

Tutorial 6: Data Structures and Multiple Files

The Art of Computer Programming

21 October 2026

Objective. This tutorial puts into practice the sixth lecture: a linked list in C, with a stack and a queue on top of it; a binary search tree as a Python class, and how its height depends on the order of insertion; a hash table with chaining; then the program of Part A split into a header, two implementation files and a `Makefile`, and the same program as a Python package. Do Parts A and B in the first half of the session and Parts C and D in the second half; Part E is for those who finish early.

Reminders

- Create a folder `tutorial06`. Compile with `$gcc -Wall -Wextra -std=c23 lists.c -o lists`, and use the memory checker of tutorial 4 (`-g -fsanitize=address,undefined`, or Compiler Explorer) in Part A.
- To stop a C program on an error, write `fprintf(stderr, "...\\n");` and then `exit(1);` (`<stdlib.h>`), as in the lecture.
- In Python, `random.shuffle(t)` shuffles the list `t` (module `random`), and `time.perf_counter()` gives a time in seconds (tutorial 2).
- On Windows, `make` is called `mingw32-make`; it was installed with the compiler in tutorial 1. On macOS and Linux, it is `make`.

Part A: Linked lists, stacks, and queues in C

Q1. A linked list. In `lists.c`, write a singly linked list of integers that also keeps a pointer to its last node, as in the lecture:

```
typedef struct Node {
    int value;
    struct Node *next;    // the next node, or NULL at the end
} Node;
typedef struct {
    Node *first, *last;   // both NULL when the list is empty
    size_t size;
} List;
void list_init(List *l);
void list_push_front(List *l, int value);
void list_push_back(List *l, int value);
int list_pop_front(List *l);    // removes and returns the first value
void list_print(const List *l); // prints [3, 1, 4]
void list_free(List *l);
```

- Write the six functions, each in $O(1)$ except `list_print` and `list_free`; `list_pop_front` stops the program on an empty list. Pay attention to `first` and `last` when the list is or becomes empty. Test with the checker: push back 1 and 4, push front 3, print, pop everything, push back 5, print.
- In `list_pop_front`, forget to set `last` to `NULL` when the list becomes empty. What happens in the test above, and what does the checker say? Put it back.
- Why is removing the *last* element not possible in $O(1)$, even with the pointer `last`? Which structure of the lecture would allow it?

Q2. A stack and a queue. In the same file, use the list as a stack (push and pop at the front) and as a queue (push at the back, pop at the front).

- Write `bool balanced(const char *s)`, which tells whether the brackets `()`, `[]`, `{}` of the string `s` are correctly nested: push each opening bracket (a `char` is an integer), and on a closing one, pop and check that it matches. Test on `"(a[b]{c})"`, `"f(t[i])"` (both correct), `"["`, `"("`, `")"`, and `"`. Do not forget to free the stack: what does the checker say if you forget, and for which test? (Leaks are not reported on macOS.)
- Josephus problem*: n people, numbered 1 to n , stand in a circle, and every k -th person leaves the circle until one remains. Put $1, \dots, n$ in a queue; then, as long as more than one person remains, move $k - 1$ people from the front to the back and remove the next one. Print the order of elimination and the survivor: for $n = 7$ and $k = 3, 3, 6, 2, 7, 5, 1$, and the survivor is 4. Who survives for $n = 41$, $k = 3$?
- Give the complexity of each operation of the stack and of the queue. Why must the stack push and pop at the front of the list, and not at the back?

Part B: Binary search trees in Python

Q3. A binary search tree. In `bst.py`, start from the following classes, where `__contains__` is the search of the lecture (value `in` tree):

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None # smaller
        self.right = None # larger

class BST:
    def __init__(self):
        self.root = None # empty
```

Python

```
def __contains__(self, value):
    node = self.root
    while node is not None:
        if value == node.value:
            return True
        if value < node.value:
            node = node.left
        else:
            node = node.right
    return False
```

Python

Add:

- a method `add(value)`, which does nothing if the value is already in the tree, and otherwise puts a new node where the search failed; write it with a `while` loop, like `__contains__` (see Q4);
- `__len__`, keeping the number of values in a property;
- `height()`, -1 for the empty tree, written with a recursive function on the nodes;
- `sorted_values()`, returning the list of the values in increasing order: a recursive function visits the left subtree, appends the value of the node, then visits the right subtree.

Build the tree of the lecture by adding 8, 3, 10, 1, 6, 14, 4, 7, 13 in this order: check its height (3), its sorted values, and a few `in` tests. Add the same values in increasing order: what is the height now?

Q4. Balance.

- Build a tree from `list(range(500))`, then from the same list shuffled. Compare their heights with $\log_2 500 \approx 9$.
- Measure the time to build a tree from n sorted values, and from n shuffled values, for $n = 1000, 2000$, and 4000. How do the times grow when n doubles? Explain with the complexity of `add`.
- Python stops recursive calls beyond a depth of about 1000 (tutorial 2). Why was it wise to write `add` with a loop? Which of your methods would fail on the sorted tree of size 4000?

Part C: Hash tables

Q5. A set with chaining. In `hashset.py`, write a class `HashSet`, a set of values stored in a list of buckets, each bucket a Python list, like the `HashTable` of the lecture but without values:

- `__init__(self, capacity=8)` creates the buckets; a method `_index(self, x)` gives the index of the bucket of `x`, which is `hash(x) % len(self._buckets)`;
- `add(x)`, `__contains__(x)`, and `discard(x)` (which does nothing if `x` is absent) work in the bucket of `x` only;

- `__len__()` and `load_factor()`; when `add` makes the load factor exceed 1, a method `_grow()` replaces the buckets by twice as many and puts every element back, in the bucket given by `_index`;
- `longest_bucket()`, the length of the longest bucket.

Test it with a few values, then add the integers 0 to 9999: how many buckets are there, and how long is the longest one? Why is this case so good? (Print `hash(5)`.) Why does `_grow` have to put every element back, instead of copying the buckets?

Q6. A bad hash function. Write a subclass `BadHashSet` of `HashSet` whose method `_index` uses the length of the string instead of its hash: `len(x) % len(self._buckets)`. Add the 20000 strings `"w" + str(i)` to a `HashSet` and to a `BadHashSet`, timing both: compare the times and the longest buckets, and explain.

Part D: Several files

Q7. A program in three files. In a new folder, split the bracket checker of Q2(a) into: `list.h`, with the two `typedefs`, the declarations of the six functions of Q1, and an include guard; `list.c`, with their bodies (it includes `"list.h"`, `<stdio.h>`, and `<stdlib.h>`); and `main.c`, with `balanced` and a `main` testing it.

(a) Build the program in one step, then in three steps, each file being compiled on its own before the two are linked:

- `$gcc -Wall -Wextra -std=c23 main.c list.c -o brackets`
- `$gcc -Wall -Wextra -std=c23 -c list.c`
- `$gcc -Wall -Wextra -std=c23 -c main.c`
- `$gcc main.o list.o -o brackets`

What do the `.o` files contain?

- (b) Make the following mistakes one at a time, read the message, and say whether it comes from the compiler or from the linker: forget `list.c` in the one-step command; remove `#include "list.h"` from `main.c`; add to `list.h` the function `bool list_empty(const List *l) { return l->size == 0; }`, with its body.
- (c) Write a `Makefile` on the model of the lecture, with `CC = gcc` and `CFLAGS = -std=c23 -Wall -Wextra -g` in place of `CXX` and `CXXFLAGS`, and three rules: `brackets`, `main.o`, and `list.o`, each object file depending on its `.c` file and on `list.h` (the command lines start with a tabulation). Run `make` twice (on Windows, `gcc` names the program `brackets.exe`: if the second run links it again, name the first rule `brackets.exe`). Then modify and save `list.c`, and run `make` again: what is rebuilt? Same question after modifying `list.h`.

Q8. The same program as a Python package. Write a package `brackets/` with three modules: `stack.py`, with a class `Stack` over a Python list (`push`, `pop`, `is_empty`); `checking.py`, with a function `balanced(text)` using it (`from .stack import Stack`); and `__init__.py`, which makes `balanced` part of the interface of the package. Next to the folder `brackets/`, a script `main.py` runs the tests of Q2(a) after `from brackets import balanced`, in a function `main` called under `if __name__ == "__main__":`.

- (a) Add `print("importing", __name__)` at the top level of each of the four files. Predict, then check, what is printed and in which order. Add `import brackets` to `main.py`: is anything printed twice?
- (b) Compare with Q7: what plays the role of the header, of the include guard, and of the linker?

Part E: If you have time

Q9. A range of values. Add to `BST` a method `between(lo, hi)`, returning in increasing order the values v with $lo \leq v \leq hi$, and visiting only the subtrees that may contain some: when should the left subtree of a node be visited, and when the right one? On the tree of Q3, `between(4, 10)` is `[4, 6, 7, 8, 10]`. Why can a hash table not do this efficiently?

Check

By the end of the session you should be comfortable with: a linked list in C and the cases where it is or becomes empty; a stack and a queue, and the complexity of their operations; a binary search tree and why its height matters; a hash table with chaining, its load factor, and what a bad hash function costs; a C program split into a header and implementation files, built by hand and with `make`, telling compiler errors from linker errors; a Python package. If one of them is still unclear, ask us, or finish the corresponding exercises at home.