Trees

Mahdi Nazari

September 2022

Introduction

In this assignment we will discuss about tree data structure and specifically about a binary tree. It will be explained which attributes does this data structure have. Some methods as add and lookup will be implemented for this data structure. We will compare the lookup method which is a search method in a binary tree with the binary search in a sorted array from previous assignment and benchmark them with an arising data sets. To have access to each node in a tree a new data structure which is called Iterator will be presented and we will explain which methods does this data structure contain and how do they work.

Binary tree

A binary tree as other type of trees has a root where the tree will be initialized. Binary tree consist of nodes which has some properties as a key, a value and references to its left or right neighbour nodes. Beyond this, in a binary tree every nodes including the root has two branches, namely left and right branches. All nodes with a smaller key will be placed on the left branch and nodes with larger key on the right branch.
A binary tree as other data structures has a constructor and some methods. In this assignment we will discuss about add and lookup methods. Add method will insert a node in the tree and depends on its key, it will be inserted in a certain place in the tree. If the key is larger than the root it will be inserted on the right branch of the root otherwise it will be inserted in the left branch and if the key is the same as the root, add function will update the value of the root. Add function looks like as below.

```
public Node add(Integer k, Integer v){
    if(this.key == k){
                this.value = v;
            }
    if(this.key > k){
```

```
                if(this.left == null)
                    this.left = new Node(k, v);
                else
                    this.left = this.left.add(k,v);
            }
        else{
                if(this.right == null)
                    this.right = new Node(k, v);
                else
                    this.right = this.right.add(k,v);
            }
            return this;
}
```

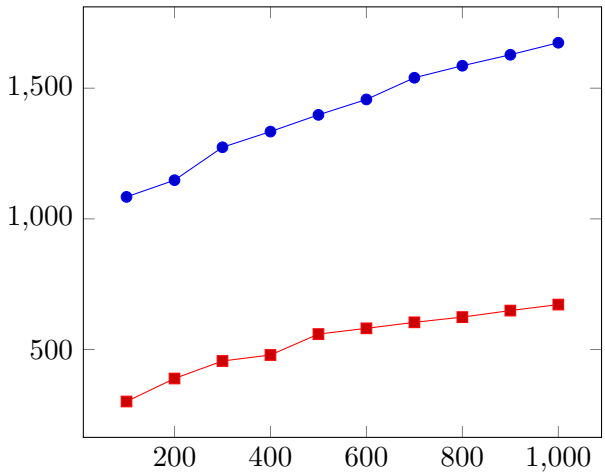
Lookup method is as a search function and will return the value of the key which have been sent to the function. The method will search after a node by its key, e.g. if the key is smaller than the root it will search on the left branch of the root. When it goes down to the left branch it will compare again the key with the key of the next node and will continue to the left or right branch of that node. This procedure goes on until the node with the searched key has been found and then the function will return the value of that key.

Binary search vs Lookup

In this part we will compare two search methods that have very similar algorithms. Binary search in a sorted array and lookup in a binary tree. From previous assignment we know that binary search in a sorted array has a time complexity of O(log n) type regard to the size of array. A benchmark was constructed with rising data sets. These two search methods were benchmarked and the results is as in the Table 1 and Graph 1 below.

Number	Binary search	Lookup
100	1084	301
200	1148	389
300	1274	456
400	1334	479
500	1398	559
600	1457	581
700	1540	604
800	1586	624
900	1628	649
1000	1674	672

Table 1: The average time for finding an element with binary search and lookup function. Time in ns



Graph 1: The result from benchmarking binary-search(blue graph) vs lookup method in a binary tree(red graph). x-axis is size of data sets, y-axis is execution time in ns.

As the result shows even the lookup function(red graph) has a time complexity of O(log n) type. This is because the keys in a binary tree is sorted and the program does not need to go through all nodes to find the searched key. Program will jump to the right- or left branch of the root if the root is not equal to the key. Then lookup function will update the root with next node and do the same, either it will be a hit or the program should jump to left- or right branch. In other words, lookup function will perform as a binary search does in a sorted array. The execution time in a binary tree depends also on how a tree is constructed and if the tree is balanced or not. If the tree was not balanced e.g. a sorted sequence of key was inserted to the tree then one of the trees branch would be long and searching in that

branch would take the same time as searching in an array. Searching in a branch with (n) nodes in raw will entail a time complexity of O(n) type.
Another observation from the benchmarking is that even both of binary search and lookup are O(log n) type but the execution time in a binary search is longer than the lookup. This can be interpreted from the Table 1 and Graph 1. This is an interesting observation and I am very insure about the reason. But it can be explained as follows: in the lookup the program does not need to read the keys value every time and just compare the key with the searched key but in a binary search the program should read the keys value after every jump to decide either if it is a hit or it should jump to the right or left next and this maybe cause a longer execution time compared to the lookup.

Iterator

To have access to each node in a tree a new data structure will be necessary and this data structure will be an iterator. An Iterator has two attributes a node(next) and a stack. The iterator was constructed based on first-depth order. This means that the iterator will traverse from leftmost node towards rightmost node in the tree and this gives us accessibility to the nodes in order from the smallest key in the tree to the largest key in the tree.
The constructor of the iterator looks like as follows:

```
public TreeIterator() {
      stack = new Stack<BinaryTree.Node>();
      next = BinaryTree.root;
      while(next != null){
          stack.push(next);
          next = next.left;
      }
 }
```

To construct an iterator a node(next) and a stack was needed. The stack which is used in this task is a dynamic stack from previous assignment. Iterator will push all leftmost nodes in the left branch of the root to the stack. To traverse in the tree the iterator needs other methods as hasNext() and next(). hasNext() method will keep track of the elements in the stack. When the hasNext() function return a boolean false then the stack is empty and it means that the iterator has traversed through all nodes in the tree. The hasNext() is just one rad of code as follows:

```
public boolean hasNext() {
      return !stack.isEmpty();
  }
```

To print out the value of the keys in a tree we need another method in the iterator which will be called next() function. Every time the next() is called it will return the value of the smallest key in the tree which we have not been traversed through. next() function is constructed as follows:

```
public Integer next() {
        if(hasNext() == false){
            return null;
        }
        BinaryTree.Node tempNode = stack.pop();
        while(tempNode.right != null){
            stack.push(tempNode.right);
            tempNode.right = tempNode.right.left;
        }
        return tempNode.value;
    }
```

First next() function will check if the stack is empty or not. If the stack is not empty then the stack will pop the last element that it had pushed. After that the next() function will check if there is a branch on the right side of the poped element. If the answer is yes, then it will push the all nodes in the leftmost of that branch to the stack. If there is no branch to the right side of the poped element, the next() function will traverse up towards the root. Next time when the next() function is called it will go through the same procedure and traverse in the tree. The iterator will have returned all keys values in the tree when hasNext() returns a boolean false.
If the next() function has been called a few times and it has retrieved some values and then we add another node with a key and value to the tree, the iterator can miss this key if the key is smaller than the last key which was poped from the stack. It means that the added node will end up in that part of the tree that the iterator has already passed by. To avoid that the added node will be missed by iterator, the added key should at least be larger than last poped keys right key (if there is one) otherwise it is enough that the added key is larger than the last poped key. Then the node will end up in that part of the tree that iterator has not traversed through and the added node will be catched by iterator.