C++ Code Documentation

Folder Search Program

Overview

This C++ program simulates searching for a specific file inside a virtual folder system. Imagine you have a main folder (root) that can contain files and other folders. These subfolders can also contain more files and even more nested folders, going as deep as needed - just like folders on your computer.

The goal of the program is to find out if a file with a certain name exists anywhere in this entire folder structure.

How it works

Folder structure

The program uses a Folder structure to represent each folder. Each folder has:

- A list of files (simple file names as strings).
- A list of subfolders (which are themselves Folder objects).

This allows creating deeply nested folder hierarchies.

Searching for a file

The function searchFile() is used to look for the target file.

Here's what it does:

1. Check the current folder's files: It goes through all the files directly inside the current folder to see if any of them match the target file name.
2. Search in subfolders: If it doesn't find the file, it looks inside each subfolder, one by one, using recursion.

If it finds the file at any point, it stops searching and returns true. Otherwise, it eventually returns false.

Example folder setup

In main(), the program builds an example folder structure:

- A root folder containing two files.

- Two subfolders inside the root:

- The first subfolder has two files.

- The second subfolder has one file and a deeper nested folder.

- The nested folder inside the second subfolder contains two files, including the target file "target.txt".

Running the search

The program then asks: Is "target.txt" anywhere inside this structure?

- If it is found, it prints: File found!

- If not, it prints: File not found!

Why recursion?

Recursion makes it easy to explore folders no matter how deep they are, without manually keeping track of which folders to visit next. It matches the natural way we think about "searching everything inside this folder, then in each subfolder, and so on."

Final takeaway

This code demonstrates how to model a folder system and perform a deep search inside it. It's a simple and clear example of how recursion can be used to handle problems with unknown or potentially infinite depth, like exploring nested directories.

Binary Search Program

Overview

This C++ program demonstrates binary search in two ways:

1. Iterative approach (using loops)
2. Recursive approach (using function calls that call themselves)

It also compares how much time each approach takes to find an element in a large sorted array.

What is binary search?

Binary search is an efficient method to find an element in a sorted array. Instead of checking each element one by one, it repeatedly divides the search interval in half:

- It looks at the middle element.
- If it matches the target, you're done.
- If the target is smaller, it searches in the left half.
- If the target is larger, it searches in the right half.

This makes searching much faster (logarithmic time, O(log n)) compared to linear search (O(n)).

Program setup

The program defines an array of size 20,000. The array is filled with numbers from 0 to 19999 in increasing order.

It then searches for the key 19999, which is the last element.

Iterative binary search

The function binarySearchIterative() performs binary search using a loop:

- It keeps updating the low and high boundaries until the element is found or the search range becomes empty.
- Returns the index of the key if found; otherwise returns -1.

Recursive binary search

The function binarySearchRecursive() does the same job but using recursion:

- It repeatedly calls itself, narrowing down the range each time.
- The base case is when the low index is greater than high, meaning the element is not found.

Measuring execution time

The program uses the chrono library to measure how long each search takes:

1. It records the start time before running the iterative search.
2. It records the end time after the search finishes.
3. It does the same for the recursive search.
4. The difference gives the duration in nanoseconds.

Output

The program prints:

- The index where the key was found.
- The time it took in nanoseconds for each method.

Why compare time?

The purpose of measuring and comparing the two methods is to understand performance differences:

- Iterative approach is usually faster and uses less memory.
- Recursive approach is easier to understand but may be slower and use more stack space.

Final takeaway

This code gives a practical example of:

- How to implement and compare iterative and recursive binary search.
- How to measure execution time using C++ chrono features.
- How binary search is much faster than linear search in large arrays.

Which approach is better for large arrays?

Iterative approach

- Better for large arrays.
- Uses a loop, so it does not create extra function calls on the stack.
- Saves memory and avoids stack overflow.
- Generally faster, because it doesn't have the overhead of function call setup/return each time.

Recursive approach

- Works fine for small or moderate arrays.
- But for large arrays, it can cause problems:
- Each recursive call adds a new frame to the call stack.
- If the recursion goes too deep, you risk a stack overflow error.
- Slower due to extra time in creating and destroying stack frames.

Conclusion

For large arrays, iterative binary search is the better choice. It is more efficient, safer, and faster.