Binary search algorithm

***Source:* https://github.com/rusevrosen/codehub.github.io/tree/main/Algorithms**

1. What is binary search algorithm?

Binary search is an efficient algorithm for finding an element in a sorted list of elements. It works by repeatedly dividing the list in half, until the desired element is found or it is clear that the element is not present in the list.

Here is the basic idea behind the binary search algorithm:

1. Compare the element you are searching for (called the "target") with the middle element of the list.
2. If the target is less than the middle element, search the left half of the list. If the target is greater than the middle element, search the right half of the list.
3. Repeat this process until the target is found or it is clear that the target is not present in the list.

For example, consider the following list of integers: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]. If we want to find the number 10 using binary search, we would do the following:

1. Compare 10 to the middle element, 12. Since 10 is less than 12, we search the left half of the list: [2, 4, 6, 8, 10].
2. Compare 10 to the middle element of the left half, 6. Since 10 is greater than 6, we search the right half of the left half: [8, 10].
3. Compare 10 to the middle element of the right half, 8. Since 10 is greater than 8, we search the right half of the right half: [10].
4. We have found the element we were searching for, so we stop the search.

Binary search is an efficient algorithm because it reduces the search space by half with each comparison, so it can find an element in a large list in a relatively small number of steps. However, it only works if the list is already sorted.

2. Code example.

```
#include <iostream>
#include <vector>

using namespace std;
// Function to perform binary search

int binarySearch(vector<int>& list, int target)
{
    // Set left and right indices for the search
    int left = 0;
    int right = list.size() - 1;

    // Continue searching while the left and right indices have not crossed
each other

   while (left <= right){

    // Calculate the middle index

    int middle = (left + right) / 2;
    // Compare the target with the middle element
    if (target == list[middle])
    {
    // If the target is found, return the index
     return middle;
    }
    else if (target < list[middle])
    {
    // If the target is less than the middle element, search the left half
     right = middle - 1;
    }
    else
    {
    // If the target is greater than the middle element, search the right
half
     left = middle + 1;
    }
   }
    // If the target is not found, return -1
    return -1;
}

int main() {

 // Define the list of integers

vector<int> list = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

 // Perform a binary search for the number 10

 int index = binarySearch(list, 10);

 // Print the result

 cout << "The number 10 is at index " << index << endl;

return 0;

}
```

3. Code explanation.

This code performs a binary search on a sorted list of integers. The binary search algorithm is used to find the index of a given target value in a sorted list of elements.

The program starts by including the necessary header files: iostream for input/output operations, and vector for using dynamic arrays (vectors). The using namespace std statement allows the use of functions and variables from the std namespace (standard library) without having to specify the namespace every time.

The binarySearch() function takes in a reference to a vector of integers (list) and an integer value (target) as arguments. It then initializes two variables, left and right, to 0 and the size of the list minus 1, respectively. These variables will be used to keep track of the indices of the left and right ends of the current search range.

The function then enters a loop, which will continue until the left index becomes greater than the right index. Inside the loop, the function calculates the middle index of the current search range by taking the average of the left and right indices. It then compares the target value with the element at the middle index.

If the target value is equal to the element at the middle index, the function returns the index. If the target value is less than the element at the middle index, the function updates the right index to be one less than the middle index and continues the search in the left half of the current search range. If the target value is greater than the element at the middle index, the function updates the left index to be one more than the middle index and continues the search in the right half of the current search range.

If the target value is not found in the list, the function returns -1.

The main() function then defines a vector of integers and calls the binarySearch() function, passing in the vector and the target value as arguments. It stores the returned index in a variable index and prints the result to the console using the cout stream. Finally, the main() function returns 0 to indicate that the program has completed successfully.

4. Time complexity.

The time complexity of the binary search algorithm in the given code is O(log n). This is because the size of the list being searched is halved at each iteration of the while loop, so the number of iterations is logarithmic in the size of the list. The time complexity is independent of the actual values in the list and only depends on the number of elements in the list.

It's worth noting that the time complexity of binary search is O(log n) only if the list is already sorted. If the list is not sorted, the time complexity would be O(n log n) if the list is sorted before performing the binary search, or O(n^2) if the list is not sorted and a linear search is used instead.

5. *What is the differences between binary search algorithm and binary tree data structure?*

The binary search algorithm is a searching technique that is used to find the index of a target element in a sorted list of elements. It works by dividing the search range in half at each step and comparing the target element with the middle element of the current range. If the target element is equal to the middle element, the search is successful and the index of the middle element is returned. If the target element is less than the middle element, the search continues in the left half of the current range. If the target element is greater than the middle element, the search continues in the right half of the current range. The search continues in this manner until the target element is found or the search range becomes empty, in which case the search is unsuccessful and the function returns -1.

Binary trees, on the other hand, are data structures that are composed of nodes, where each node can have at most two children. They are often used for storing data in a tree-like structure in order to facilitate searching, insertion, and deletion operations.

6. *What is a binary three data structure?*

A binary tree is a data structure that is composed of nodes, where each node can have at most two children. It is called a "binary" tree because each node has at most two children. The children of a node are referred to as the left child and the right child.

The root node is the topmost node in the tree and has no parent. Each node in the tree has a value, and the nodes are organized in such a way that the values of the nodes in the left subtree of a node are less than the value of the node, and the values of the nodes in the right subtree are greater than the value of the node.

Here is an example of a binary tree with integer values:

```
     10
    /  \
   5    15
  / \   / \
 3  7 12 17
```

In this example, the root node has value 10, the left child has value 5, and the right child has value 15. The left child of the root has a left child with value 3 and a right child with value 7, and so on.

Binary trees have many applications, including searching and sorting algorithms, expression trees for arithmetic expressions, and representation of hierarchical data structures. They are also used in computer science for storing data in a tree-like structure in order to facilitate searching, insertion, and deletion operations.

The structure of a binary tree can be represented using pointers in C++. Each node in the tree is represented as an object with a value and pointers to the left and right children. The left and right children of a node can themselves be binary trees, so the structure forms a recursive data type.

7. Code example.

```
#include <iostream>

using namespace std;

struct Node {
  int data;
  Node* left;
  Node* right;
};

// Function to create a new node
Node* createNode(int data) {
  Node* newNode = new Node;
  newNode->data = data;
  newNode->left = newNode->right = nullptr;
  return newNode;
}

int main() {
  // Create the root node
  Node* root = createNode(10);
  // Set the left and right children of the root node
  root->left = createNode(5);
  root->right = createNode(15);
  // Set the left and right children of the left child
  root->left->left = createNode(3);
  root->left->right = createNode(7);
  // Set the left and right children of the right child
  root->right->left = createNode(12);
  root->right->right = createNode(17);
  return 0;
}
```

8. Code explanation.

This code creates a binary tree with integer values and the following structure:

```
   10
  /  \
 5    15
/ \   / \
3  7 12 17
```

The **Node** structure defines a node in the binary tree with three fields: an integer value (data), and pointers to the left and right children (left and right). The **createNode()** function creates a new node with a given value and initializes the left and right children to nullptr.

In the **main()** function, a root node is created with value 10, and left and right children are created with values 5 and 15, respectively. The left and right children of the left child are created with values 3 and 7, and the left and right children of the right child are created with values 12 and 17, respectively.

9. Time complexity.

The time complexity of the binary tree data structure depends on the operations that you perform on it.

Insertion and deletion operations on a binary tree generally have a time complexity of O(h), where h is the height of the tree. The time complexity of these operations can be reduced to O(log n) if the tree is balanced, but it can become O(n) if the tree is unbalanced.

Searching for a specific element in a binary tree has a time complexity of O(h) in the worst case, but it can be reduced to O(log n) if the tree is balanced.

Traversing the elements of a binary tree (e.g., in-order, pre-order, or post-order) has a time complexity of O(n), as all the nodes in the tree must be visited.

It's worth noting that the time complexity of these operations can vary depending on the specific implementation of the binary tree data structure.

I wish you happy learning,

Yours Rosen Rusev