Trees

Laasyasree Desu

Fall 2023

1 Introduction

This assignment involved using linked list to implement a data-structure know as binary tree. Additionally, several methods were also implemented such as adding and searching for an element. Another method implemented was iterator, which has the function of traversing the tree. The methods were also bench-marked to find the run-time and time complexity.

2 Binary Tree

A tree is a data-structure that is implemented using the features of linked lists. A tree starts at the root which is the starting node and continuously separates into different branches, which are new nodes added. It can be seen as a hierarchical structure branches as different levels and the root at the top. This process of continuous division makes the tree a recursive data-structure. A binary tree specifically is a tree that only branches into two branches, one left and one right. Each node also has a key and an element, where the key acts as a pointer and maps to the specific element. The end node of the tree, at the last branch is called a leaf. These end nodes point towards null. The tree is also sorted so that compared to the root, smaller keys are sorted to the all left branches while keys larger are sorted to all the right. As mentioned before, the implementation of a tree uses linked lists. Some parts of the code for this was already provided, the additional parts included adding constructors and making a for loop to create the tree. A section of the code is seen below:

```
public Integer value;
public Node left, right;

public Node(Integer key, Integer value) {
    this.key = key;
    this.value = value;
    this.left = this.right = null;
}
```

2.1 Add method

The add method involves adding new nodes to existing leaves in the tree. If a node with a certain key already exists then the element it maps to should be updated. Otherwise if the key is smaller, the node should be added to the left branch. If its greater, the node is added to the right branch. The implementation started with the method being created with two parameters - the key and the element. An if statement would then check if the tree is empty (equal to null) and in that case the root would instead be updated to equal the new node. The second if statement checked if the new node is equal to the current, which in that case the element would be updated and the method execution terminated (returned). Otherwise two more if else statements compared the new nodes key to the current node (smaller or bigger), checked if they are equal to null and added to the left and right branches accordingly. In case they were not null the key and elements were updated. The example below shows a section of the code:

```
if (key > newkey){
    if (left == null)
        left = new Node (newkey, element)
    else
        left.add(newkey, element)
}
```

2.2 Search method

The search or "lookup" method was implemented similar to the add method. The purpose of this method was to find the element or value at a specific node. To achieve this a method with the key as a parameter was created. The method started the root as the current node and entered a while loop with the condition of the root not being equal to null. In the while loop, if the current node is equal to key parameter, the value at the key was returned. Otherwise if the key was smaller then the current node would be updated to the left branch. If it was bigger then it would update to the right branch and continue searching. A section of the code can be seen below:

```
if ( key == newkey )
    return element ;
else if ( key < newkey )
    right.search(newkey)
else
    left.search(newkey)
```

2.2.1 Benchmark

Array size	Time taken (ns)
100	79
200	112
400	98
800	125
1600	131

Table 1: Average time taken to a binary tree compared to binary search algorithm for various sizes of trees/arrays.

The algorithm for the search through the tree was bench-marked and the results are shown above (ref. table 1). The results have slight variations that can be attributed to different factors. The first could be due to the balance of the branches. The time complexity can vary depending on the distribution of the left and right branches and the position of the node being searched for. In the worst case scenario, where for example the node is in the far right of the right sub-tree then the entire tree would have to be searched, which would result in the time complexity O(n) where n is the number of nodes. If the tree however has relatively equally distributed branches then it could result in a time complexity of O(log(n)). This can be compared to the binary search algorithm which uses the same principle of dividing into two parts (recursively).

2.3 Iterator

The third task of this assignment was to create or implement a method to traverse the binary tree. This method, called iterator can be thought of as an in-order traversal of the tree. This means that the left sub-tree is first traversed, then root is reached and then the right sub-tree is traversed.

To achieve this the iterator uses three methods - Next, HasNext and Remove. In this particular assignment the remove method was not implemented therefore it was set to throw an exception if called upon. The iterator also implemented a stack to keep track of the elements in the tree.

2.3.1 Next method

This method had the function of keeping track of and returning the next element in the tree during in-order traversal. The method started by verifying that there exists a next element (using the has next method explained below). If there is no element an exception would be thrown. Using a while loop, it then traverses the left sub-tree, continuously moving to the left until it reaches a leaf or a node with no left branch. Throughout the traversal the nodes would be pushed onto the stack and upon reaching the "end" the first node on the stack is then popped and updated as the "current node". the method then checks if the node has a right branch which the current node is then updated to and the element at the node is returned. The process is looped to continue until the entire tree is traversed. Below is a section of the code:

```
public Integer next() {

    while (curr != null) {
        stack.push(curr);
        curr = curr.left;
    }
```

2.3.2 HasNext method

The has next method had the function of checking that there exists additional elements to be traversed. This was implemented as a boolean method that would return false if the stack was found empty (there are no elements left), otherwise true. This can be seen in the code example below:

```
public boolean hasNext(){
        if(stack.isEmpty)
        return false;
return true;
    }
```

To summarise the iterator uses two main functions and the stack to be able to traverse the tree while keeping track of the nodes to return the elements. Due to this process, adding new elements would only work if the node in question have not already been traversed, otherwise the new element goes unaccounted for.