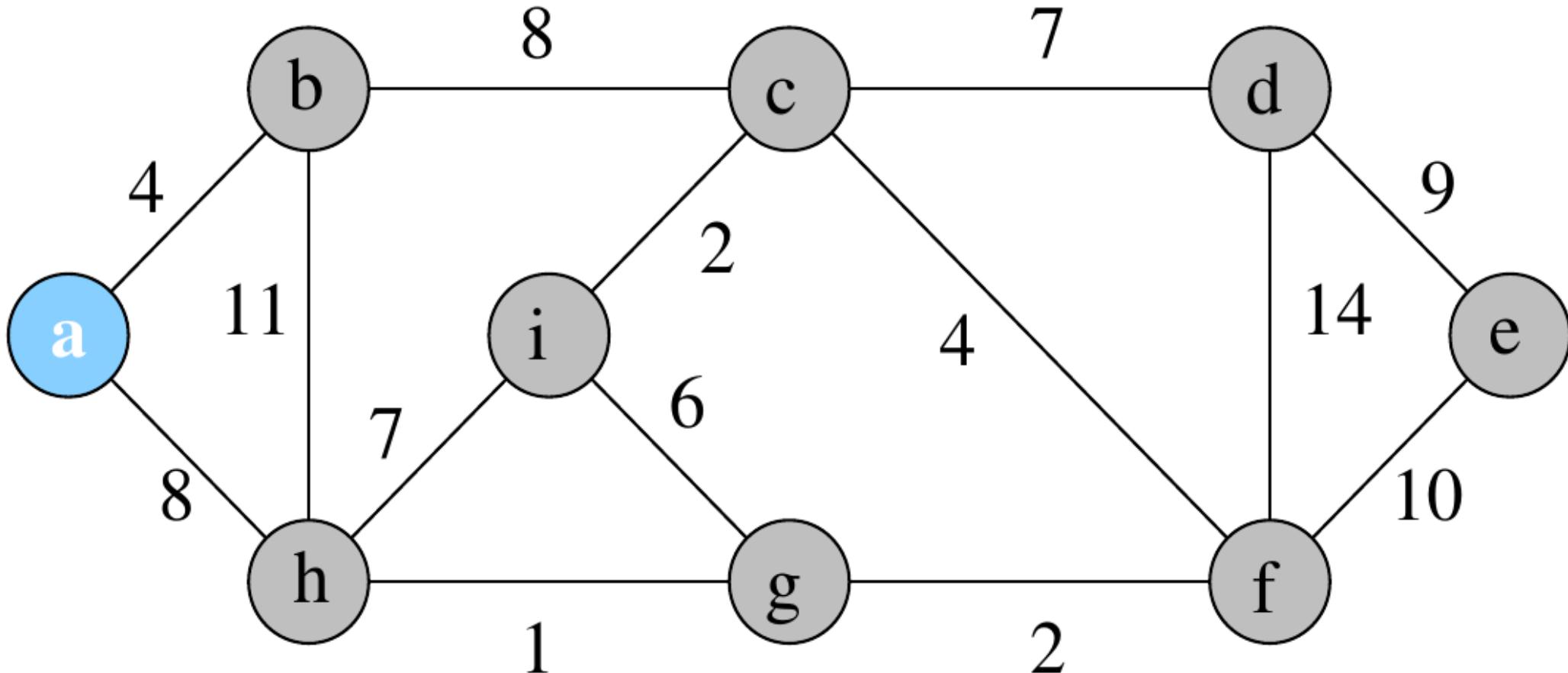
---

In *Prim's algorithm*, the set  $A$  is a tree with some root  $r$  (arbitrarily chosen at the beginning).

1. Initially,  $A$  is **empty**.
2. At each iteration, the algorithm considers a cut of the set of vertices which are endpoints in  $A$ .
3. The algorithm finds a **light** edge  $(u, v)$  in this cut and adds it to the set  $A$  and starts another iteration until  $A$  is a MST.

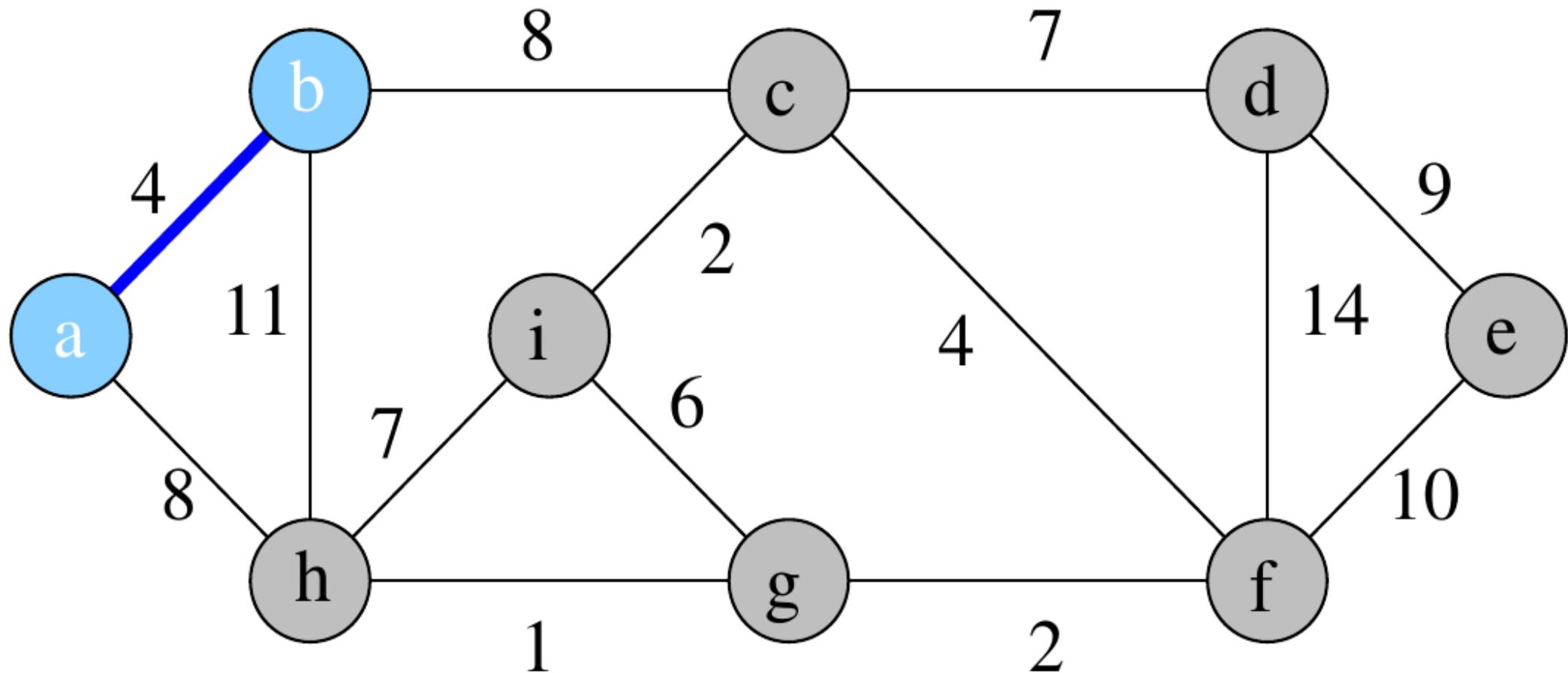
# PRIM'S ALGORITHM

---



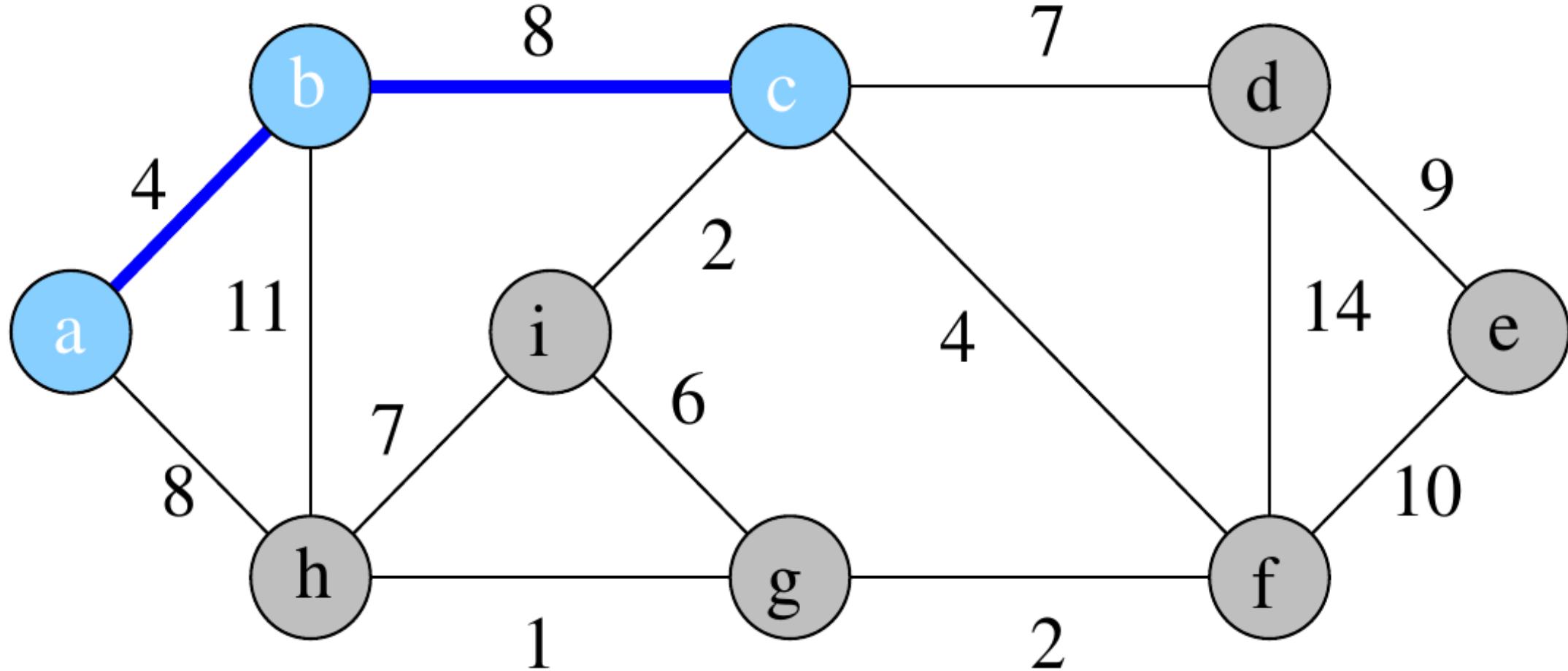
# PRIM'S ALGORITHM

---



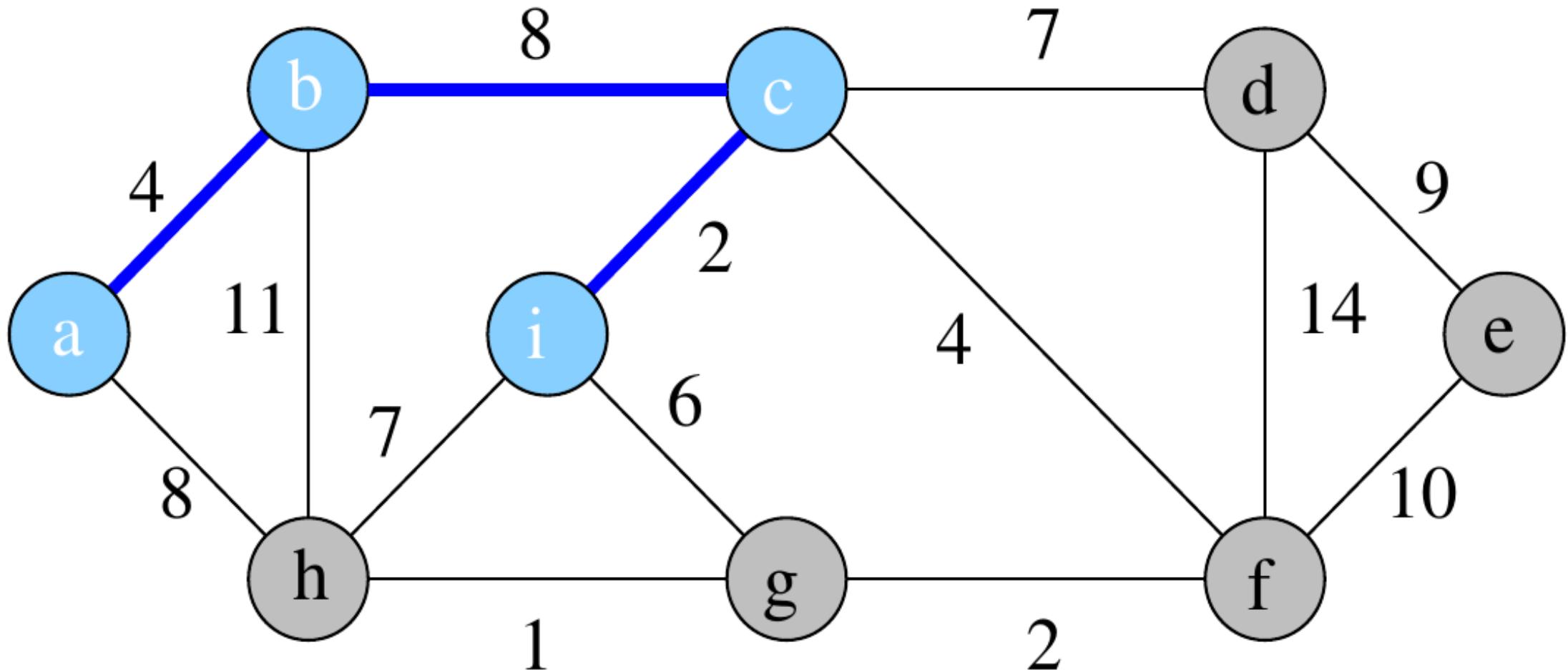
# PRIM'S ALGORITHM

---



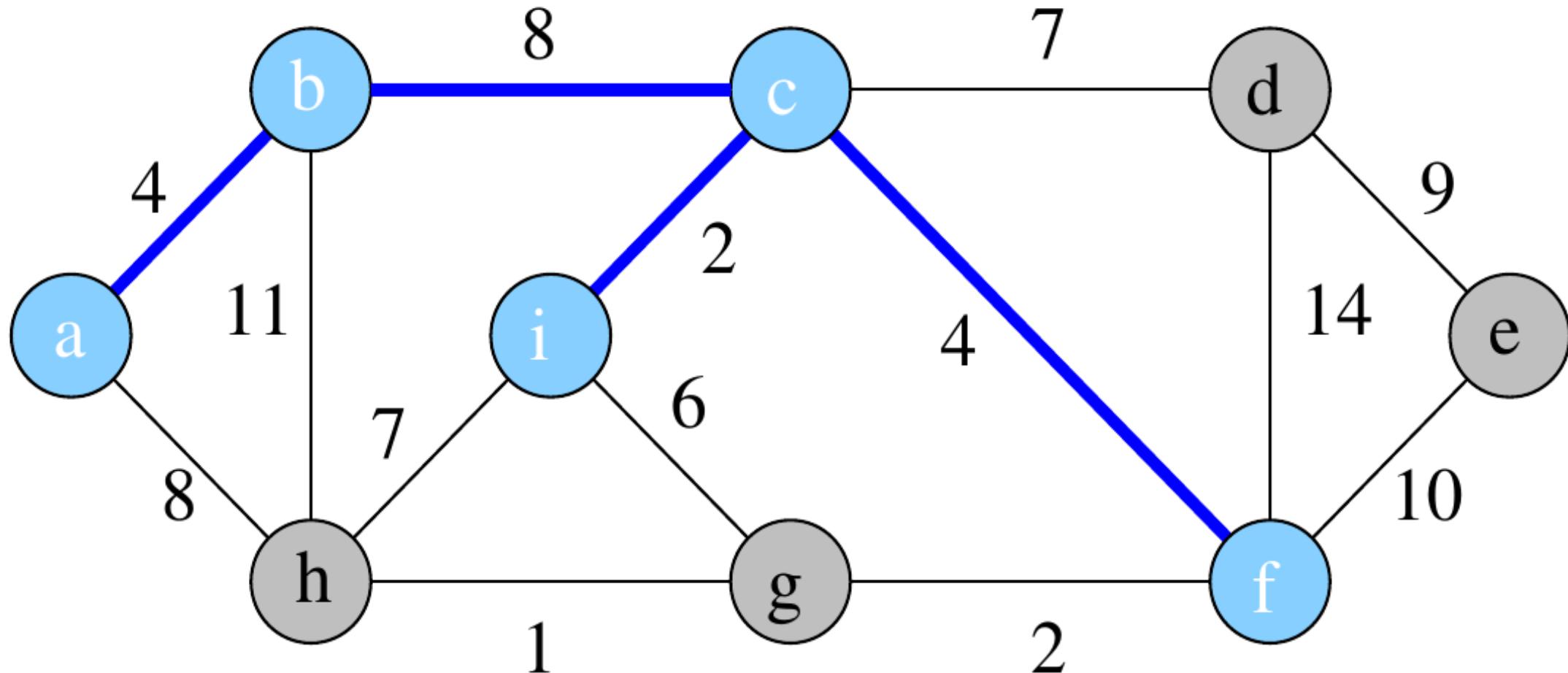
# PRIM'S ALGORITHM

---



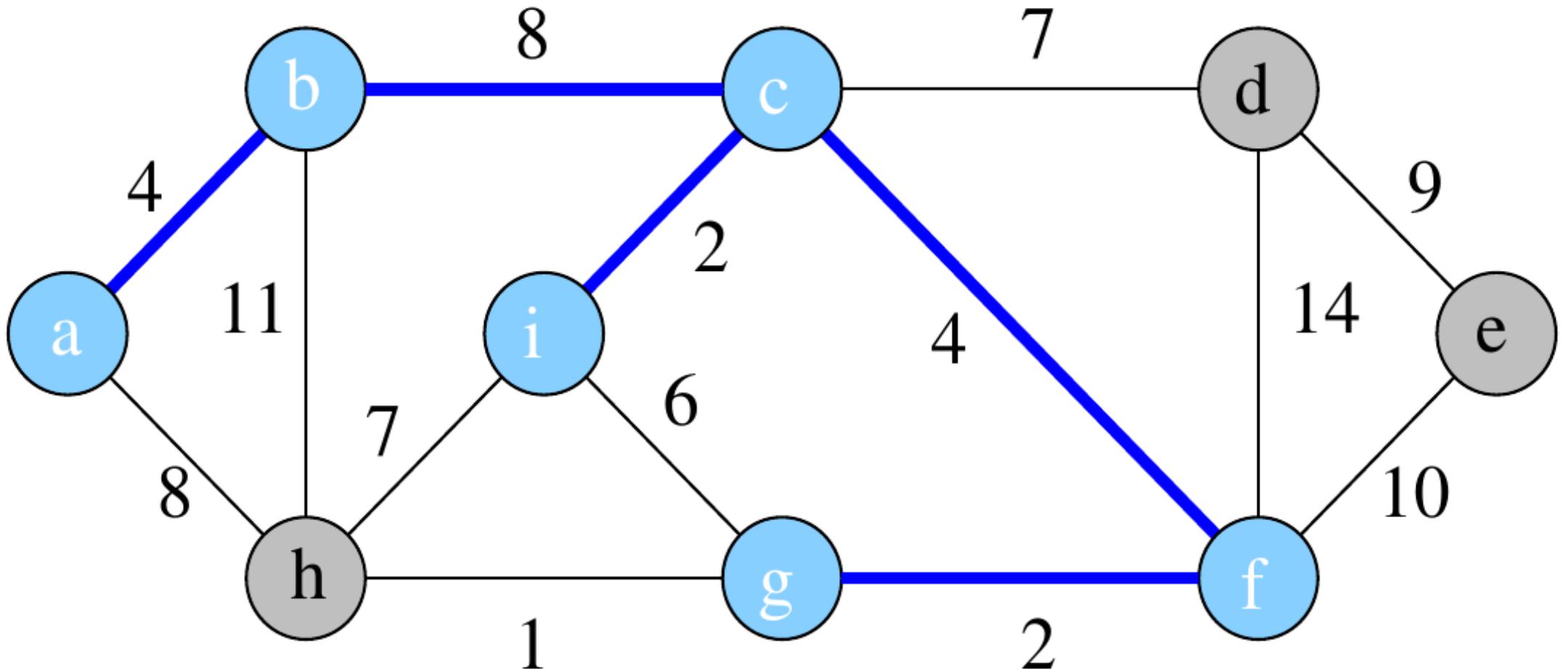
# PRIM'S ALGORITHM

---



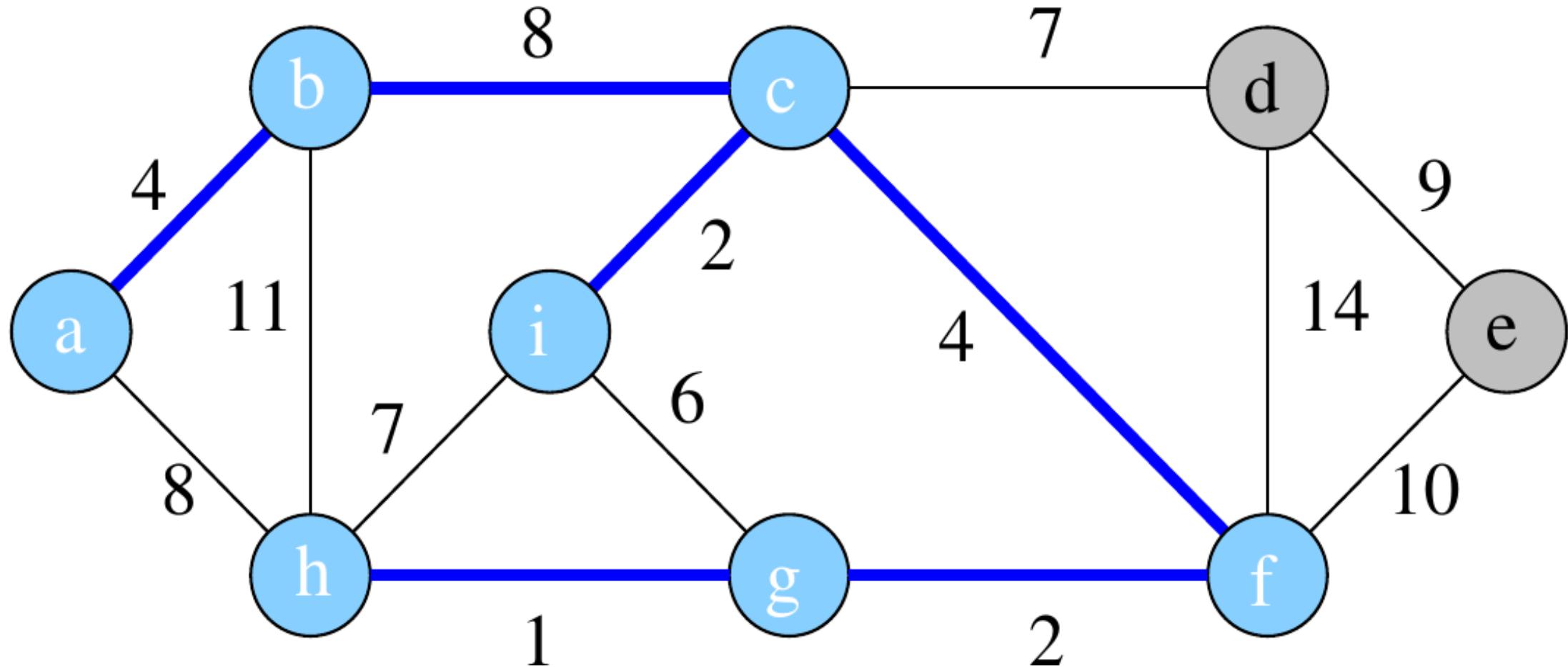
# PRIM'S ALGORITHM

---



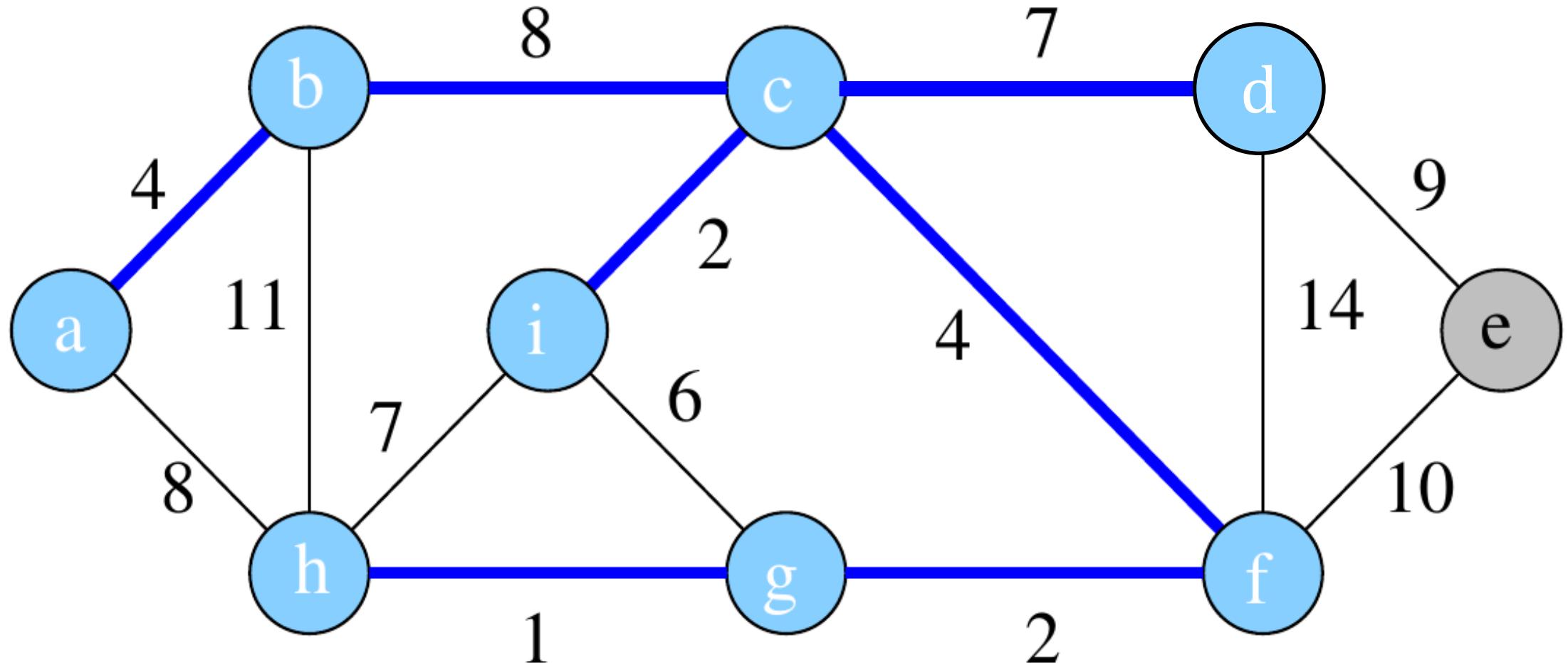
# PRIM'S ALGORITHM

---



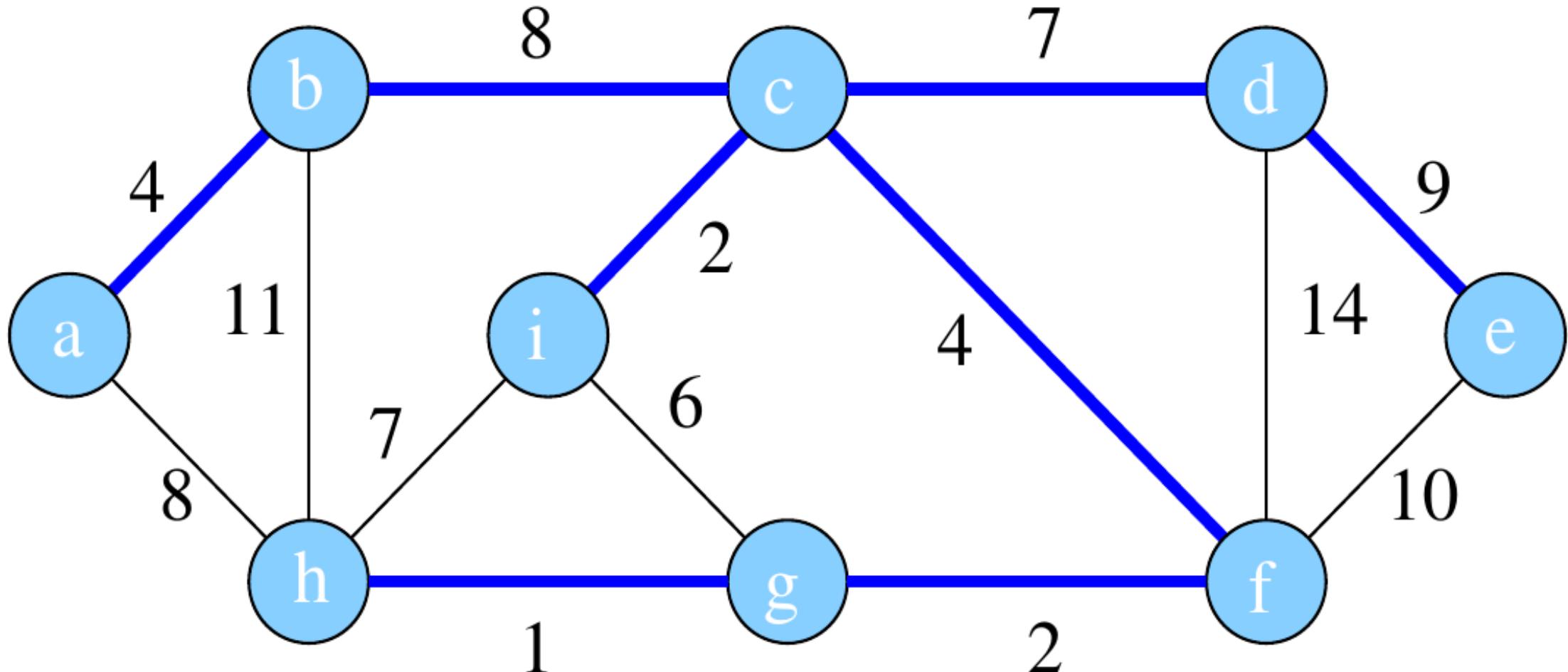
# PRIM'S ALGORITHM

---



# PRIM'S ALGORITHM

---



# PRIM'S ALGORITHM

The algorithm initializes and maintains a **vector** of all vertices **not in the tree A**.

It then runs  $/V/$  iterations to extract the **minimum** vertex and update the distances in the **cut**.

```
185 int Graph::primMST() {
186     int mst_wt = 0; // Initialize result
187     vector<int> parent(adj.size()); // Array to store MST
188     vector<int> key(adj.size()); // Values to pick minimum weight edge in cut
189     vector<bool> visited(adj.size()); // To represent set of vertices included
190
191     // Initialize all keys as INFINITE
192     for (int i = 0; i < adj.size(); i++) {
193         key[i] = INFINITY, visited[i] = false;
194     }
195
196     // Always include first 1st vertex in MST, make sure it is picked first.
197     key[0] = 0;
198     parent[0] = -1; // First node is always root of MST
199
200     // The MST will have V vertices
201     for (int count = 0; count < adj.size(); count++) {
202         // Pick the minimum key vertex not yet included in MST
203         int min = INFINITY, u;
204         for (int v = 0; v < adj.size(); v++) {
205             if (visited[v] == false && key[v] < min) {
206                 min = key[v], u = v;
207             }
208         }
209         // Add the picked vertex to the MST Set
210         visited[u] = true;
211         if (u != 0) {
212             mst_wt += min;
213             cout << u << " - " << parent[u] << ", ";
214         }
215
216         // Update key/parent of the adjacent vertices of the picked vertex.
217         for (auto v : adj[u]) {
218             if (weight[u][v] && visited[v] == false && weight[u][v] < key[v]) {
219                 parent[v] = u, key[v] = weight[u][v];
220             }
221         }
222     }
223     return mst_wt;
224 }
```

graph\_class.cpp

# PRIM'S ALGORITHM

---

We need to execute  $|V|$  iterations of Prim's algorithm.

The algorithm **extracts** the minimum  $|V|$  times and **decreases** distances at most  $|E|$  times. The complexity would be different if implemented with different data structures:

1. **Vector:**  $O(|V|^2 + E) = O(|V|^2)$
2. **Min-heap:**  $O((|V| + |E|)\lg|V|)$

*When does a priority queue give a speedup?*

# REFLECTION

---

1. What is the total weight of the MST found?
2. At the second slide of the example, another node could have been chosen – which? How would the MST look then? What would the total weight of the MST then have been?
3. “Run” prim on the example where you start with the node “f” instead
4. Draw the arrays in the C++ implementation of Prims based on a graph with the nodes c,d,e,f (and the edges between them) from the previous example. Run the code and describe the changes of the variables



# AGENDA

---

- ✓ All-pairs shortest-paths
- ✓ Minimum Spanning Tree (MST)
- ✓ Prim's MST algorithm
- Kruskal's MST algorithm

# KRUSKAL 'S ALGORITHM

---

In Kruskal's algorithm, the subgraph  $H = (V, A)$  is a **forest**.

1. Initially,  $A$  is empty.
2. At each iteration, the algorithm chooses a **light** edge  $(u, v)$  of that links vertices to distinct trees of  $H = (V, A)$ .
3. The procedure adds  $(u, v)$  to the set  $A$  and starts another iteration until  $A$  becomes an MST.

# PSEUDO CODE

---

```
vector<Edge> kruskal( vector<Edge> edges, int numVertices )
{
    DisjSets ds{ numVertices };
    priority_queue pq{ edges };
    vector<Edge> mst;

    while( mst.size( ) != numVertices - 1 )
    {
        Edge e = pq.pop( );           // Edge e = (u, v)
        SetType uset = ds.find( e.getu( ) );
        SetType vset = ds.find( e.getv( ) );

        if( uset != vset )
        {
            // Accept the edge
            mst.push_back( e );
            ds.union( uset, vset );
        }
    }

    return mst;
}
```

The diagram illustrates the time complexity of Kruskal's algorithm. It uses blue arrows and boxes to point from specific code segments to their corresponding complexity calculations:

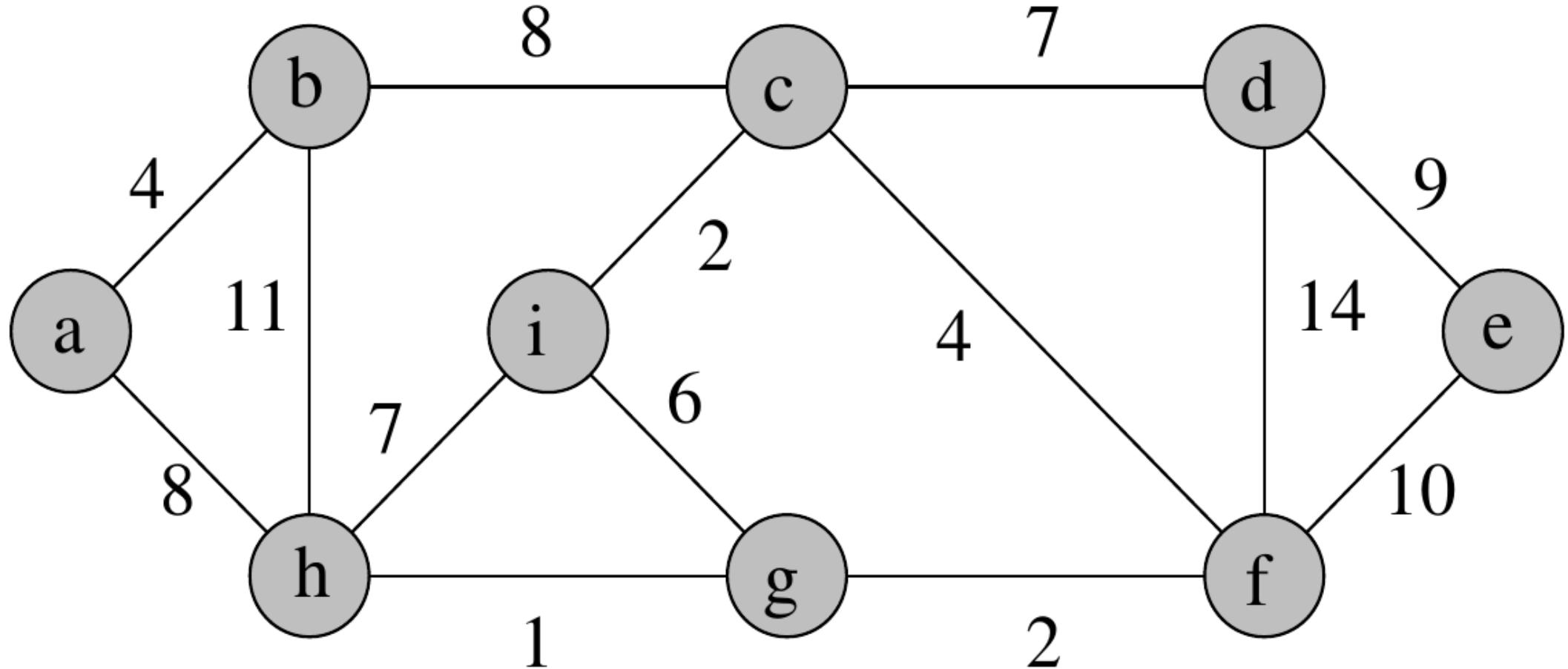
- An arrow points from the declaration of `DisjSets ds{ numVertices };` to a box labeled  $O(|E| \log |E|)$ .
- An arrow points from the `while` loop condition to a box labeled  $O(|V|)$ .
- An arrow points from the assignment `Edge e = pq.pop();` to a box labeled  $O(\log |E|)$ .
- An arrow points from the call to `ds.find` to a box labeled  $O(\log |E|)$ .
- An arrow points from the `if` statement to a box labeled  $O(\log |E|)$ .
- An arrow points from the call to `ds.union` to a box labeled  $O(\log |E|)$ .
- A large bracket groups the complexity of the `if` block and the `ds.union` call, labeled "Max  $O(\log |E|)$  – amortized  $O(1)$  (realistic value)".

**Figure 9.60** Pseudocode for Kruskal's algorithm

Complexity overall:  $O(|E| \log |E|)$

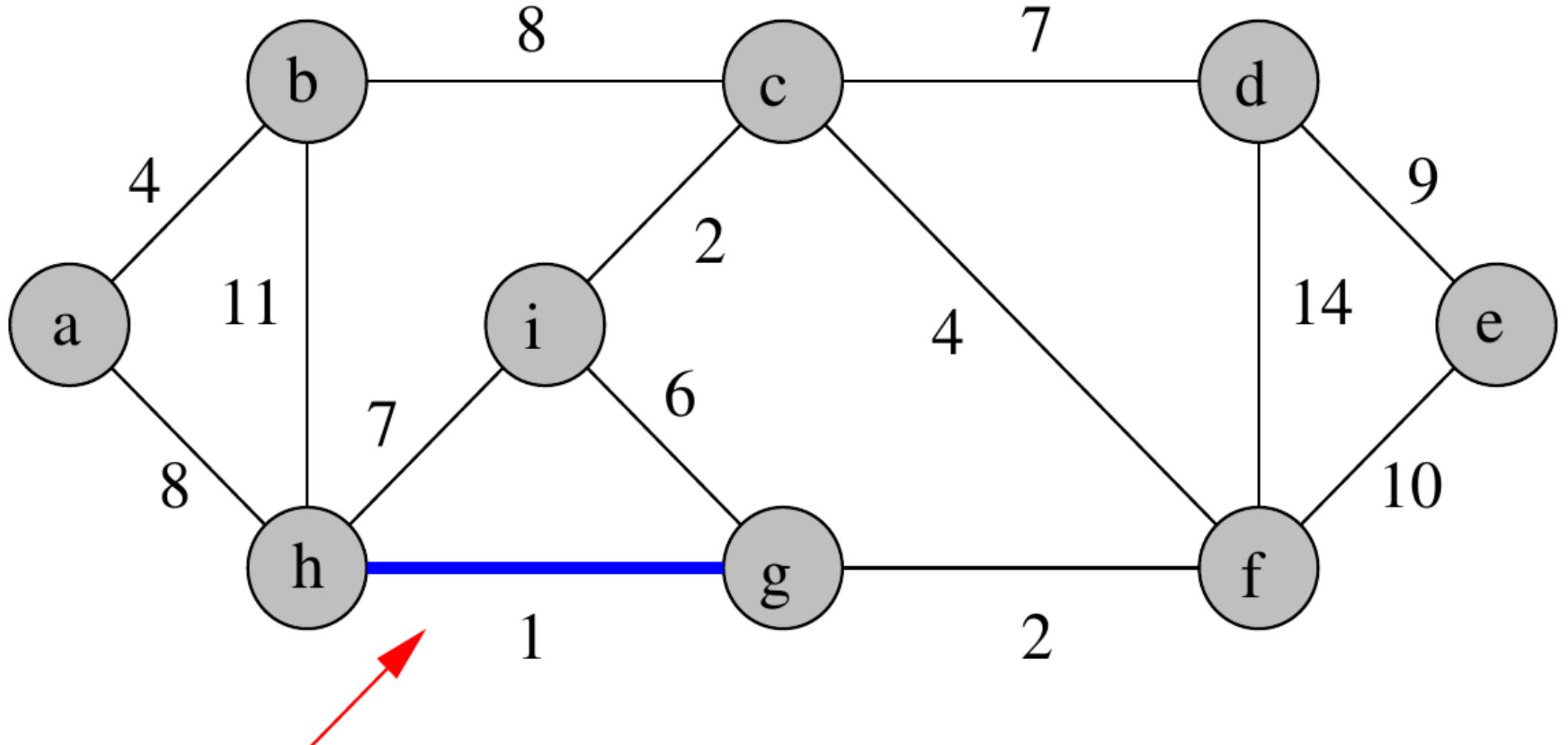
# KRUSKAL 'S ALGORITHM

---



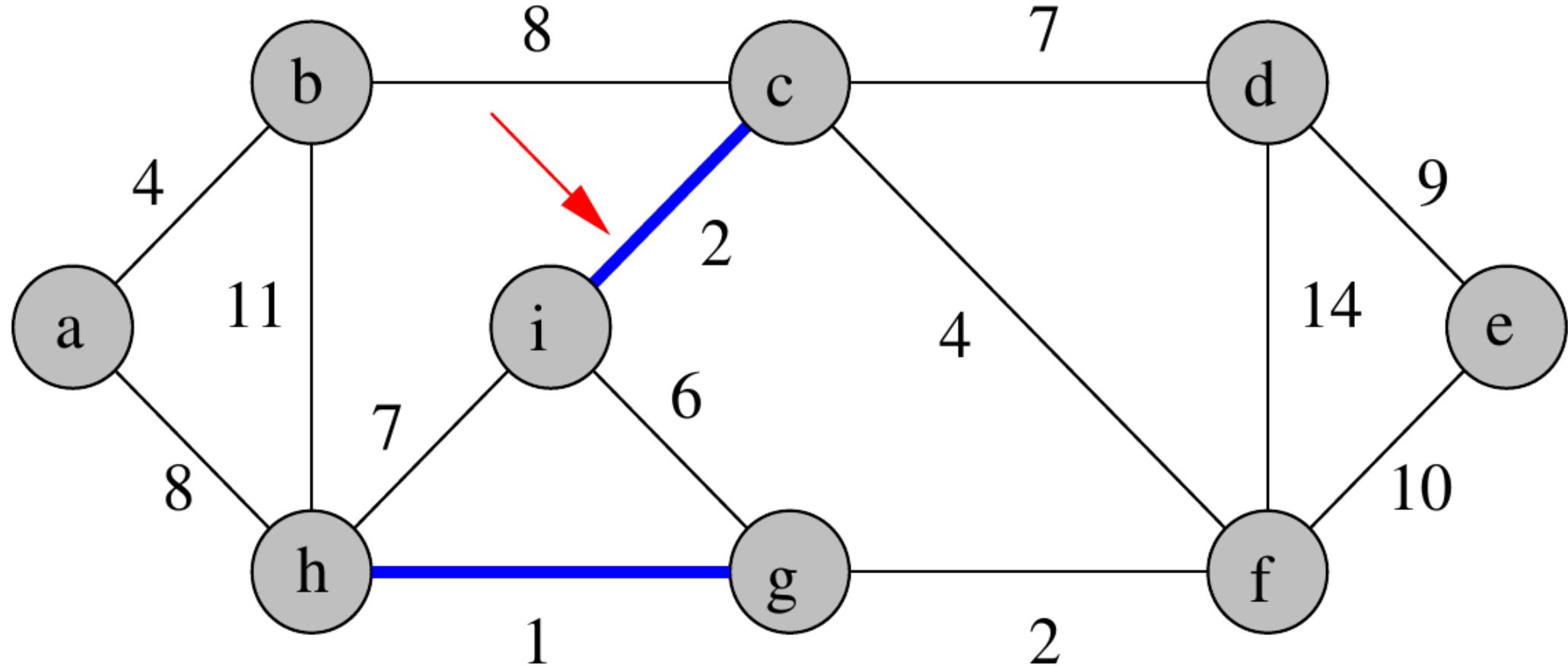
# KRUSKAL 'S ALGORITHM

---



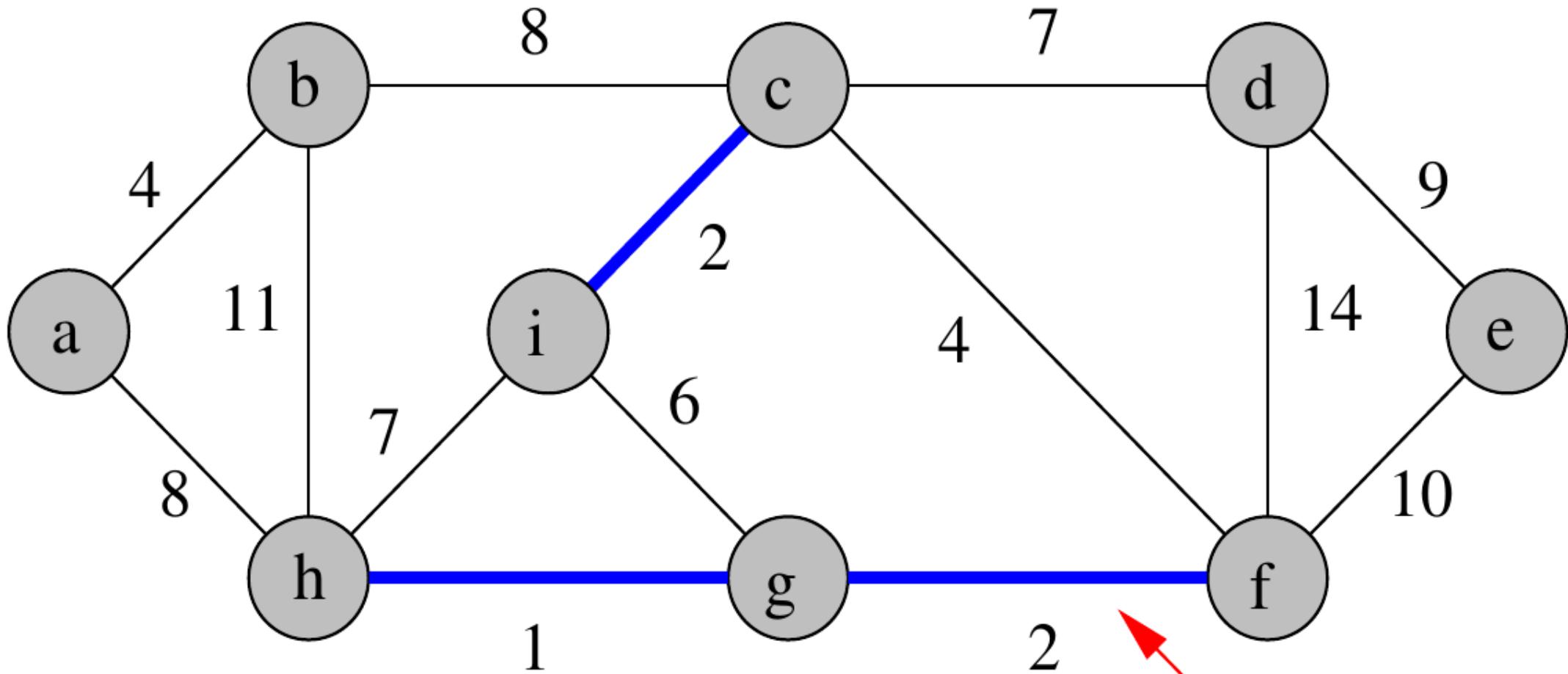
# KRUSKAL 'S ALGORITHM

---



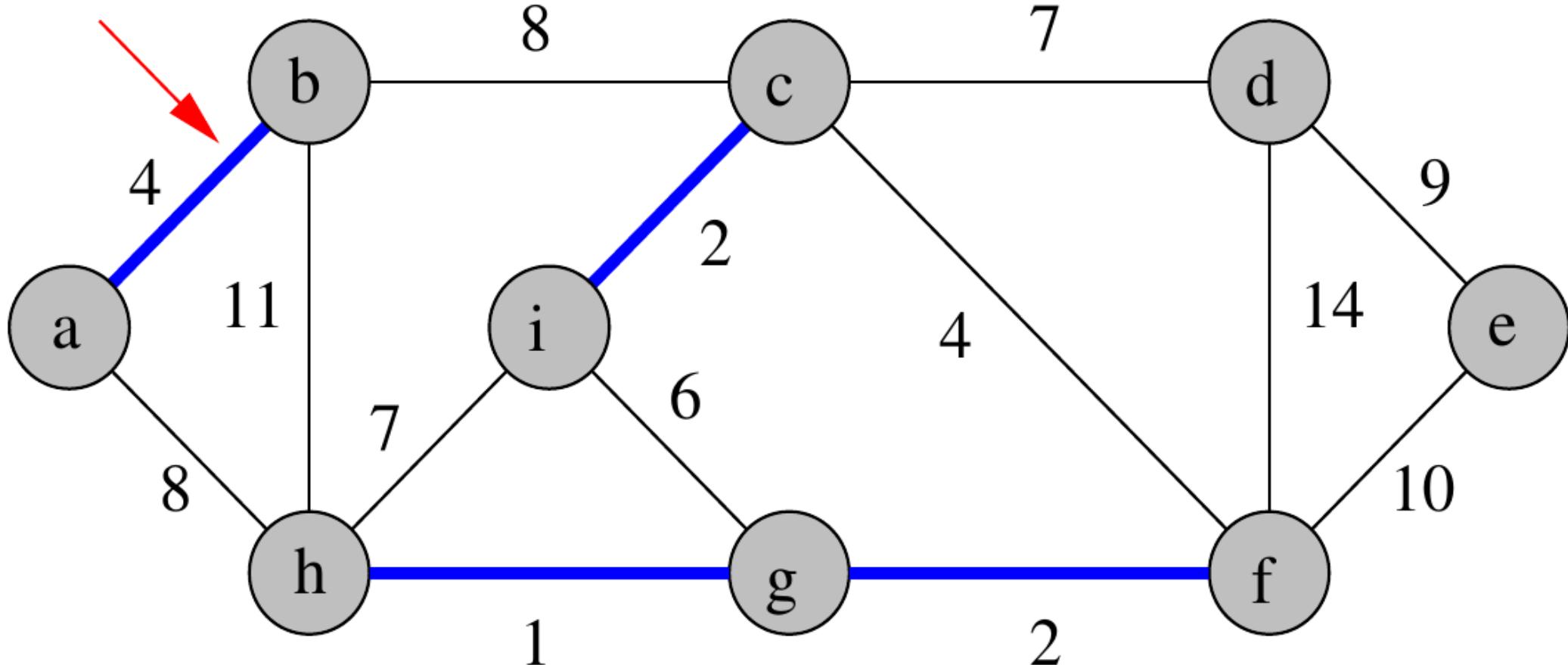
# KRUSKAL 'S ALGORITHM

---



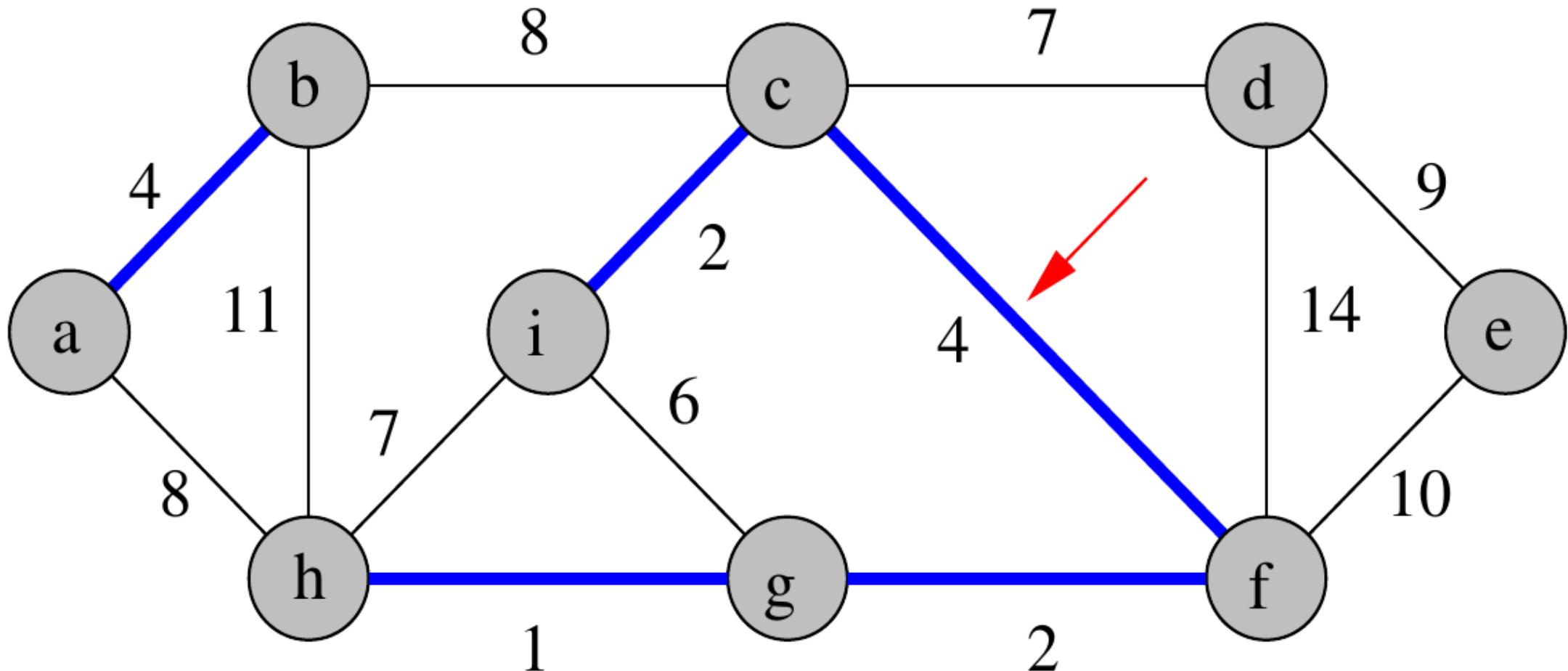
# KRUSKAL 'S ALGORITHM

---

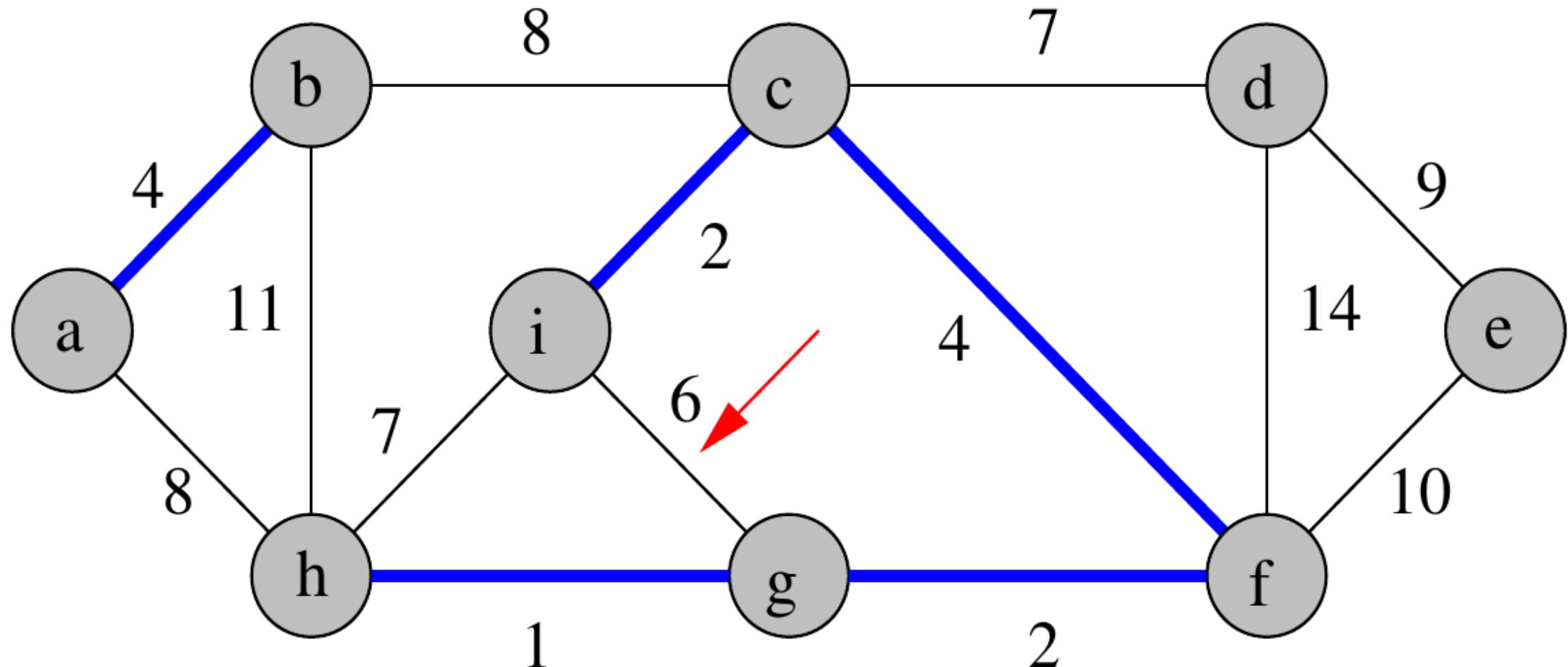


# KRUSKAL 'S ALGORITHM

---

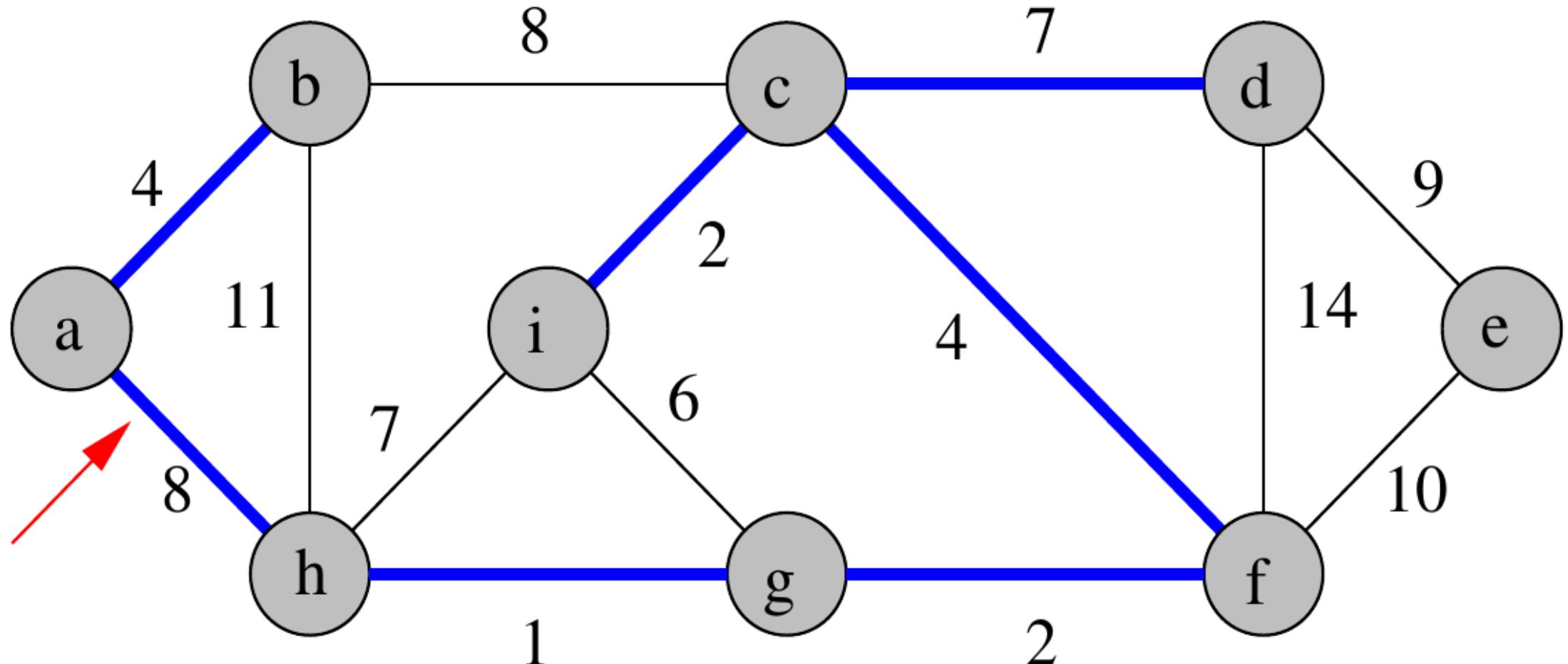


# KRUSKAL 'S ALGORITHM



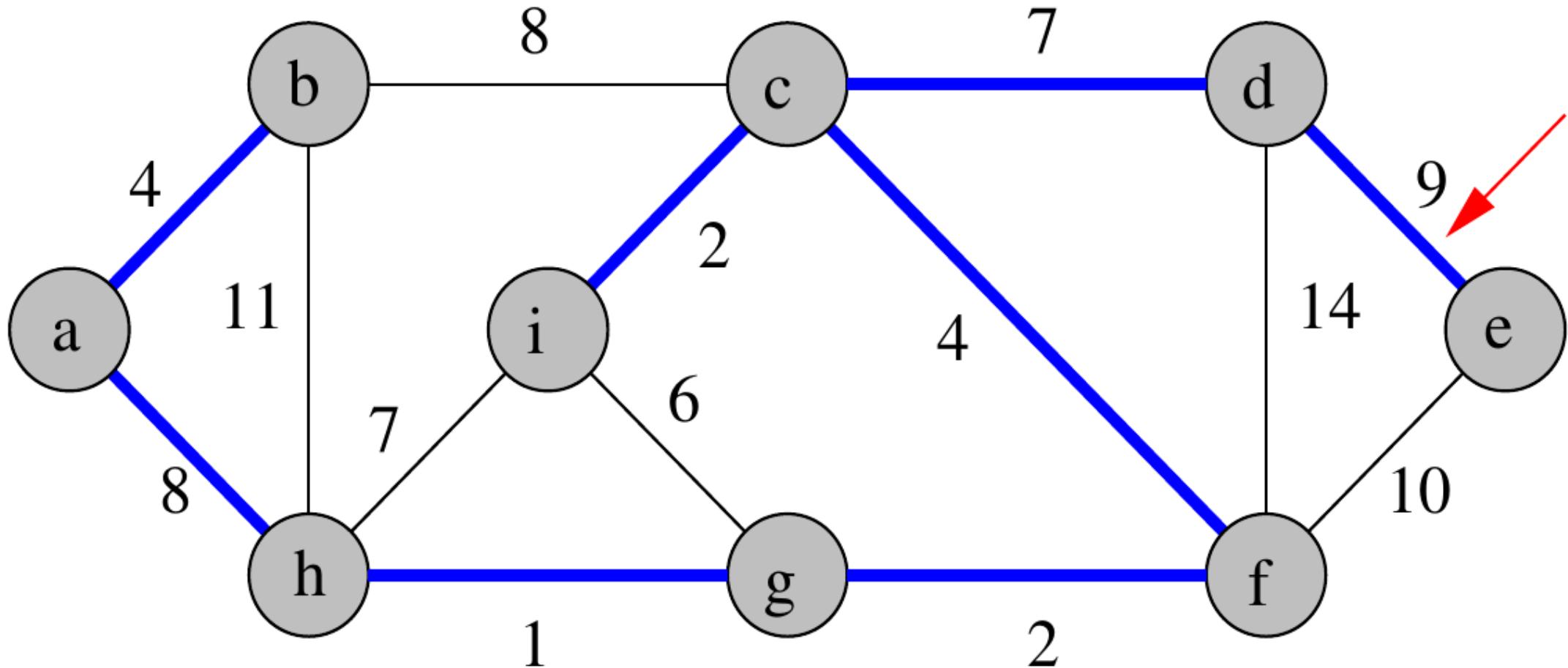
# KRUSKAL 'S ALGORITHM

---



# KRUSKAL 'S ALGORITHM

---



# KRUSKAL'S ALGORITHM

The algorithm initializes and maintains a **disjoint forest** of all vertices **in the tree A**.

It sorts the  $/E/$  edges by cost and joins the sets  $/V/-1$  times until a **single** set remains (the MST).

```
142 int Graph::kruskalMST() {
143     int mst_wt = 0; // Initialize result
144     vector<pair<int, pair<int, int>>> edges;
145
146     for (int i = 0; i < adj.size(); i++) {
147         for (auto j : adj[i]) {
148             edges.push_back({weight[i][j], {i, j}});
149         }
150     }
151
152     // Sort edges in increasing order on basis of cost
153     sort(edges.begin(), edges.end());
154
155     // Create disjoint sets
156     DisjSets ds(adj.size());
157
158     // Iterate through all sorted edges
159     vector<pair<int, pair<int, int>>>::iterator it;
160     for (it = edges.begin(); it != edges.end(); it++) {
161         int u = it->second.first;
162         int v = it->second.second;
163
164         int set_u = ds.find(u);
165         int set_v = ds.find(v);
166
167         // Check if the selected edge is creating
168         // a cycle or not (Cycle is created if u
169         // and v belong to same set)
170         if (set_u != set_v) {
171             // Current edge will be in the MST
172             cout << u << " - " << v << ", ";
173
174             // Update MST weight
175             mst_wt += it->first;
176
177             // Merge two sets
178             ds.unionSets(set_u, set_v);
179         }
180     }
181
182     return mst_wt;
183 }
```

graph\_class.cpp

# REFLECTION

---

1. What is the total weight of the MST found?
2. “Run” Kruskal on the example with the following changes of the weights: B-H:1, H-G:8 and G-F:23
3. Draw the vectors and DisjointSet array in the C++ implementation of Kruskal based on a graph with the nodes c,d,e,f (and the edges between them) from the previous example. Run the code and describe the changes of the variables



# STRING MATCHING

# AGENDA

---

- Strings in C++ and String Matching
  - The naïve algorithm and its complexity
  - The Rabin-Karp algorithm and its complexity
  - The Knuth-Morris-Pratt algorithm and its complexity

# STRINGS IN C++

---

Just like the `vector` class is a **first-class citizen** array in C++, the `string` class is a first-class citizen for an array of bytes. It supports **comparison** operators (`==`, `<`, `>`, etc) and a rich API to copy, append, replace, find, erase, etc.

It can be processed through **iterators**, as an array using the **overloaded** operator `[ ]` in the interval `[0, ..., length () - 1]`, and even as a C string using member function `c_str ()`.

Notice that many of the C standard library functions for strings are **unsafe**, because they do not perform bounds checking.

# THE STRING MATCHING PROBLEM

---

## Definition:

Given a longer string  $T$  of length  $N$ , named the **text**, find all exact occurrences of a shorter substring  $P$  with length  $M < N$ , called the **pattern**.

We will study **three** different approaches to solve the problem, with different **trade-offs**.

Instead of finding all **occurrences**, our algorithms will return a boolean value for when it is possible to find  $P$  in  $T$ , but it should be simple to modify them for the **general** case.

# AGENDA

---

- ✓ Strings in C++ and String Matching
- The naïve algorithm and its complexity
- The Rabin-Karp algorithm and its complexity
- The Knuth-Morris-Pratt algorithm and its complexity

# THE NAÏVE ALGORITHM

---

The **simplest** way to solve the problem is looking for  $P$  in every possible starting position of  $T$ . What is the **complexity**?

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4
5 bool naiveMatch(string pat, string txt) {
6     size_t n = txt.length();
7     size_t m = pat.length();
8
9     for (int i = 0; i <= n - m; i++) { // try all potential starting indices
10        bool found = true;
11        for (int j = 0; j < m && found; j++) { // use boolean flag `found`
12            if (pat[j] != txt[i + j]) { // if mismatch found
13                found = false; // abort this, shift starting index i by +1
14            }
15        }
16        if (found) { // if pat[0 .. m - 1] == txt[i .. i + m - 1]
17            return true;
18        }
19    }
20    return false;
21 }
```

string\_match.h

# THE NAÏVE ALGORITHM

---

The worst-case complexity is  $(N - M + 1)M = O(NM) = O(N^2)$  due to the nested loops. *What is the best-case complexity?*

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4
5 bool naiveMatch(string pat, string txt) {
6     size_t n = txt.length();
7     size_t m = pat.length();
8
9     for (int i = 0; i <= n - m; i++) { // try all potential starting indices
10        bool found = true;
11        for (int j = 0; j < m && found; j++) { // use boolean flag `found`
12            if (pat[j] != txt[i + j]) { // if mismatch found
13                found = false; // abort this, shift starting index i by +1
14            }
15        }
16        if (found) { // if pat[0 .. m - 1] == txt[i .. i + m - 1]
17            return true;
18        }
19    }
20    return false;
21 }
```

string\_match.h

# AGENDA

---

- ✓ Strings in C++ and String Matching
- ✓ The naïve algorithm and its complexity
- The Rabin-Karp algorithm and its complexity
- The Knuth-Morris-Pratt algorithm and its complexity

# RABIN-KARP

---

The idea behind the algorithm is using **hashing** for faster matching of hash values. It computes the hash of the pattern and matches it with the hashes of **all possible prefixes** of the text with length  $M$ .

If a pair of hashes match, this does **not** guarantee an exact match, but a probable one due to **collisions** in the hash function. An exact match is then performed to check for *false positives*.

# RABIN-KARP

---

```
bool rabinKarp(string s[1..n], string pat[1..m])
    hpattern := hash(pattern[1..m]);
    for i from 1 to n-m+1
        hs := hash(s[i..i+m-1])
        if hs = hpattern
            if s[i..i+m-1] = pattern[1..m]
                return true
    return false
```

# RABIN-KARP

---

For a string  $s$  over an **alphabet** of size  $A$ , we will take the hash value to be (for **prime**  $q$ ) for  $i$  in  $[0, M]$ :

$$v = (\sum s_i A^{M-i-1}) \bmod q$$

What is nice about this choice of hash function is that we can quickly **update** the hash by subtracting or adding **multiples of powers** of  $A \bmod q$ .

# RABIN-KARP

Note how  $h$  is **precomputed** as  $A^M \bmod q$  for a certain pattern length. Hashes for pat and  $\text{txt}[0, \dots, M-1]$  are also computed.

The loop scans  $\text{txt}$  looking for hash matches and compares the bytes in the positive case.

If no match is found, the hash is updated.

```
23 #define ALPHABET 256
24 #define Q 101
25
26 bool hashMatch(string pat, string txt) {
27     size_t n = txt.length(), m = pat.length();
28     int i, j, p = 0, t = 0, h = 1;
29
30     // Calculate the hash value of pat and first window of txt
31     for (i = 0; i < m; i++) {
32         h = (h * ALPHABET) % Q; // precompute
33         p = (ALPHABET * p + pat[i]) % Q;
34         t = (ALPHABET * t + txt[i]) % Q;
35     }
36
37     // Slide the pat over txt one by one
38     for (i = 0; i <= n - m; i++) {
39         // Check the hash values
40         if (p == t) {
41             /* Check for characters one by one */
42             for (j = 0; j < m; j++) {
43                 if (txt[i + j] != pat[j])
44                     break;
45             }
46             if (j == m) {
47                 return true;
48             }
49         }
50         // Calculate hash value for next window of txt: Remove
51         // leading digit, add trailing digit
52         if (i < n - m) {
53             t = (ALPHABET * t - txt[i] * h + txt[i + m]) % Q;
54             // We might get negative value of t, converting it to positive
55             if (t < 0)
56                 t = (t + Q);
57         }
58     }
59     return false;
60 }
```

Hash computations

Clever update

string\_match.h



# AGENDA

---

- ✓ Strings in C++ and String Matching
- ✓ The naïve algorithm and its complexity
- ✓ The Rabin-Karp algorithm and its complexity
- The Knuth-Morris-Pratt algorithm and its complexity

# KNUTH-MORRIS-PRATT

---

A **problem** with the previous algorithms is that when they start matching the pattern with the string and find a **mismatch**, they start matching again from the **next starting position**.

The next algorithm does not have this problem, and it never revisits a letter from  $T$  already compared with a letter from  $P$ .

As consequence, the algorithm is **linear in the worst-case** and runs with complexity  $O(N + M) = O(N)$ .

# KNUTH-MORRIS-PRATT

---

Let  $j$  be an index for the pattern and  $i$  for the text in the example:

$T = \text{"I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVEN"}$

$P = \text{"SEVENTY SEVEN"}$

$P = \text{"SEVENTY SEVEN"}$

**Starting** the algorithm with  $j = 0$ , there is no match between  $P[j]$  and  $T[i]$  so it proceeds like the **naïve** algorithm. For  $i = 14, \dots, 24$  there are 11 matches until  $i = 25$ .

The naïve algorithm would start matching again at  $i = 15$ , but KMP **stays** at  $i = 25$  and  $j = 3$ .

# KNUTH-MORRIS-PRATT

**Idea:** Shift pattern more than 1. Rule out shifts that cannot be valid and find first shift *possibly* valid.

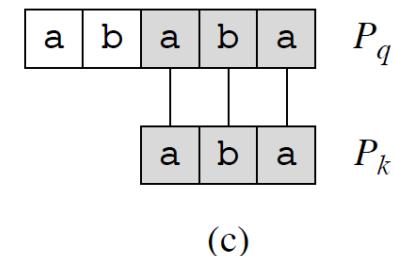
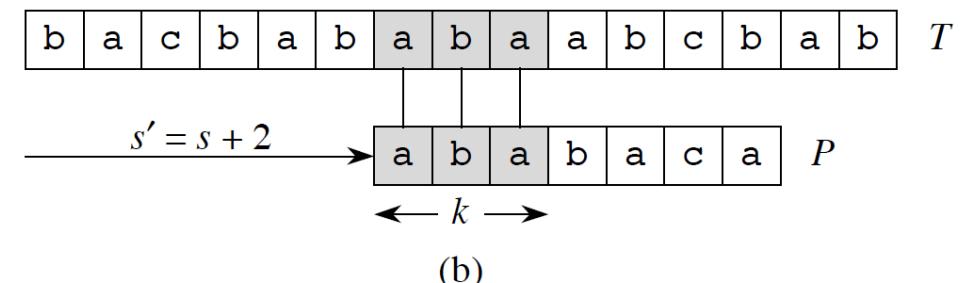
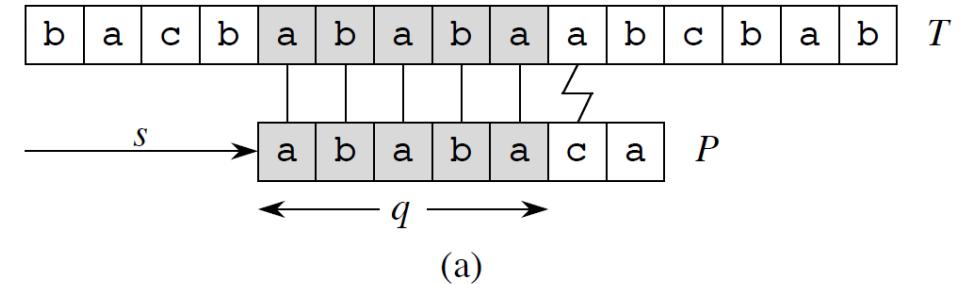
Suppose  $q$  contained no duplicates. E.g.  $q = abcdef$ . Then new shift  $s' = s + q = s + 5$ .

We want to find *first* shift  $s'$  such that shifting by  $s'$  may result in a *possible* valid match.

In (b) if we shift by 2 we find the first shift that may result in a possible match.

This occurs if  $P$  has a prefix that also matches a suffix of  $P$ .

In this case the checking can continue after the matching suffix (index where prev match failed)



# KNUTH-MORRIS-PRATT

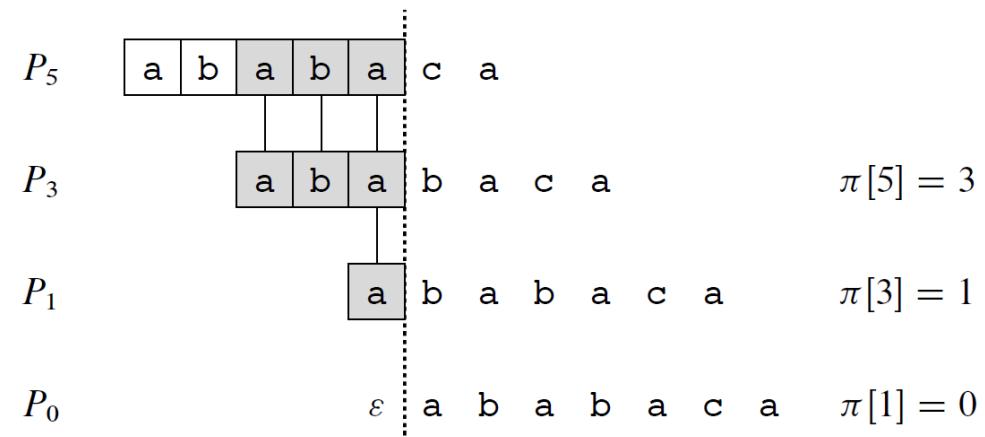
$\pi[q]$  is the length of the longest prefix of  $P$  that is a proper suffix of the prefix  $P_q$

COMPUTE-PREFIX-FUNCTION( $P$ ) ← Assumes  $P[1..m]$

```
1   $m = P.length$ 
2  let  $\pi[1..m]$  be a new array
3   $\pi[1] = 0$ 
4   $k = 0$ 
5  for  $q = 2$  to  $m$ 
6    while  $k > 0$  and  $P[k + 1] \neq P[q]$ 
7       $k = \pi[k]$ 
8      if  $P[k + 1] == P[q]$ 
9         $k = k + 1$ 
10    $\pi[q] = k$ 
11  return  $\pi$ 
```

$i$	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

# KNUTH-MORRIS-PRATT

---

KMP-MATCHER( $T, P$ )

```
1   $n = T.length$ 
2   $m = P.length$ 
3   $\pi = \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4   $q = 0$                                 // number of characters matched
5  for  $i = 1$  to  $n$                   // scan the text from left to right
6    while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
7       $q = \pi[q]$                       // next character does not match
8      if  $P[q + 1] == T[i]$ 
9         $q = q + 1$                     // next character matches
10     if  $q == m$                       // is all of  $P$  matched?
11       print "Pattern occurs with shift"  $i - m$ 
12        $q = \pi[q]$                   // look for the next match
```



AARHUS  
UNIVERSITY

# ALGORITHM DESIGN TECHNIQUES

# INTRODUCTION

---

There are several **heuristics** that can be used to think of an algorithmic solution to a problem. Some examples are the **structure** of the problem, expected **size** of the instances and **similarity** with well-known problems.

We will briefly (re)visit some of these and relate with algorithms we have seen throughout the course.

And we will see in more details **two additional techniques** that we didn't cover so far in the course.

# AGENDA

---

- Brief overview of techniques
  - Greedy Algorithms
  - Divide-and-Conquer Algorithms
  - Backtracking Algorithms
  - **Dynamic Programming Algorithms**
  - **Randomized Algorithms**
- Dynamic Programming Example – Rod Cutting
- Randomized Algorithm Example – Skip Lists

# REUSE

---

“it is important to be a clever thief” –  
“do not reinvent the wheel”

- Many problems have been solved before
  - Google – AND evaluate
  - Libraries (STL, ...)
  - Open source projects
  - ...
- Tweak your requirement into something that has been solved
- Remember the 80-20 rule



Photo by [David Clode](#) on [Unsplash](#)

# DE COMPOSE

---

Usage vs. implementing

- USE the abstractions
- If YOU need something, ask the wish fairy. (S)he will create a function/method to do the job, then you can just call it

Photo by [Anthony Tran](#) on [Unsplash](#)



# GREEDY ALGORITHMS

---

*Greedy* algorithms work as a sequence of **phases**. In each phase, a decision is taken that appears to be good, without **regard for future consequences**.

Generally, this means that some **local optimum** is chosen. This “*take what you can get now*” strategy is the source of the name for this class of algorithms.

When the algorithm terminates, we hope that the local optimum is equal to the **global optimum**.



Photo by [Henley Design Studio](#) on [Unsplash](#)

# GREEDY ALGORITHMS

---

One advantage is that greedy algorithms **degrade gracefully**: if the greedy solution is not optimal, typically an **approximate** solution is obtained instead.

We saw 3 examples in the course of greedy algorithms:

1. **Dijkstra's shortest distance**: greedy **step** is node selection based on shortest tentative distance
2. **Prim's MST**: edge selection with **minimum weight**
3. **Kruskal's MST**: edge selection with **minimum weight**

# GREEDY ALGORITHMS

---

Another example is the *coin-changing problem*: to break down a certain amount of money in the **minimum** number of bills and coins, we iteratively select the **largest** value. For example, breaking \$17.61 with values (10, 5, 1, 0.25, 0.1, 0.01)



Not **general solution**. Breaking \$11 with bills (1, 5, 6, 9) leads to suboptimal solutions.

→ A major pitfall of greedy algorithms is giving the **temptation of optimality** because greedy solutions are so simple.

# REFLECTION

---

Verify that Breaking \$11 with bills (1, 5, 6, 9) leads to suboptimal solutions.

# DIVIDE-AND-CONQUER

---

Another common technique is **divide-and-conquer**, which consists of two parts:

1. *Divide*: break problem in smaller problems and solve recursively (until base case)
2. *Conquer*: combine solutions to subproblems as the solution of the original problem

Generally the subproblems are **disjoint** (can be solved without interference from one to another). We saw several divide-and-conquer sorting algorithms with good performance metrics, such as **MergeSort** and **QuickSort**.

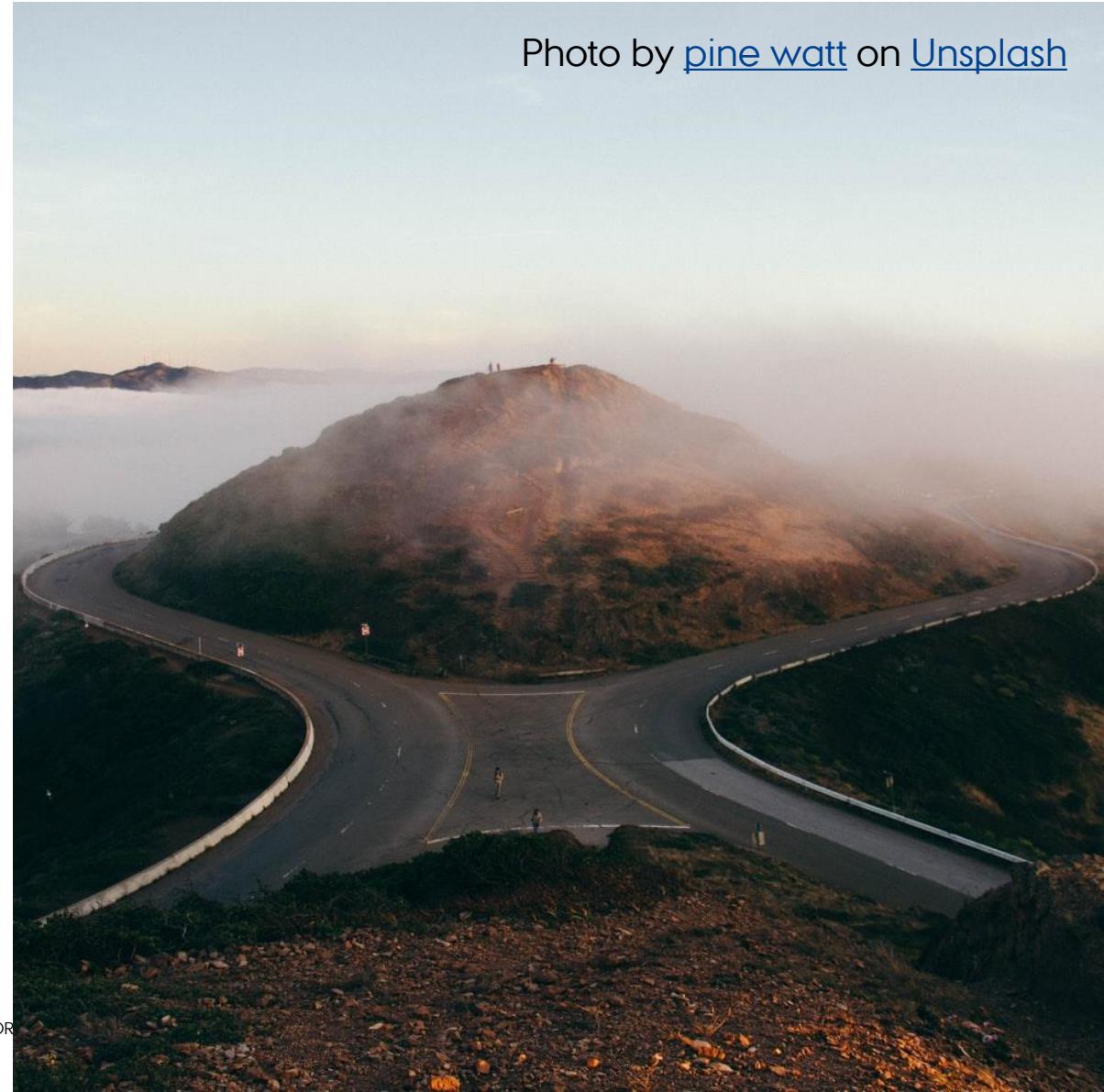


Photo by [pine watt](#) on [Unsplash](#)

# DIVIDE-AND-CONQUER

---

Notice that there are recursive algorithms that do **not** necessarily follow the divide-and-conquer approach, but merely reduce the original problem to a **simpler** case.

Some examples are *binary search*, binary search tree *traversals*, the *find* operation in disjoint sets, etc.

In any case, it is important recall that complexity of recursive algorithms is typically obtained by solving a **recurrence relation**.

# BACKTRACKING

---

In many cases, a **backtracking** algorithm amounts to a clever implementation of **exhaustive search**, with generally unfavorable performance.

Example **Maze challenge** uses backtracking to solve the maze. The search will try all 4 directions.

```
36 bool searchMaze(char maze[][COLS], bool visited[][][COLS], int x, int y)
37 {
38     bool foundExit;
39     if (maze[y][x]==‘E’)
40         return true;
41     visited[y][x]=true;
42     if (validMove(maze, visited, x, y-1)) // up
43         foundExit = searchMaze(maze, visited, x, y-1);
44     if (!foundExit && (validMove(maze, visited, x+1, y))) // right
45         foundExit = searchMaze(maze, visited, x+1, y);
46     if (!foundExit && (validMove(maze, visited, x, y+1))) // down
47         foundExit = searchMaze(maze, visited, x, y+1);
48     if (!foundExit && (validMove(maze, visited, x-1, y))) // left
49         foundExit = searchMaze(maze, visited, x-1, y);
50     return foundExit;
51 }
52
53
54 int main(void)
55 {
56     char maze[ROWS][COLS] = {
57         {'X', 'X', 'X', 'X', 'X'},
58         {'X', ' ', ' ', ' ', 'X'},
59         {'X', ' ', 'X', ' ', 'X'},
60         {'X', ' ', 'X', ' ', 'X'},
61         {'X', 'E', 'X', 'X', 'X'}
62     };
63
64     bool visited[ROWS][COLS];
65     std::fill(*visited, *visited + ROWS*COLS, false);
66
67     int x = 1, y = 1;
68     cout << searchMaze(maze, visited, x, y) << endl;
69 }
```



# REFLECTION

---

Sketch a recursive algorithm for the coin changing problem that exhaustively searches for all possible solutions.

# DYNAMIC PROGRAMMING

---

We saw that any problem that can be mathematically expressed **recursively** can be expressed as a **recursive** algorithm, in many cases yielding a significant performance improvement over a more naïve exhaustive search.

However, naïve recursive solutions can happen to break down the problem in the **same** subproblems and solve them many times in sequence. This particularly happens when solving **optimization problems**.



# DYNAMIC PROGRAMMING

---

There are two requirements for dynamic programming to compute an **optimal** solution **efficiently**:

1. *Overlapping subproblems*: recursive calls tend to solve the same subproblems.
2. *Optimal substructure*: optimal solution to problems are built from optimal solutions of subproblems.

# DYNAMIC PROGRAMMING

---

A dynamic programming solution to a problem follows the strategy below:

1. Characterize the **structure** of an optimal solution.
2. Define the value of an optimal solution **recursively**.
3. Apply **memoization** to store subproblem solutions in table.
4. Compute the value of an optimal solution from optimal subproblem solutions.
5. Build the optimal solution from the table.

***Important:*** *To obtain an efficient solution, the total number of subproblems must be small.*

# DYNAMIC PROGRAMMING

Fibonacci: An example of **using a table to store subproblem solutions.**

Recursive solution (top) is **inefficient and has exponential time complexity.** It calculates the same subproblem solutions multiple times.

Iterative solution (bottom) uses a table (variables) to store subproblem results so they are not recomputed. It is **efficient and has linear time complexity.**

```
1  /**
2   * Compute Fibonacci numbers as described in Chapter 1.
3   */
4  long long fib( int n )
5  {
6      if( n <= 1 )
7          return 1;
8      else
9          return fib( n - 1 ) + fib( n - 2 );
10 }
```

```
1  /**
2   * Compute Fibonacci numbers as described in Chapter 1.
3   */
4  long long fibonacci( int n )
5  {
6      if( n <= 1 )
7          return 1;
8
9      long long last = 1;
10     long long last nextToLast = 1;
11     long long answer = 1;
12
13     for( int i = 2; i <= n; ++i )
14     {
15         answer = last + nextToLast;
16         nextToLast = last;
17         last = answer;
18     }
19     return answer;
20 }
```

# DYNAMIC PROGRAMMING – ROD CUTTING

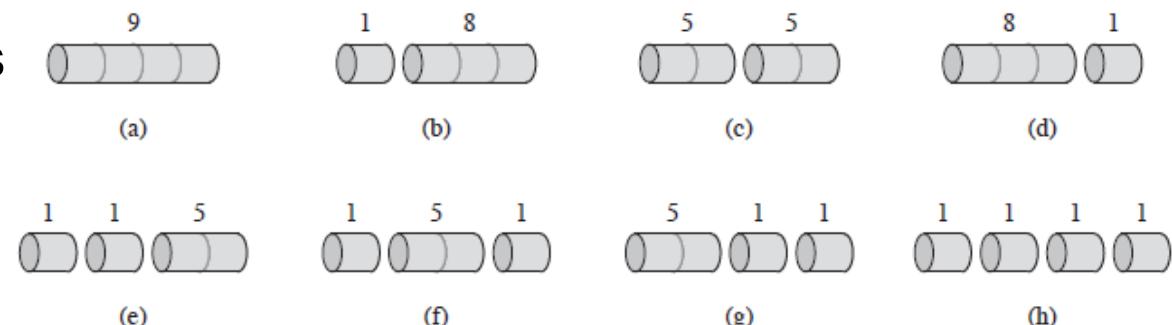
“WeCut Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The company wants to know the best way to cut up the rods.

The **rod-cutting problem**: Given a rod of length  $n$  inches and a table of prices  $p_i$  for  $i$  in  $1 \dots n$ , **determine the maximum revenue  $r_n$**  obtainable by cutting up the rod and selling the pieces.

Cutting a 4-inch rod into  $2 \times 2$ -inch produces revenue  $r_2 = p_2 + p_2 = 5 + 5 = 10$ , which is optimal (fig c).



length $i$	1	2	3	4	5	6	7	8	9	10
price $p_i$	1	5	8	9	10	17	17	20	24	30



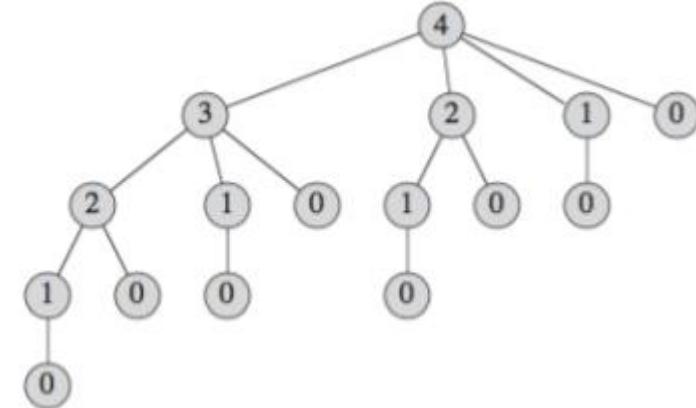
# DYNAMIC PROGRAMMING – ROD CUTTING

A naïve solution is to **generate all configuration** of pieces and then find the highest-priced one. But a rod of length  $n$  can be cut in  $2^{n-1}$  ways since there are  $n-1$  places to make the cut (either cut or don't). So it is an **inefficient exponential solution**.

Instead we can arrange a **recursive structure** for the rod cutting problem. We view a decomposition as consisting of a first piece of length  $i$  cut off the left-hand end, and then a right-hand remainder of length  $n - i$ :

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

(optimal substructure + overlapping subproblems)



Recursion tree with node labels the size  $n$  of corresponding subproblem. Edge from  $s$  to  $t$  corresponds to cutting of a piece of size  $s-t$  and the subproblem of size  $t$ .

# DYNAMIC PROGRAMMING – ROD CUTTING

**Recursive top down implementation not using memoization (naïve) (top)** is direct translation of recursive structure – but is inefficient due to duplicated recursive calls.

**Recursive top down implementation using memoization to store result of recursive calls (bottom).** Future recursive calls can use the precomputed results from earlier calls.

**Runtime:**  $\Theta(n^2)$ . Each subproblem is solved exactly once, and to solve a subproblem of size  $i$ , we run through  $i$  iterations of the for loop. So the total number of iterations of the for loop, over all recursive calls, forms an arithmetic series, which produces  $\Theta(n^2)$  iterations.

```
CUT-ROD( $p, n$ )
1 if  $n == 0$ 
2   return 0
3  $q = -\infty$ 
4 for  $i = 1$  to  $n$ 
5    $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
6 return  $q$ 
```

```
MEMOIZED-CUT-ROD( $p, n$ )
1 let  $r[0..n]$  be a new array
2 for  $i = 0$  to  $n$ 
3    $r[i] = -\infty$ 
4 return MEMOIZED-CUT-ROD-AUX( $p, n, r$ )
```

```
MEMOIZED-CUT-ROD-AUX( $p, n, r$ )
1 if  $r[n] \geq 0$ 
2   return  $r[n]$ 
3 if  $n == 0$ 
4    $q = 0$ 
5 else  $q = -\infty$ 
6 for  $i = 1$  to  $n$ 
7    $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$ 
8  $r[n] = q$ 
9 return  $q$ 
```

# DYNAMIC PROGRAMMING – ROD CUTTING

---

**Iterative bottom up implementation (still using memoization).** Compute solutions for smaller rods first, knowing that they will be used to compute the solutions for larger rods. We still use array r to store/memorize sub results.

Often the bottom up approach is simpler to write, and has less overhead, because you don't have to keep a recursive call stack.

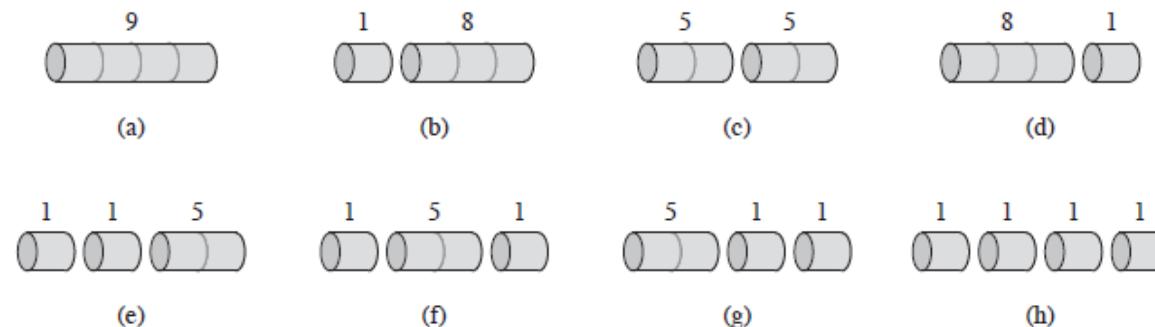
**Runtime:**  $\Theta(n^2)$ , because of the double for loop.

BOTTOM-UP-CUT-ROD( $p, n$ )

```
1 let  $r[0..n]$  be a new array
2  $r[0] = 0$ 
3 for  $j = 1$  to  $n$ 
4    $q = -\infty$ 
5   for  $i = 1$  to  $j$ 
6      $q = \max(q, p[i] + r[j - i])$ 
7    $r[j] = q$ 
8 return  $r[n]$ 
```

# DYNAMIC PROGRAMMING – ROD CUTTING

**Reconstructing a solution.** If we want to actually find the optimal way to split the rod, instead of just finding the maximum profit we can get, we can create another array  $s$ , and let  $s[j] = i$  if we determine that the best thing to do when we have a rod of length  $j$  is to cut off a piece of length  $i$ .



EXTENDED-BOTTOM-UP-CUT-ROD( $p, n$ )

```
1  let  $r[0..n]$  and  $s[1..n]$  be new arrays
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$ 
4     $q = -\infty$ 
5    for  $i = 1$  to  $j$ 
6      if  $q < p[i] + r[j-i]$ 
7         $q = p[i] + r[j-i]$ 
8         $s[j] = i$ 
9     $r[j] = q$ 
10   return  $r$  and  $s$ 
```

$i$	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$		1	2	3	2	2	6	1	2	3	10

# REFLECTION

---

Sketch the memorization version of the Fibonacci algorithm.

Then apply the dynamic programming technique from previous slides.

# RANDOMIZED ALGORITHMS

---

A **randomized algorithm** will (at least once) during its running use a random number to make a decision.

The **worst-case** running time of a randomized algorithm is often the **same** as the same non-randomized algorithm. But **expected running time is often better** since there are no bad inputs.



Example: Quicksort has  $O(n^2)$  worst-case time complexity both deterministic and randomized (random pivot). **In the deterministic algorithm, a particular input can elicit that worst-case behavior. In the randomized algorithm, no input can always give the worst-case behavior.**

# RANDOMIZED ALGORITHMS

---

Deterministic Quicksort has **average complexity** of  $O(n * \log(n))$  but that is unimportant if the input we are faced with are the bad ones (e.g. nearly sorted already, which is not uncommon in practice).

Randomized Quicksort (random selection of pivot) has **worst-case expected complexity** of  $O(n * \log(n))$  which is better than an average-case bound because **it assumes nothing about input**.

Expected time complexity says: “**No matter what input**, the operation will have X time complexity **averaging over the internal decisions (choices in randomized)** of algorithm”.

Average-case complexity says: “**With average/typical input**, the operation will have X time complexity **averaging over the internal decisions (no choices in deterministic)** of algorithm”.

# RANDOMIZED ALGORITHMS – SKIP LISTS

---

**Skip lists.** A probabilistic alternative to balanced search trees (like AVL trees).

AVL trees have worst case time complexity  $O(\log n)$  but somewhat complicated (to code) and expensive (to perform) rotations (insert/delete).

Skip lists have worst case time complexity  $O(n)$  but average/expected  $O(\log n)$  with very high probability. And they are much simpler to code and don't have expensive rotations.

Implementation	Search Time	Insertion Time	Deletion Time
<i>Skip lists</i>	0.051 msec (1.0)	0.065 msec (1.0)	0.059 msec (1.0)
<i>non-recursive AVL trees</i>	0.046 msec (0.91)	0.10 msec (1.55)	0.085 msec (1.46)
<i>recursive 2–3 trees</i>	0.054 msec (1.05)	0.21 msec (3.2)	0.21 msec (3.65)
<i>Self-adjusting trees:</i>			
<i>top-down splaying</i>	0.15 msec (3.0)	0.16 msec (2.5)	0.18 msec (3.1)
<i>bottom-up splaying</i>	0.49 msec (9.6)	0.51 msec (7.8)	0.53 msec (9.0)

# RANDOMIZED ALGORITHMS – SKIP LISTS

Skip list is **sorted linked list** with **fast lanes**.

Searching starts at top lane/list and **looks ahead to next node to see if our search key is larger. If so move to next node in same lane; otherwise move down.**

If each node in layer  $i$  has link to cell  $2^i$  cells ahead, then the structure will have  $\log n$  levels and we only move at most 1 link in each level in a search (otherwise we could have moved in lane above). **So search time is  $O(\log n)$ .**

Problem with this structure is that it is too rigid.

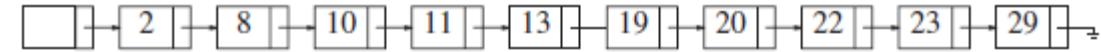


Figure 10.57 Simple linked list

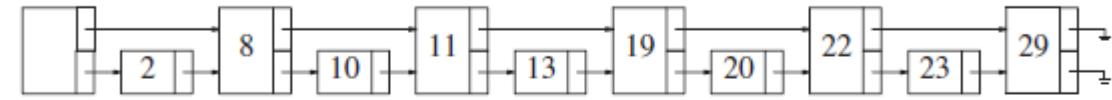


Figure 10.58 Linked list with links to two cells ahead

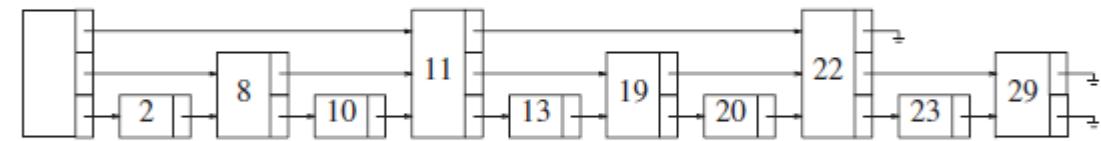


Figure 10.59 Linked list with links to four cells ahead

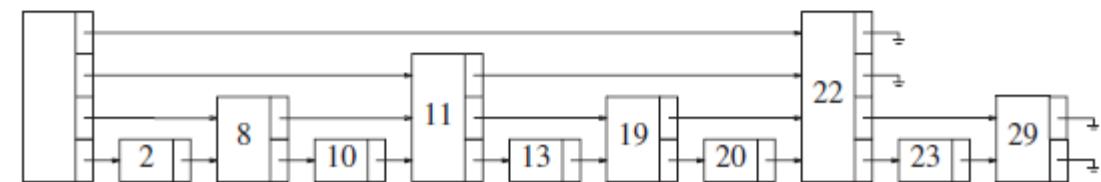


Figure 10.60 Linked list with links to  $2^i$  cells ahead



# RANDOMIZED ALGORITHMS – SKIP LISTS

We relax to less rigid structure by **deciding the level (number of lanes) for an inserted node by a random choice (coin flip)**. This gives us the skip list structure.

When **inserting** and **deleting** nodes, links to/from other nodes must be updated. This requires careful implementation an e.g. an auxiliary data structure to track nodes and layers to update.

The analysis requires a detailed formal proof but the result will be the same as for the *perfect* skip list. Thus the expected time complexity will be  $O(\log n)$ .

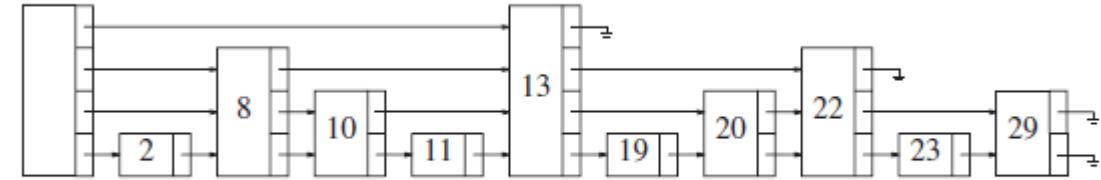


Figure 10.61 A skip list

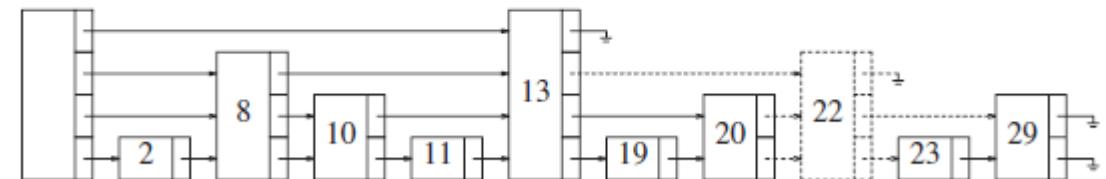
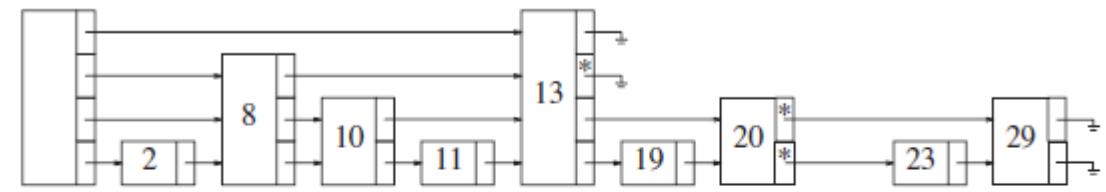


Figure 10.62 Before and after an insertion