Data Representation and Abstraction

Generated by LectureMate

February 22, 2025

1 Introduction

In this lecture, we will discuss data representation and possibly touch on data abstraction. The focus will be on different representations of sets and their efficiencies. We will also cover the upcoming lectures, including a special guest lecture by John Hughes from Chalmers University, who will discuss QuickCheck.

2 Recap of Previous Lecture

Last time, we discussed the rates of growth of functions used to measure the efficiency of programs. We categorized these rates as follows:

- Linear
- Quadratic
- Cubic
- Logarithmic

We emphasized that logarithmic growth is preferable as it indicates better efficiency.

3 Data Representation of Sets

We will explore four different representations of sets, starting with sets as lists.

3.1 Sets as Lists

The basic operations for sets represented as lists include:

- Constant (empty set)
- Insert function
- Function to convert lists to sets
- Membership function (element)
- Equality testing on sets

For example, the implementation of these operations for sets of integers was discussed.

3.2 Sets as Ordered Lists

To improve efficiency, we can represent sets as ordered lists. This representation allows for easier equality testing since all elements are in ascending order. The equality testing for unordered lists is quadratic, while for ordered lists, it becomes linear.

3.2.1 Code Example

The code for inserting an element into an ordered list must maintain the order:

```
insert :: (Ord a) => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys)
    | x < y     = x : y : ys
    | x == y    = y : ys
    | otherwise  = y : insert x ys
```

3.3 Efficiency of Operations

The efficiency of operations for ordered lists is as follows:

- Empty set: Constant time
- Inserting an element: Linear time
- Converting a list to a set: Quadratic time
- Membership: Linear time
- Equality: Linear time

4 Sets as Trees

Next, we will discuss representing sets as trees, specifically ordered binary trees. This representation allows for logarithmic time complexity for membership checks.

4.1 Ordered Binary Trees

An ordered binary tree has the property that for any node, all elements in the left subtree are less than the node's value, and all elements in the right subtree are greater.

4.1.1 Insertion in Trees

The insertion process involves traversing the tree to find the correct position for the new element:

```
insertTree :: (Ord a) => a -> Tree a -> Tree a
insertTree x Empty = Node x Empty Empty
insertTree x (Node y left right)
    | x < y     = Node y (insertTree x left) right
    | x > y     = Node y left (insertTree x right)
    | otherwise  = Node y left right
```

4.2 Efficiency of Tree Operations

The efficiency of operations for trees is as follows:

- Empty set: Constant time
- Inserting an element: Logarithmic time
- Converting a list to a set: $O(n \log n)$
- Membership: Logarithmic time
- Equality: Linear time

5 Balanced Trees

To ensure efficiency, we need to consider balanced trees, such as AVL trees. These trees maintain balance during insertions, ensuring that the depth remains logarithmic relative to the number of elements.

5.1 AVL Trees

An AVL tree is a self-balancing binary search tree where the difference in heights between the left and right subtrees is at most one. This property ensures that operations remain efficient.

5.1.1 Rebalancing

After an insertion, a local rebalance may be necessary to maintain the AVL property. This involves rotations to ensure balance.

6 Conclusion

In this lecture, we covered various data representations, focusing on sets as lists, ordered lists, and trees. We also introduced the concept of balanced trees to maintain efficiency. The next lecture will continue this discussion and delve deeper into AVL trees.