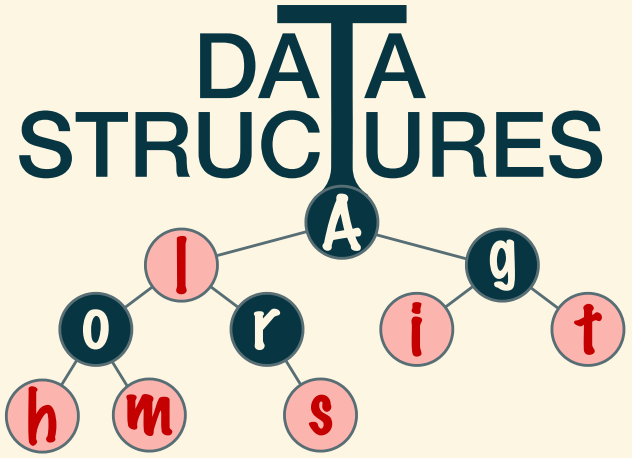
Home



WEEK 04: LINKED LISTS AND HASH TABLES (LAB)

Stefan-Cristian Roata

2026-01-29

Home

INTRO

Home

LEARNING OBJECTIVES

By the end of this lab session, you should be able to:

1. Develop an understanding of how stacks and queues work and implement basic algorithms using these data structures. ↪
2. Manipulate linked lists through operations such as traversal and reversal. ↪
3. Use hash tables to solve simple problems involving frequency counts (e.g., finding the majority element in an array). ↪

Home

OVERVIEW OF LAB ACTIVITIES

Submit your work for the following activities to xSITe and/or Gradescope (as specified on each slide):

1. Multiple-Choice Questions:
   Submit using xSITe Quiz. Total 25 marks.
2. Implement a Queue Using 2 Stacks:
   Submit using Gradescope. Total 25 marks.
3. Reverse a Singly-Linked List:
   Submit using Gradescope. Total 25 marks.
4. Find the Majority Element in an Array Using a Hash Table:
   Submit using Gradescope. Total 25 marks.

Home

MULTIPLE-CHOICE QUESTIONS

Home

EXERCISE 1: MULTIPLE-CHOICE QUESTIONS

On the xSITe course page, navigate to **Assessments** → **Quizzes** → **Week 04 Lab Exercise 1: MCQs**.

Select the best answer for each of the following five questions. You may refer to your notes or the lecture slides.

Home

EXERCISE 1: QUESTION 1

Consider a **singly linked list** with n nodes. You are given a pointer to the head of the list, but **no tail pointer**.

Which of the following operations can be performed in $\Theta(1)$ time?

a. Insert a new node at the end of the list
b. Delete the last node of the list
c. Insert a new node at the beginning of the list
d. Search for a given key in the list

Home

EXERCISE 1: QUESTION 2

In a **singly linked list**, you are given a pointer to a node x that is **not the last node** in the list.

Which operation can be performed in $\Theta(1)$ time?

a. Delete node x
b. Delete the predecessor of x
c. Insert a node before x
d. Find the last node in the list

Home

EXERCISE 1: QUESTION 3

A hash table uses **chaining** with m slots and stores n keys. Assume **simple uniform hashing**, and let the load factor be $\alpha = n/m$.

What is the **average-case time** for an **unsuccessful search**?

a. $\Theta(1)$
b. $\Theta(\alpha \log n)$
c. $\Theta(1 + \alpha)$
d. $\Theta(n)$

Home

EXERCISE 1: QUESTION 4

A hash table uses **chaining** to resolve collisions. Each table slot stores a pointer to the head of a linked list.

Which of the following statements is **always true** for a hash table using chaining?

a. The linked lists are kept in sorted order by key

b. Each linked list contains at most α keys

c. The number of linked lists equals the number of stored keys

d. Each key is stored in exactly one linked list

Home

EXERCISE 1: QUESTION 5

What property should a “good” hash function $h(k)$ have when mapping keys to an m-slot hash table?

a. Always produce prime numbers

b. Distribute keys uniformly across slots

c. Generate values larger than m

d. Map similar keys to similar locations

STACKS AND QUEUES

Home

STACKS

WHAT IS A STACK AND HOW DOES IT WORK?

A stack is a linear data structure that follows the **Last In, First Out (LIFO) principle**.

- The last element added to the stack will be the first one to be removed.
- Think of it like a stack of plates: **you add new plates to the top and remove plates from the top**.

Home

STACK BASIC OPERATIONS

Push

Add an element to the top of the stack.

Pop

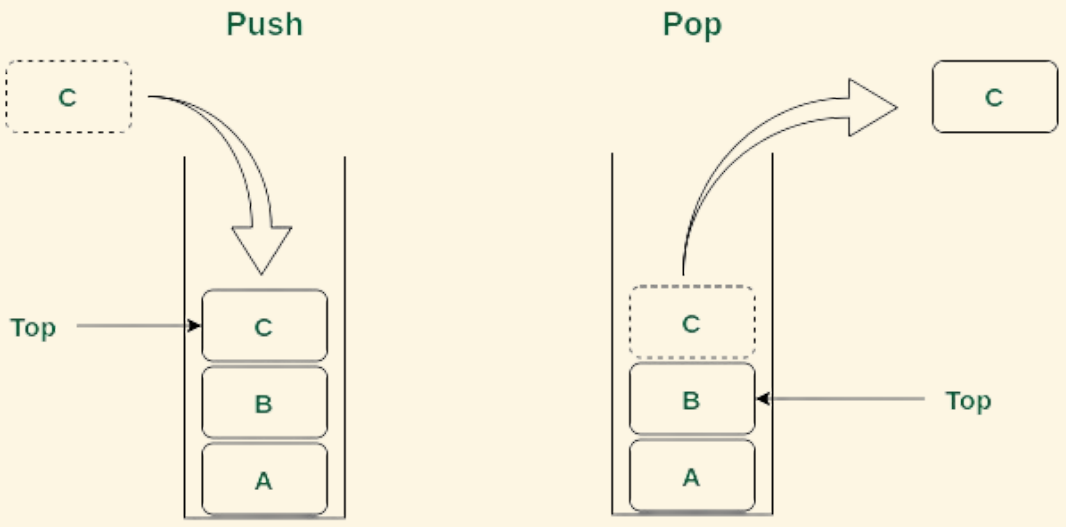
Remove the top element from the stack.

Top/Peek

Retrieve the top element without removing it.

isEmpty

Check if the stack is empty.



Stack Data Structure

Home

C++ IMPLEMENTATION OF STACKS

Stacks can be implemented using native C++ arrays or using libraries like `<stack>` in the Standard Template Library (STL):

```cpp
#include <iostream>
#include <stack>

int main() {
    // Declare stack
    std::stack<int> s;

    // Push elements onto stack
    s.push(10);
    s.push(20);
    s.push(30);

    // Peek at the top
    std::cout << s.top() << '\n';

    // Pop element from stack
    s.pop();

    return 0;
}
```

Home

QUEUES

WHAT IS A QUEUE AND HOW DOES IT WORK?

A queue is a linear data structure that follows the **First In, First Out (FIFO) principle**.

- The first element added to the queue will be the first one to be removed.
- Think of it like a line of people waiting for a service: **the first person in line is the first one to be served**.

Home

QUEUE BASIC OPERATIONS

Enqueue

Add an element to the end of the queue.

Dequeue

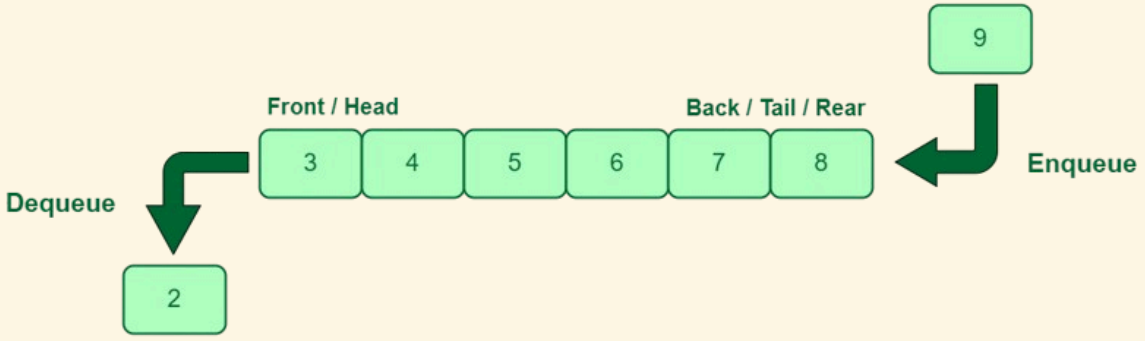
Remove the front element from the queue.

Front/Peek

Retrieve the front element without removing it.

isEmpty

Check if the queue is empty.



Home

C++ IMPLEMENTATION OF QUEUES

Queues can be implemented using native C++ arrays or using libraries like <queue> in the Standard Template Library (STL):

```cpp
#include <iostream>
#include <queue>

int main() {
    // Declare queue
    std::queue<int> q;

    // Enqueue elements into queue
    q.push(10);
    q.push(20);
    q.push(30);

    // Peek at the front
    std::cout << q.front() << '\n';

    // Dequeue element from queue
    q.pop();

    return 0;
}
```

Home

EXERCISE 2: IMPLEMENT A QUEUE USING 2 STACKS

TASK

Your task is to implement a queue using 2 stacks, `s1` and `s2`.

TEMPLATE

You can find the template files on xSITe (.zip archive).

Complete the partial `queue_using_2_stacks.cpp` file. The `make` command will compile the project.

SUBMISSION

Implement the `.enqueue()` and `.dequeue()` methods of the `Queue` struct inside `queue_using_2_stacks.cpp`. **Do NOT use** `std::queue` in your implementation.

To receive credit for your work, upload your completed `queue_using_2_stacks.cpp` file to the Gradescope assignment titled **Week 04 Lab Exercise 2: Implement a Queue Using 2 Stacks**.

Home

EXERCISE 3: REVERSE A LINKED LIST

TASK

Given the head of a singly-linked list, your task is to implement a function that reverses this linked list iteratively.

TEMPLATE

You can find the template files on xSITe (.zip archive).

Complete the partial `reverse_linked_list.cpp` file. The `make` command will compile the project.

SUBMISSION

Implement the `reverseList()` function inside `reverse_linked_list.cpp`.

Do NOT use any built-in functions for reversing the list (e.g., `std::reverse`).

To receive credit for your work, upload your completed `reverse_linked_list.cpp` file to the Gradescope assignment **Week 04 Lab Exercise 3: Reverse a Linked List**.

HASH TABLES

Home

C++ IMPLEMENTATION OF HASH TABLES (UNORDERED MAP)

One of the ways to create a hash table in C++ is to use `std::unordered_map`. Sample code is presented on this and the following slide.

Here is an example of declaring, inserting, and accessing elements:

```cpp
#include <iostream>
#include <unordered_map>

int main() {
    // Declare unordered_map
    std::unordered_map<std::string, int> umap;

    // Insert elements into unordered_map
    umap["apple"] = 1;
    umap["banana"] = 2;
    umap["cherry"] = 3;

    // Access elements
    std::cout << "apple: " << umap["apple"] << '\n';
    std::cout << "banana: " << umap["banana"] << '\n';
```

Home

C++ IMPLEMENTATION OF HASH TABLES (CONTINUED)

Additional operations include checking for existence, removing elements, and iterating:

```
    // Check if an element exists
    if (umap.find("cherry") != umap.end()) {
        std::cout << "cherry is in the map\n";
    }

    // Remove an element
    umap.erase("banana");

    // Iterate through the unordered_map
    for (const auto& pair : umap) {
        std::cout << pair.first << ": " << pair.second << '\n';
    }
}
```

Home

EXERCISE 4: FIND THE MAJORITY ELEMENT IN AN ARRAY

TASK

Given an array of positive integers, your task is to implement a function that finds the majority element in this array using a hash table. **An element is considered a majority element if it appears strictly more than `n / 2` times in an array containing `n` elements.**

If no majority element exists, then the function should return -1.

EXAMPLES

1. `[1, 2, 2]` — element 2 is the majority element, since it appears $2 > 3/2$ times in the array.
2. `[1, 2, 3]` — no majority element exists, -1 is returned.
3. `[5, 5, 5, 8, 8, 8]` — no majority element exists, -1 is returned. Both 5 and 8 appear 3 times, but either of them should appear at least $6/2 + 1 = 4$ times to be considered the majority element.

Home

EXERCISE 4: FIND THE MAJORITY ELEMENT IN AN ARRAY

TEMPLATE

You can find the template files on xSITe (.zip archive).

Complete the partial `majority_element.cpp` file. The `make` command will compile the project.

SUBMISSION

Implement the `findMajority()` function inside `majority_element.cpp`.

Do NOT use any built-in functions for finding the majority element (e.g., `std::count`). You may however use `std::unordered_map` to instantiate your hash table.

To receive credit for your work, upload your completed `majority_element.cpp` file to the Gradescope assignment titled **Week 04 Lab Exercise 4: Find the Majority Element**.

CONCLUSION

Home

SUMMARY OF KEY LEARNING OUTCOMES

1. Reviewed key properties of hash tables with chaining, including average-case search time under simple uniform hashing. ↪
2. Understood how stacks and queues work, and implemented a queue using two stacks:
    - Stacks follow the Last In, First Out (LIFO) principle. ↪
    - Queues follow the First In, First Out (FIFO) principle. ↪
3. Manipulated singly-linked lists by implementing an iterative reversal algorithm using three pointers. ↪
4. Used `std::unordered_map` to build a hash table and solve a frequency-counting problem (finding the majority element). ↪
5. Applied hash table lookups and insertions to count element occurrences in a single pass through an array. ↪

OUTLOOK

Next week's topics:

This lab introduced fundamental data structures—stacks, queues, linked lists, and hash tables—that appear throughout computer science. Next week continues with another essential structure:

- **Binary Search Trees (BSTs):** A tree-based data structure that supports efficient search, insertion, and deletion in $O(h)$ time, where h is the tree height.
- **Tree traversals:** In-order, pre-order, and post-order traversal algorithms.