



Data Structures for Collections & Multi-File Programs

The Art of Computer Programming

Pierre Senellart



19 October 2026



Outline

- + **Collections**
- + **Sequences**
- + **Stacks and queues**
- + **Trees**
- + **Hash tables**
- + **Multiple files**
- + **Modules**
- + **Coda**

Collections



Where we are

- ★ Lecture 3: the **abstract kinds** of collections and the complexity of their operations; Python lists, sets, dictionaries
- ★ Lecture 4: the same in C++ (`std::vector`, `std::set`, `std::map...`), and how a dynamic array grows
- ★ Lecture 5: classes, so that a data structure can be **written** with its operations and its invariants
- ★ **Today**, two subjects:
 - what is **inside** those collections: the **data structures** that implement them, and what each one makes cheap or expensive
 - how to split a program into **several files**, which is what a data structure of your own asks for
- ★ **Why care about the inside?** To choose the right collection, to explain a program that is too slow, and because you will have to build the structures a library does not provide

The abstract kinds of collections

Ordered sequences the elements are in a definite order, reached by their position;
with random access (any position in $O(1)$) or **without** (only the ends)

Value-based collections no order, no position: the only question is whether an element **belongs** (sets)

Associative arrays each element is a **key** with an associated **value**, reached by its key (maps, dictionaries)

Kind	Python	C++
Ordered sequence, random access	<code>list</code>	<code>std::vector</code> , <code>std::deque</code>
Ordered sequence, ends only	<code>collections.deque</code>	<code>std::list</code> , <code>std::forward_list</code>
Value-based collection	<code>set</code>	<code>std::set</code> , <code>std::unordered_set</code>
Associative array	<code>dict</code>	<code>std::map</code> , <code>std::unordered_map</code>

Each line has **several** implementations, with different trade-offs

Sequences



What we ask of a sequence

A data structure holding $S = (s_0, \dots, s_{n-1})$, with some of these operations:

Random access given i , read or write s_i

Access at the ends read s_0 or s_{n-1}

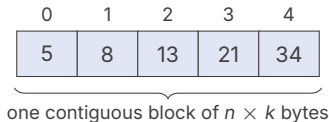
Insertion before s_0 , after s_{n-1} , or between s_i and s_{i+1}

Deletion of s_0 , of s_{n-1} , or of any s_i

Search find an i with $s_i = x$: always $\Theta(n)$ for the structures of this section, unless the sequence is **sorted** (lecture 3: binary search in $\Theta(\log n)$)

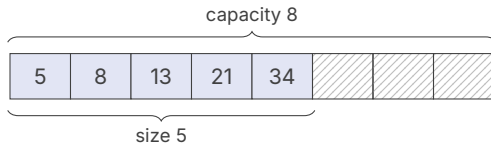
No structure makes all of them cheap: choosing one means deciding which operations the program performs often

Fixed-size array



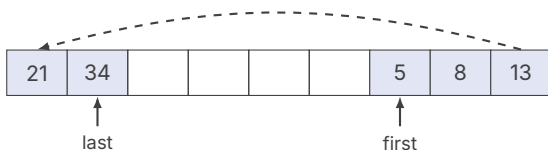
- + A contiguous block, of a size fixed when it is created
- + **Random access in $\Theta(1)$** : with the block starting at address t and elements of k bytes, s_i sits at $t + ki$ (lecture 4)
- + **No insertion, no deletion**: the size cannot change
- + As **compact** as possible: nothing but the elements
- + **In practice**: C arrays and `std::array<T, N>` (lecture 4); in Python, no fixed-size array in the standard library, but `numpy.array` in the `numpy` package, which stores numbers compactly and is the basis of numerical computing in Python

Dynamic array



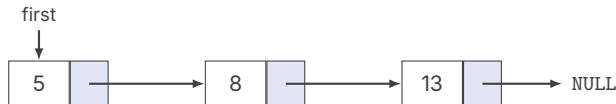
- + Lecture 4: a contiguous block of a fixed **capacity**, holding a variable **size** of elements; full, it is replaced by a block of **twice** the capacity, and the elements copied
- + **Random access in $\Theta(1)$** , insertion and deletion **at the end** in amortized $\Theta(1)$, $\Theta(n)$ elsewhere
- + Slightly less compact than a fixed-size array (the unused capacity), and the occasional reallocation costs $\Theta(n)$
- + **In practice:** Python's `list`, `std::vector<T>`; the default choice for a sequence, in both languages

Double-ended dynamic array



- + **Goal:** insertion and deletion at **both ends** in amortized $\Theta(1)$, keeping random access in $\Theta(1)$
- + **Means:** a block used as a **circular buffer**, with the sequence running from *first* to *last* and wrapping around the end of the block; s_i is at index $(first + i) \bmod capacity$
- + When the block is full, allocate a larger one, as before
- + **In practice:** `std::deque<T>` (in blocks rather than one array); no such structure in Python's standard library, but `collections.deque` gives the same operations

Singly linked list



- + One **node** per element, anywhere on the heap, holding the element and a **pointer to the next node**; the structure itself is a pointer to the first node
- + $\Theta(1)$: access to the first element, insertion at the front, deletion of the front, and insertion or deletion **after a node we already hold**
- + $\Theta(n)$: random access, access to the last element – the only way to reach s_i is to follow i pointers
- + One extra pointer (8 bytes) **per element**, and the elements are **scattered** in memory: see further

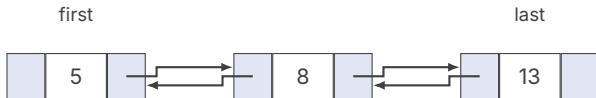
A linked list by hand, in C

C

```
typedef struct Node {  
    int value;  
    struct Node *next;           // the next node, or NULL at the end  
} Node;  
  
void list_push_front(Node **first, int value) {  
    Node *n = malloc(sizeof(Node)); // one allocation per element  
    n->value = value;  
    n->next = *first;  
    *first = n;                     // O(1), whatever the length  
}  
  
int list_get(const Node *first, size_t i) { // O(i): follow the pointers  
    const Node *n = first;  
    for (size_t j = 0; j < i; j++) n = n->next;  
    return n->value;  
}
```

A `struct` that refers to itself (lecture 5), with one `malloc` per node

Doubly linked list



- ★ Each node also points to the **previous** one, and the structure holds a pointer to the **first and the last** node
- ★ $\Theta(1)$: access, insertion and deletion at **both ends**, and deletion of a node we hold (a singly linked list needs its predecessor)
- ★ Still $\Theta(n)$ for random access; two pointers per element
- ★ **In practice**: `std::list<T>` (doubly linked) and `std::forward_list<T>` (singly linked); in Python, `collections.deque` is a doubly linked list of **blocks** of 64 elements

Measured: array or linked list?

	<code>std::vector</code>	<code>std::list</code>	Ratio
Traversing 10^7 integers	0.003 s	0.022 s	$7 \times$ slower
Inserting 2×10^5 at the front	1.05 s	0.003 s	$300 \times$ faster

- + Same operations, opposite verdicts – and the first one is **not** predicted by the complexities, which are $\Theta(n)$ for both traversals
- + The reason is the **memory**: a vector's elements are **side by side**, and the computer reads consecutive memory much faster than memory scattered here and there; a list's nodes are **scattered** over the heap, each with an extra pointer
- + **Consequence**: prefer a dynamic array, unless the program really inserts and deletes at the front or in the middle. Linked lists remain useful as **building blocks** of other structures (queues, hash tables)

Comparison of sequence structures

Structure	Access			Insertion		Deletion	
	s_i	first	last	ends	middle	ends	middle
Fixed-size array	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	—	—	—	—
Dynamic array	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)^*$ (back)	$\Theta(n)$	$\Theta(1)^*$ (back)	$\Theta(n)$
Circular buffer	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(n)$	$\Theta(1)^*$	$\Theta(n)$
Singly linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$ (front)	$\Theta(n)^\dagger$	$\Theta(1)$ (front)	$\Theta(n)^\dagger$
Doubly linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)^\dagger$	$\Theta(1)$	$\Theta(n)^\dagger$

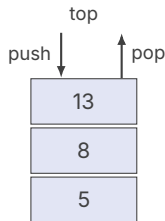
* amortized (lecture 4)

† $\Theta(1)$ once the position is reached, so $\Theta(n)$ in the worst case to find it and $\Theta(1)$ to act
Search for a value is $\Theta(n)$ everywhere, and $\Theta(\log n)$ in a **sorted** array

Stacks and queues



Stack: last in, first out



- + An **abstract** structure: only three operations, all in $\Theta(1)$ – **push** an element on top, **pop** the top one, read the top one
- + Implemented by a dynamic array (push and pop at the end) or a singly linked list (at the front): both give $\Theta(1)$
- + **Last in, first out** (LIFO)
- + **Applications:** **call stack** (lecture 2), undo history of an editor, parentheses to match in an expression

Queue: first in, first out



- + Three operations in $\Theta(1)$: **enqueue** at the back, **dequeue** from the front, read the front
- + Implemented by a doubly linked list or a circular buffer – **not** by a plain dynamic array, where removing the first element costs $\Theta(n)$
- + **First in, first out** (FIFO): the order of a waiting line
- + **Applications**: tasks waiting for a worker, messages waiting to be sent
- + A **double-ended queue** (deque) allows insertion and deletion at both ends, and covers stacks and queues at once

Stacks and queues in practice

Python

```
# stack: a plain list
s = []
s.append(1); s.append(2)
print(s.pop())          # 2
# queue: never list.pop(0)!
from collections import deque
q = deque([1, 2])
q.append(3)              # at the back
print(q.popleft())      # 1, in O(1)
```

C++

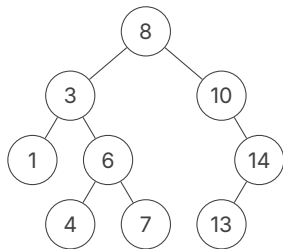
```
#include <stack>
#include <queue>           // and <deque>
std::stack<int> s;
s.push(1); s.push(2);
s.pop();                  // returns nothing
std::queue<int> q;
q.push(1);                // at the back
q.pop();                  // from the front
std::deque<int> d;        // both ends
```

- ✦ `std::stack` and `std::queue` are **adapters**: they restrict another container (a `std::deque` by default) to the LIFO or FIFO operations, which **documents** the intent
- ✦ In Python, `list.pop(0)` is the classic mistake: $\Theta(n)$ per element, where `deque.popleft` is $\Theta(1)$

Trees

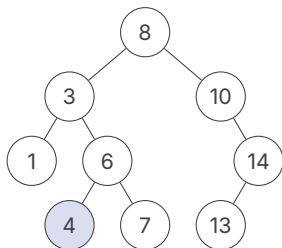


Binary trees



- + A collection of **nodes**, each holding a value and **up to two children** (left and right); one node is the **root**, nodes without children are **leaves**
- + **Depth** of a node: the number of edges from the root (0 for the root). **Height** of the tree: the largest depth, here 3
- + A tree of height h holds up to $2^{h+1} - 1$ nodes, so n nodes fit in height $\log_2(n + 1) - 1$: **a tree can be shallow**
- + **Applications**: parsed expressions, file systems, decision trees, structure of a document

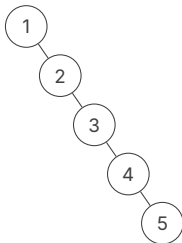
Binary search trees



Looking for 4: $8 \rightarrow 3 \rightarrow 6 \rightarrow 4$,
three comparisons

- + A binary tree where, for every node v : all the values in its **left** subtree are smaller than v , all those in its **right** subtree are larger
- + **Search**: compare with the root, go left or right, repeat – binary search (lecture 3) in a tree, $O(h)$
- + **Insertion**: search for the value, and put a new node where the search failed, $O(h)$
- + **Deletion**: $O(h)$
- + **Ordered traversal**: visit the left subtree, the node, then the right subtree, and the values come out **sorted**, in $\Theta(n)$

Balanced search trees



1, 2, 3, 4, 5 inserted in order:
height 4, a linked list in
disguise

- + Every operation costs $O(h)$, so everything depends on the **height**: $\Theta(\log n)$ if the tree is **balanced**, $\Theta(n)$ in the worst case
- + Not rare: inserting **sorted** values builds exactly that degenerate tree
- + **Solution**: **self-balancing** trees, which restructure themselves on insertion and deletion (local **rotations**) to keep the height $\Theta(\log n)$: **AVL trees**, **red-black trees**, B-trees on disk
- + The algorithms are intricate (see Cormen et al. [2022], chapters 12 and 13); the guarantee is simple: **$\Theta(\log n)$** for search, insertion, deletion

Search trees in practice

- + **C++**: `std::set`, `std::map` (and their `multi` variants) **are** balanced search trees – hence the $O(\log n)$ of lecture 4, and the **sorted** iteration

C++

```
std::map<std::string, int> m = {{"pear", 2}, {"apple", 5}};  
for (const auto &[k, v] : m) std::cout << k << " "; // apple pear  
auto it = m.lower_bound("b"); // first key >= "b"
```

- + **Python**: no search tree in the standard library (`dict` and `set` are hash tables); when a sorted collection is needed, either sort a list, keep it sorted with `bisect`, or use the third-party `sortedcontainers`
- + **Choose a tree over a hash table** when you need the elements **in order**, a range of keys, or the nearest key – operations a hash table cannot do at all

Hash tables



Hashing

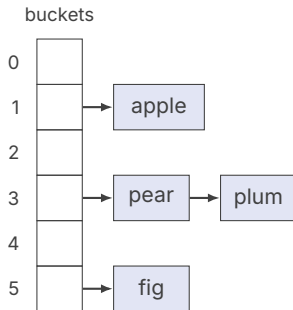
- ✦ A **hash function** h maps a value of any size to an integer of fixed size, **spreading** the values as evenly as possible over its range
- ✦ Equal values must give equal hashes; different values **should** give different hashes, but cannot always (**collisions** are unavoidable: there are more values than hashes)

Python

```
hash(42)           # 42
hash("apple")      # -6521171757592548406 (changes at each run)
hash((1, 2))       # -3550055125485641917
hash([1, 2])       # TypeError: unhashable type: 'list'
```

- ✦ Only **immutable** values are hashable: a mutable one could change after being stored, and would then never be found again – hence the rule of lecture 3 on the elements of a set and the keys of a dict
- ✦ In C++, `std::hash<T>` plays the same role, and a class of your own provides its own specialization

Hash tables



- + An **array of buckets**; the element x goes into the bucket $h(x) \bmod m$, where m is the number of buckets
- + **Insertion, search, deletion**: compute the hash, go to the bucket, look inside it – $\Theta(1)$ if the buckets stay short
- + Colliding elements share a bucket: **chaining** stores them in a linked list (above), **open addressing** puts them in the next free slot of the array
- + **No order**: the elements come out in an arbitrary order, which is the price of the $\Theta(1)$

Load factor, resizing, and the worst case

- The **load factor** is n/m , the average number of elements per bucket: the cost of an operation is $\Theta(1 + n/m)$
- The table **grows**: when the load factor exceeds a threshold (1 in CPython, 1 in `std::unordered_map` by default), allocate twice as many buckets and **rehash** every element – $\Theta(n)$, but rare: **amortized $\Theta(1)$** per insertion, exactly as for the dynamic array (lecture 4)
- **Average-case** $\Theta(1)$ assumes the hash spreads the keys well. In the **worst case**, all n keys land in one bucket and every operation costs $\Theta(n)$

A hash table by hand

Python

```
class HashTable:
    def __init__(self, capacity=8):
        self.buckets = [[] for _ in range(capacity)]    # one list per bucket
        self.size = 0

    def _bucket(self, key):
        return self.buckets[hash(key) % len(self.buckets)]

    def __setitem__(self, key, value):                  # table[key] = value
        bucket = self._bucket(key)
        for i, (k, _) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value); return        # replace
        bucket.append((key, value)); self.size += 1
        if self.size > 2 * len(self.buckets): self._grow()    # rehash
```

Tree or hash table?

	Search	Insert, delete	Ordered traversal
Balanced search tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$, possible
Hash table	$\Theta(1)$ on average*	$\Theta(1)$ on average*	impossible

- * amortized and average-case; $\Theta(n)$ in the worst case
- + **Hash table** when you only ask “is it there?” and “what is its value?”: the usual answer, and the default in Python (`set`, `dict`)
- + **Search tree** when order matters: sorted iteration, ranges, nearest key, or a guaranteed worst case
- + **Duplicates**: sets and maps hold each key once; C++ offers `std::multiset` and `std::multimap`, and in Python one uses a dict of counts (`collections.Counter`) or of lists

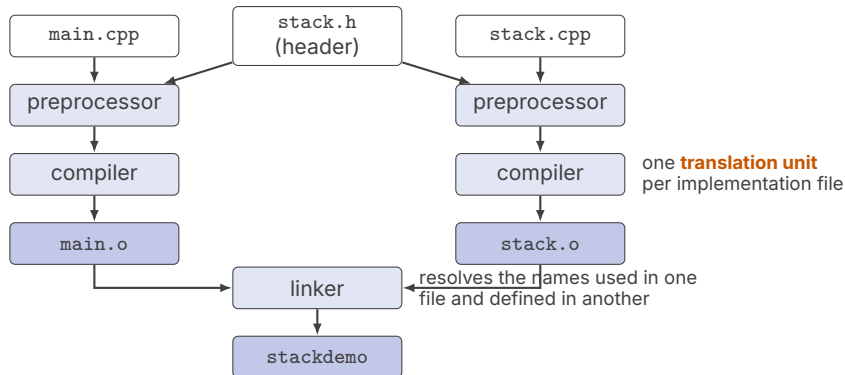
Multiple files



Why several files

- + So far, one file per program. A real program is spread over many
- + **What it buys:**
 - Modularity** one file per component, with a clear **interface**: the stack of the previous section can be used without reading its implementation
 - Reuse** the same file serves several programs
 - Maintenance** a change stays local, and what depends on it is visible
 - Collaboration** several people work without touching the same file
 - Compilation time** only the files that changed are recompiled
- + Two mechanisms to look at: **headers** and **implementation files** in C and C++, **modules** and **packages** in Python

How a C or C++ program is built



`g++ -c main.cpp` stops after the compiler; `g++ -o stackdemo main.o stack.o` runs the linker

Headers and implementation files

- + The rule of lectures 2 and 5 still holds: everything must be **declared before it is used**. With several files, the declarations go into a **header** that every file needing them **includes**
- + **In a header** (.h, or .hpp in C++): function declarations, class definitions, **struct** definitions, constants, declarations of global variables, and the few methods short enough to be read as part of the interface
- + **In an implementation file** (.c, .cpp): the bodies of the functions and methods, the definitions of global variables and of static properties, and everything the rest of the program has no business knowing
- + **Rule of thumb**: a header is what you would show to someone who wants to **use** your code; the implementation file is what you would show to someone who wants to **change** it

The preprocessor

- Before the compiler proper, the **preprocessor** performs **textual** substitutions on the source, obeying the lines that start with #

```
#include <iostream>    // insert a standard header: system directories
#include "stack.h"      // insert one of ours: this directory first
#define MAX_SIZE 100    // replace MAX_SIZE by 100 everywhere below
#ifndef STACK_H         // include the following only if not defined
```

C++

- After preprocessing, the compiler sees **one long file** per implementation file: g++ -E main.cpp shows it (tens of thousands of lines, most of them from <iostream>)
- #define** is a blunt tool: it knows nothing of types or scopes. Prefer a **const** global variable (lecture 4) or a function

Include guards

- A header included **twice** in the same file (directly, and through another header) would define its classes twice: an error
- Every header therefore protects itself with an **include guard**, so that the second inclusion expands to nothing:

```
#ifndef STACK_H
#define STACK_H

class Stack { /* ... */ };

#endif // STACK_H
```

stack.h

```
#pragma once

class Stack { /* ... */ };
```

or, in every compiler

- The guard name must be unique in the program, hence the convention `FILENAME_H`
- `#pragma once` does the same, in one line; it is not in the standard, but every compiler in use supports it

Example: a stack over three files

stack.h

```
#ifndef STACK_H
#define STACK_H
class Stack {
    int *data;           // private
    std::size_t size_, capacity;
public:
    Stack();
    ~Stack();
    void push(int x);
    int pop();
    bool empty() const { return size_ == 0; }
};
#endif
```

stack.cpp

```
#include "stack.h"
#include <stdexcept>

void Stack::push(int x) {
    /* doubling, as in lecture 4 */
}

int Stack::pop() {
    if (empty())
        throw std::out_of_range("empty");
    return data[--size_];
}
```

main.cpp

```
#include "stack.h"
int main() { Stack s; s.push(1); }
```

Building it: make

- + Recompiling everything by hand at each change does not scale: **make** reads a Makefile of **rules**, each naming a target, what it depends on, and how to build it
- + It rebuilds a target only when a file it depends on is **newer**: change `stack.cpp`, and only `stack.o` and the link are redone

Make

```
CXX = g++
CXXFLAGS = -std=c++17 -Wall -Wextra -g

stackdemo: main.o stack.o
    $(CXX) $(CXXFLAGS) -o stackdemo main.o stack.o

main.o: main.cpp stack.h          # the header is a dependency too
    $(CXX) $(CXXFLAGS) -c main.cpp
```

Careful: the command lines must start with a **tabulation**, not spaces. Larger projects generate their Makefiles (CMake, Meson) instead of writing them

Reading the errors of the linker

- + **Compiler error**, in one file: a name is **used but not declared** – a missing `#include`, or a typo

```
main.cpp:5:3: error: 'Stack' was not declared in this scope
```

error

- + **Linker error**, between files: a name is **declared and used but never defined** – the implementation file was forgotten on the command line, or the method was never written, or its signature differs from the declaration

```
undefined reference to `Stack::push(int)'
```

error

- + **The other linker error**: the same name **defined twice** (a function body in a header included by two files)

```
multiple definition of `greet(char const*)'
```

error

- + **One definition, many declarations**: the rule behind all three

Modules



Modules in Python

- ✦ A **module** is simply a .py file; importing it **runs** it once and gives access to the names it defines

Python

```
import counting          # the file counting.py
print(counting.frequencies(w)) # qualified by the module name
from counting import frequencies # bring one name into this namespace
print(frequencies(w))
import numpy as np        # under a shorter name
from counting import *    # everything: avoid, it hides clashes
```

- ✦ No declaration, no header: the **definitions themselves** are the interface, and a leading underscore (lecture 5) marks what is internal
- ✦ The module name is a **namespace**: counting.frequencies and reading.frequencies can coexist
- ✦ Python searches the current directory, then the standard library, then the installed packages (sys.path)

Packages

a package

```
pypackage/  
main.py  
wordcount/  
__init__.py  
reading.py  
counting.py
```

Python

```
from wordcount import words, frequencies  
from wordcount.counting import most_common
```

- A **package** is a directory of modules, imported with a dotted name; `__init__.py` is run when the package is imported
- Its usual job is to define the package's **interface**, re-exporting from the modules what the users need – so that `from wordcount import words` works although `words` lives in `wordcount.reading`

wordcount/__init__.py

```
from .reading import words # . : this package (relative import)  
from .counting import frequencies, most_common
```

Scripts and modules: `__main__`

- A module is **run** at import, so top-level code executes when someone imports the file – rarely what is wanted
- Python sets the variable `__name__` to `"__main__"` in the file given to the interpreter, and to the module name in an imported file, hence the idiom:

Python

```
from wordcount import words, frequencies, most_common

def main():
    counts = frequencies(words(TEXT))
    for word, n in most_common(counts):
        print(f" {word}: {n}")

if __name__ == "__main__":    # run as a script, not imported
    main()
```

- **The same file** is then a program (`python3 main.py`) and a module that others may import (`data/pypackage/`)

Two languages, one idea

	C and C++	Python
Unit of code	implementation file	module (.py file)
Interface	header, included textually	the definitions themselves
Grouping	directory, library	package (directory)
Name qualification	namespace (lecture 4)	module name
Assembling	compiler, then linker	the interpreter, at import
Entry point	main function	<code>if __name__ == "__main__":</code>
Checked when	compiling and linking	running the import

- + The same discipline in both: **small components**, each with an interface others depend on, and an implementation they do not
- + **The practical difference**: C++ catches a missing or mismatched name before the program runs; Python only when the import happens

Coda



Which structure for which need?

Need	Structure	Python / C++
Indexed sequence, growing at the end	dynamic array	list / vector
Insertions at both ends	circular buffer, deque	deque / deque
Many insertions in the middle	linked list	- / list
Last in, first out	stack	list / stack
First in, first out	queue	deque / queue
Membership, lookup by key	hash table	set, dict / unordered_*
The same, in sorted order	balanced search tree	- / set, map

- + **Start from the operations** your program performs often, then read the table – not the other way round
- + And measure: the complexities do not see the memory (the vector against the list), and the constant factors are sometimes the whole story (lecture 3)

Wednesday's tutorial and references

- ★ Wednesday's tutorial: implementing a linked list in C and a stack and a queue on top of it; a binary search tree as a class in Python, with sorted traversal, and watching its height degenerate on sorted input; a small hash table with chaining; then splitting a program into a header, two implementation files and a Makefile, and the same program as a Python package
- ★ The complete programs: `linkedlist.c`, `hashtable.py`, `multifile/`, `pypackage/`
- ★ Next lecture: **common algorithmic techniques** – divide and conquer, dynamic programming, greedy algorithms – which use these structures, and where the dictionary of today becomes the cache of tomorrow
- ★ References: Cormen et al. [2022], chapters 10 (arrays, linked lists, stacks, queues), 11 (hash tables), 12 and 13 (search trees); King [2008], chapters 15 (writing large programs) and 17 (linked structures); `cppreference.com` [2026], Python Software Foundation [2026], and Free Software Foundation [2023]

Licensing

This work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.



Bibliography I

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
Introduction to Algorithms. MIT Press, 4th edition, 2022. ISBN 978-0-262-04630-5.
URL <https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/>.
- cppreference.com. Containers library. <https://en.cppreference.com/w/cpp/container>, 2026.
- Free Software Foundation. GNU make manual. <https://www.gnu.org/software/make/manual/>, 2023.
- K. N. King. *C Programming: A Modern Approach*. W.W. Norton & Company, 2 edition, 2008. ISBN 0393979504. URL <http://knking.com/books/c2/index.html>.
- Python Software Foundation. The python tutorial: Modules.
<https://docs.python.org/3/tutorial/modules.html>, 2026.