CPSC 320: Intermediate Algorithm Design and Analysis
Assignment #5, due Thursday, March 8th, 2012 at 11:00

[4] 1. Suppose that we have a potential function Φ such that, for every $i \geq 0$,

- $\Phi(D_i) \geq 0$, and
- $\Phi(D_i) = \Phi(D_{i-1}) + \mathit{cost}_{\mathit{am}}(\mathit{op}_i) - \mathit{cost}_{\mathit{real}}(\mathit{op}_i)$

but that $\Phi(D_0) \neq 0$.

Prove that the worst-case running time of a sequence of n operations on this data structure is in $O(\Phi(D_0) + \sum_{i=1}^{n} \mathit{cost}_{\mathit{am}}(\mathit{op}_i))$.

Solution: We use the fact that

$$\mathit{cost}_{\mathit{am}}(\mathit{op}_i) = \mathit{cost}_{\mathit{real}}(\mathit{op}_i) + \Phi(D_i) - \Phi(D_{i-1})$$

Summing over all values of i, we get

$$\sum_{i=1}^{n} \mathit{cost}_{\mathit{am}}(\mathit{op}_i) = \sum_{i=1}^{n} (\mathit{cost}_{\mathit{real}}(\mathit{op}_i) + \Phi(D_i) - \Phi(D_{i-1}))$$

and, using the fact that the $\Phi(D_i) - \Phi(D_{i-1})$ terms mostly cancel each other out, we obtain

$$\sum_{i=1}^{n} \mathit{cost}_{\mathit{am}}(\mathit{op}_i) = \left(\sum_{i=1}^{n} \mathit{cost}_{\mathit{real}}(\mathit{op}_i)\right) + \Phi(D_n) - \Phi(D_0)$$

Hence

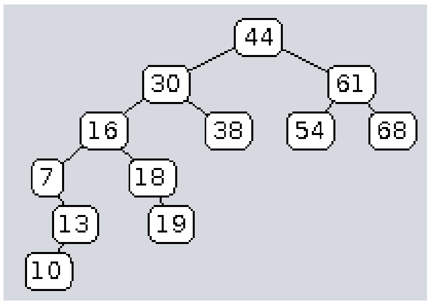
$$\sum_{i=1}^{n} \mathit{cost}_{\mathit{real}}(\mathit{op}_i) = \left(\sum_{i=1}^{n} \mathit{cost}_{\mathit{am}}(\mathit{op}_i)\right) + \Phi(D_0) - \Phi(D_n)$$

Finally, since $\Phi(D_n) \geq 0$, we get

$$\sum_{i=1}^{n} \mathit{cost}_{\mathit{real}}(\mathit{op}_i) \leq \left(\sum_{i=1}^{n} \mathit{cost}_{\mathit{am}}(\mathit{op}_i)\right) + \Phi(D_0)$$

as required.

[12] 2. In this problem, we consider a sequence of n `successor` operations on a binary search tree. The `successor` operation finds the node whose key would immediately follow the key stored in the current node if we listed the keys stored in the tree in order. For instance, in the following tree, the successor of the node with key 44 is the node with key 54, while the successor of the node with key 19 is the node with key 30.



The `successor` operation needs to handle two separate cases:

- If the current node has a right child (for instance the node with key 44), then it goes right once, and then left as long as possible.
- If the current node has no right child (for instance the node with key 19), then it goes up until it arrives at a node from its left child.

Here is Java code for this operation:

```
public BinaryTreeNode<ElementType> getSuccessor(BinaryTreeNode<ElementType> node)
{
    if (node.getRightChild() != null)
    {
        return minimum(node.getRightChild());
    }
    while (node.getParent() != null && node == node.getParent().getRightChild())
    {
        node = node.getParent();
    }
    return node.getParent();
}

public BinaryTreeNode<ElementType> minimum(BinaryTreeNode<ElementType> node)
{
    while (node.getLeftChild() != null)
    {
        node = node.getLeftChild();
    }
    return node;
}
```

Using the potential method, prove that a sequence of n `successor` operations on a binary search tree with n elements, starting from an unspecified element that is *before* the first element of the tree[1] will run in $O(n)$ time.

Hint: use $\Phi(D_i) = r_i + (n - l_i)$ where r_i is the number of "right edges" (edges that go from a node to its right child) on the path from the root of the tree to the current node and l_i is the number of "left edges" on the same path.

[1] You can think of adding a node "null" that has no left child, and the real root as its right child, and then starting at this "null" node.

Solution: We start by analyzing the amortized cost of the ith `successor` operation. We break the analysis down into two subcases:

- If the current node has a right child, then `successor` goes right once, and then left k times for some integer $k \geq 0$. The real cost of the operation is $k+1$. The potential goes up by 1 for the move right, and then goes down by 1 for each of the left moves, and hence $\Phi(D_i) - \Phi(D_{i-1}) = 1 - k$. Therefore

  $$cost_{am}(op_i) = (k+1) + (1-k) = 2$$

- If the current node does not have a right child, then `successor` goes up from a right child k times, and then up from a left child once. The real cost of the operation is $k+1$. The potential goes down by k for each of the first up moves from a right child, and then up by 1 for the move from a left child, and hence $\Phi(D_i) - \Phi(D_{i-1}) = 1 - k$. Therefore

  $$cost_{am}(op_i) = (k+1) + (1-k) = 2$$

The initial potential is $0 + (n - 0) = n$, and so by the theorem proved in question 1, the worst-case running time of the sequence of n operations is in $O(n + \sum_{i=1}^{n} 2)$ which is $O(n)$.

[12] 3. Suppose that we want to implement a dynamic, open-address hash table. We have a load factor α which is defined by:

$$\alpha = n/s$$

where n is the number of elements in the hash table, and s is the size of the array used to store these elements. One possible scheme to make sure that the array is neither too big nor to small is the following:

- If an insertion causes α to become greater than or equal to 90%, then we allocate a new array whose size is 1.5 times the size of the old array, and copy all of the elements to the new array.
- If a deletion causes α to become less than or equal to 40%, then we allocate a new array whose size is 2/3rd the size of the old array, and copy all of the elements to the new array.

Use the potential method to prove that every sequence of t operations on an initially empty hash table using this scheme runs in $O(t)$ time. You may assume the following:

- inserting or deleting an element (without reallocating the array) takes $\Theta(1)$ time.
- creating a new array A' to replace an old array A, and copying all of the elements of A into A', takes in $\Theta(\max\{\text{length}(A), \text{length}(A')\}$.

Hints: use $\Phi(D_i) = 5n_i$, where n_i is the number of operations that were performed on the data structure since the last time the array was reallocated. When you compute the amortized cost of an insertion or deletion that causes the array to be reallocated, determine the amortized cost of the insertion or deletion, and the amortized cost of the reallocation operation, separately.

Solution: First, observe that if the array is 90% full, then after it is reallocated it is only 90%/1.5 = 60% full. If it becomes only 40% full, then after the reallocation it is 40% $*$ 1.5 = 60% full.

To prove the result we want, it will be enough to prove that the amortized cost of each operation is in $\Theta(1)$. For both insertions and deletions, an operation that does not result in a reallocation has a real cost of 1, and the potential difference is $\Phi(D_i) - \Phi(D_{i-1}) = 5$. So the amortized cost of the operation is 6.

Let us now consider operations that force a reallocation of the array. We can split them into two distinct operations: the insertion or deletion (without reallocation), and the array reallocation. The first part has an amortized cost of 6, but what about the second part?

Insertions After the previous reallocation, the array was 60% full. Hence $n_i \geq 30\%s$ where s is the size of the array before we perform the reallocation. The real cost of the reallocation is $1.5s$. The potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = 0 - 5 * 0.3s = -1.5s$$

Hence the amortized cost of the reallocation is $1.5s - 1.5s = 0$.

Deletions After the previous reallocation, the array was 60% full. Hence $n_i \geq 20\%s$ where s is the size of the array before we perform the reallocation. The real cost of the reallocation is s. The potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = 0 - 5 * 0.2s = -s$$

Hence the amortized cost of the reallocation is $s - s = 0$.

Thus the amortized cost of an insertion or deletion that forces the array to be reallocated is also 6. This proves that a sequence of t insertions and/or deletions into an array that is initially empty runs in $O(t)$ time.