Functional Programming

Purely Functional Data Structures

Prepared by: Fatima al Rabih 221517 , Lea Chamseddine 220335, Karen Khoury 220402

Introduction : What is a purely Funct. Data Structure?

A purely functional data structure is one that is designed to work within a functional programming paradigm, where **immutability** and **referential transparency** are key principles.

Immutability: Once created, it cannot be modified

Referential transparency: No Side Effects. An expression can be replaced with its value.

Advantages

No Hidden State: Easier to debug
Persistent: Previous versions remain accessible after updates
Memory: Structural Sharing
Concurrency: purely functional data structures are automatically thread safe

Structures Explored:

- **Lists / Streams / Haskell Std lib**
- **Binomial Heap**
- **23 Trees**
- **Zippers**
- **Finger Tree**

Language Used:

- **Haskell**
- **Python**

Benchmarking

Criterion: To measure the execution time of Haskell code with high precision and reliability.

Deepseq: To force the FULL evaluation of data structures instead of lazily evaluation which is important for controlling memory usage and performance

PLEASE NOTE: Our benchmark is forcing haskell to fully evaluate our data structures. So in reality, the time measured and the efficacity of those data structures should be

Lists Implementation - Haskell

File: list.py and List1.hs

```haskell
-- Custom List Definition
data List a = Empty | Cons a (List a) deriving (Show, Eq)

-- Making List an instance of NFData
instance NFData a => NFData (List a) where
  rnf Empty = ()
  rnf (Cons x xs) = x `deepseq` rnf xs
```

INSTANCE OF THE TYPE CLASS NFData -> can be fully evaluated using deepseq

rnf method recursively forces evaluation during benchmarking

Implemented Functions: map,filter,fold,toList,fromList

- toList: Converts a standard Haskell list into a custom List (a list of Cons).
- fromList: Converts a custom List back into a standard Haskell list.

Complexity:

- append : O(n)
- Insert : O(1)
- AddFirst : O(1)
- Search : O(n)

Mutable Lists: (normal concatenation)

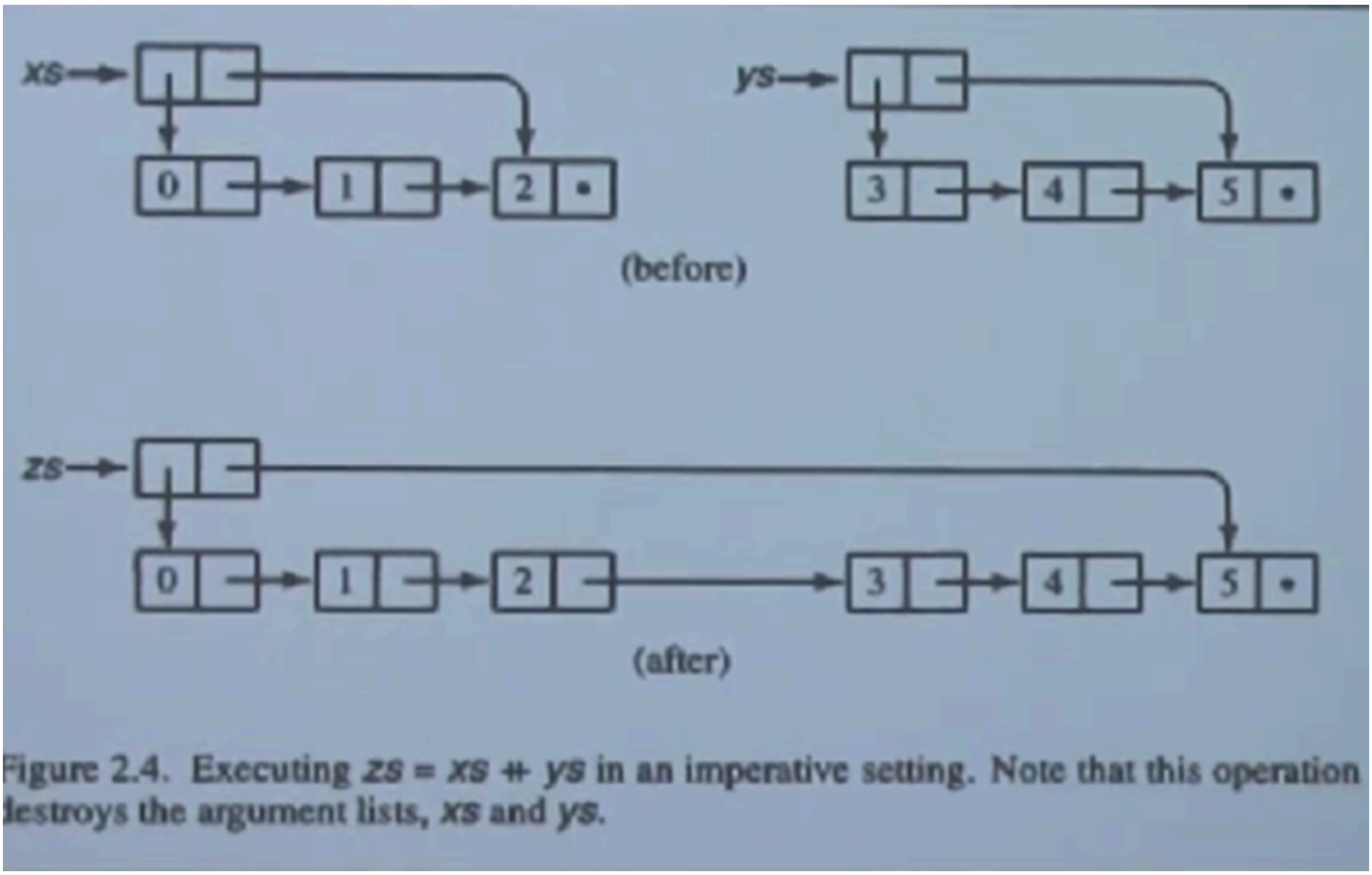
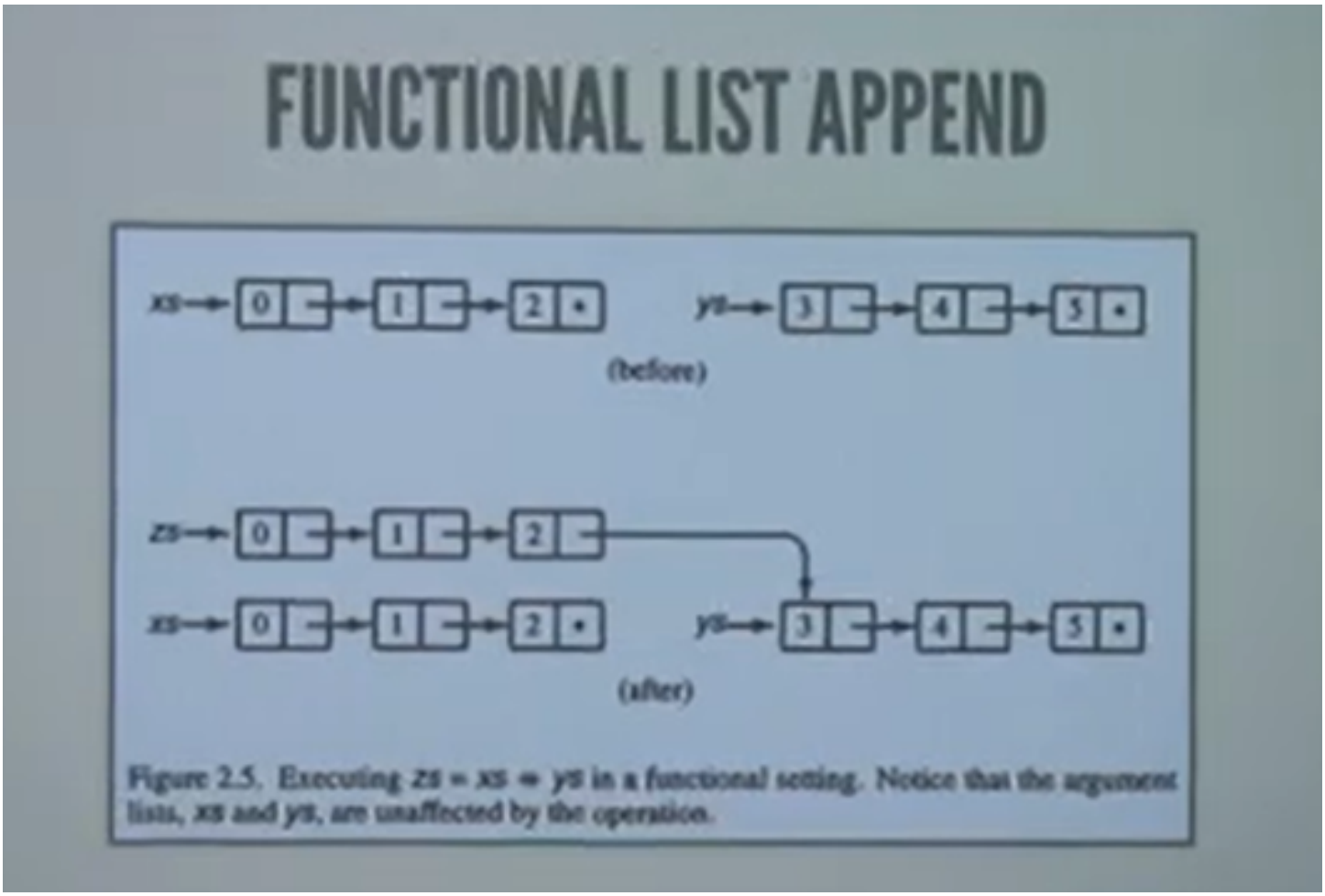


Figure 2.4. Executing *zs = xs ++ ys* in an imperative setting. Note that this operation destroys the argument lists, *xs* and *ys*.

y has not changed, but x is not only formed by 3 elements anymore, because the Node 2 has now a reference to node 3 => Mutation of x

```
def concatenate_lists(lst1, lst2):
    # If lst1 is empty, return lst2
    if lst1 is None:
        return lst2
    # Otherwise, find the tail of lst1 and attach lst2 to it
    current = lst1
    while current.next:
        current = current.next
    current.next = lst2
    return lst1
```

Immutable Lists Concatenation:



We copy XS because of immutability.

Increase Usage of memory? To compensate: Structural Sharing / Lazy Evaluation

```haskell
-- Appending two Lists
append :: List a -> List a -> List a
append Empty ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Lists Implementation - Python

Linked List

```python
class ListNode:
    def __init__(self, value, next_node=None):
        self.value = value
        self.next = next_node

    def __str__(self):
        values = []
        current = self
        while current is not None:
            values.append(str(current.value))
            current = current.next
        return " -> ".join(values)
```

Feature	Python Linked List	Haskell Linked List
Mutability	Mutable	Immutable
Ease of Implementation	Manual implementation required	Built-in with extensive support
Memory Efficiency	Higher overhead due to explicit nodes	Efficient due to structural sharing
Lazy Evaluation	Not supported	Supported, enabling infinite lists
Performance	Manual optimization needed	Optimized for immutability

Lists Implementation Haskell benchmark

```
benchmarking mapList/smallList
time                 1.160 µs   (1.149 µs .. 1.172 µs)
                     0.999 R²   (0.999 R² .. 1.000 R²)
mean                 1.164 µs   (1.155 µs .. 1.173 µs)
std dev              30.66 ns   (25.41 ns .. 37.51 ns)
variance introduced by outliers: 35% (moderately inflated)

benchmarking mapList/largeList
time                 35.16 ms   (34.05 ms .. 36.35 ms)
                     0.996 R²   (0.993 R² .. 0.999 R²)
mean                 33.58 ms   (33.06 ms .. 34.26 ms)
std dev              1.213 ms   (937.7 µs .. 1.573 ms)
variance introduced by outliers: 11% (moderately inflated)
```

```
benchmarking reverseList/smallList
time                 1.476 µs   (1.411 µs .. 1.547 µs)
                     0.988 R²   (0.984 R² .. 0.994 R²)
mean                 1.411 µs   (1.363 µs .. 1.461 µs)
std dev              168.1 ns   (140.1 ns .. 212.1 ns)
variance introduced by outliers: 92% (severely inflated)

benchmarking reverseList/largeList
time                 33.59 ms   (29.22 ms .. 40.50 ms)
                     0.936 R²   (0.880 R² .. 0.987 R²)
mean                 40.36 ms   (37.72 ms .. 46.51 ms)
std dev              7.922 ms   (3.297 ms .. 14.33 ms)
variance introduced by outliers: 72% (severely inflated)

Memory in use: 77594624 bytes
```

```
benchmarking filterList/smallList
time                 938.4 ns   (922.0 ns .. 958.9 ns)
                     0.997 R²   (0.996 R² .. 0.999 R²)
mean                 924.0 ns   (909.6 ns .. 937.8 ns)
std dev              48.33 ns   (39.66 ns .. 59.62 ns)
variance introduced by outliers: 69% (severely inflated)

benchmarking filterList/largeList
time                 26.18 ms   (25.65 ms .. 26.89 ms)
                     0.995 R²   (0.988 R² .. 0.999 R²)
mean                 25.82 ms   (25.37 ms .. 26.31 ms)
std dev              1.031 ms   (804.6 µs .. 1.332 ms)

benchmarking foldList/smallList
time                 498.8 ns   (491.8 ns .. 505.1 ns)
                     0.998 R²   (0.997 R² .. 0.999 R²)
mean                 506.2 ns   (498.2 ns .. 515.9 ns)
std dev              29.57 ns   (24.33 ns .. 37.50 ns)
variance introduced by outliers: 74% (severely inflated)

benchmarking foldList/largeList
time                 17.56 ms   (17.09 ms .. 18.07 ms)
                     0.995 R²   (0.989 R² .. 0.998 R²)
mean                 17.05 ms   (16.71 ms .. 17.37 ms)
std dev              782.0 µs   (574.0 µs .. 1.040 ms)
variance introduced by outliers: 16% (moderately inflated)
```

Lists Implementation

Benchmark Group	Benchmark Name	Time (Mean)
mapList	`smallList`	1.236 µs
	`largeList`	33.81 ms
filterList	`smallList`	925.7 ns
	`largeList`	26.26 ms
foldList	`smallList`	472.7 ns
	`largeList`	16.43 ms
reverseList	`smallList`	1.177 µs
	`largeList`	31.26 ms
append	`small + small`	1.408 µs
	`large + small`	24.46 ms
insert	`insert at start`	819.3 ns
	`insert at middle`	1.193 µs
	`insert at end`	1.746 µs
listLookup	`lookup 2 in smallList`	377.9 ns
	`lookup 100000 in largeList`	15.68 ms

Streams - Lazy Evaluated lists

PS: In Haskell all lists are Steams.

Advantages:

- Good for defining sequences can be infinite
- Append is still O(n) but paid as it's used

Side effect of Lazy Evaluation:

- In non-functional programming, lazy evaluation can lead to unintended consequences.
- log messages may not be written as expected.
- This behavior can result in hidden or missing logs, making debugging and monitoring more challenging.

Binary Tree

Binary Tree Data Structure

The binary tree is a fundamental structure used to store data hierarchically. Here's how it's defined:

`data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)`

- `Tree a` represents a binary tree where each `Node` contains:
  - A value of type `a`.
  - A left subtree.
  - A right subtree.
- `Empty` represents a leaf or an empty tree.

For example, a tree like this:

```
  1

/ \

2  3
```

is represented as:

`Node 1 (Node 2 Empty Empty) (Node 3 Empty Empty)`

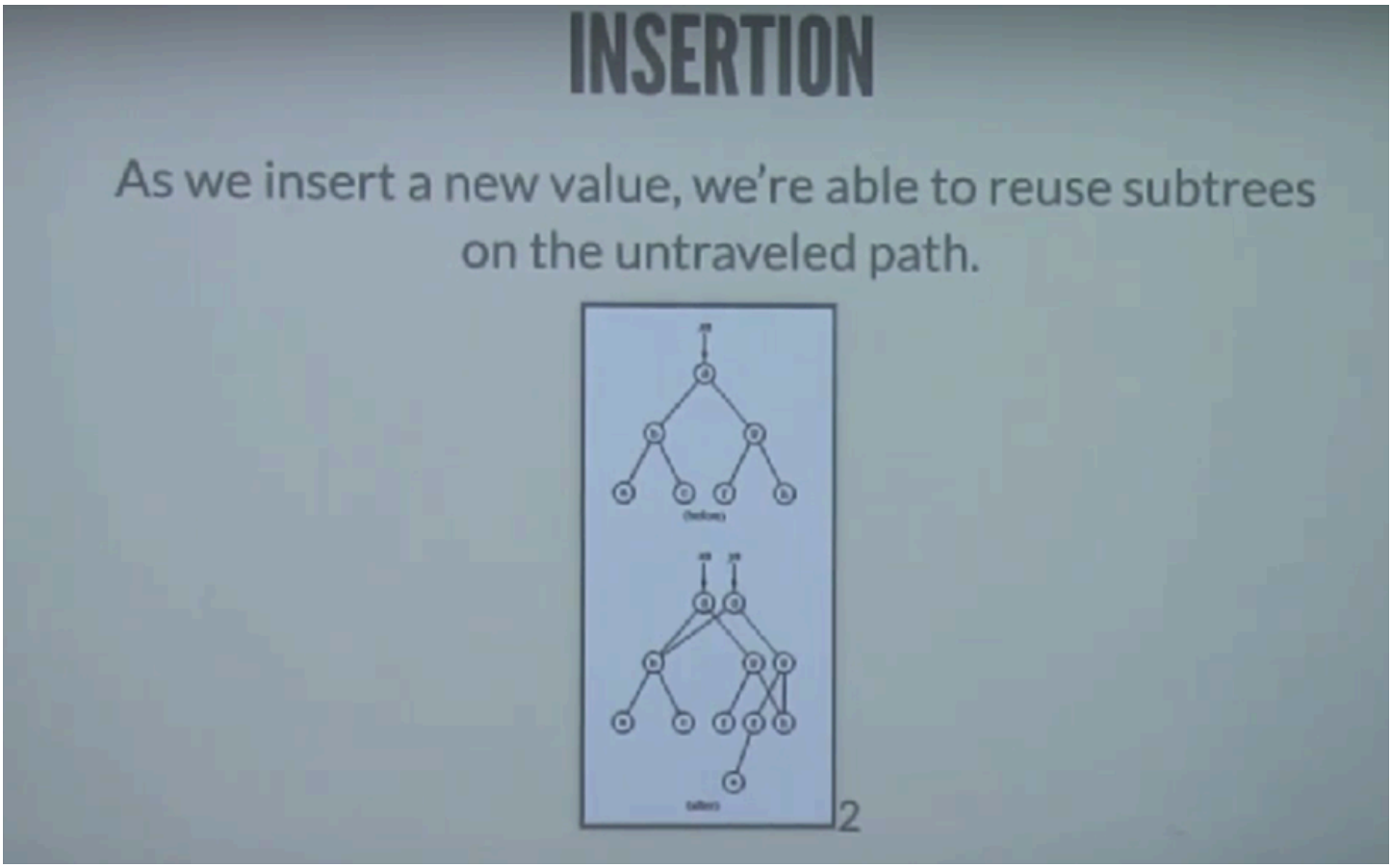
```haskell
data BinaryTree a = Empty
                  | Node a (BinaryTree a) (BinaryTree a)
                  deriving (Show, Eq)

-- Insert an element into the binary search tree
insert :: (Ord a) => a -> BinaryTree a -> BinaryTree a
insert x Empty = Node x Empty Empty
insert x (Node v left right)
  | x < v     = Node v (insert x left) right
  | x > v     = Node v left (insert x right)
  | otherwise = Node v left right  -- No duplicates allowed

-- In-order traversal
inOrder :: BinaryTree a -> [a]
inOrder Empty = []
inOrder (Node v left right) = inOrder left ++ [v] ++ inOrder right
```

Complexity: (balanced tree)

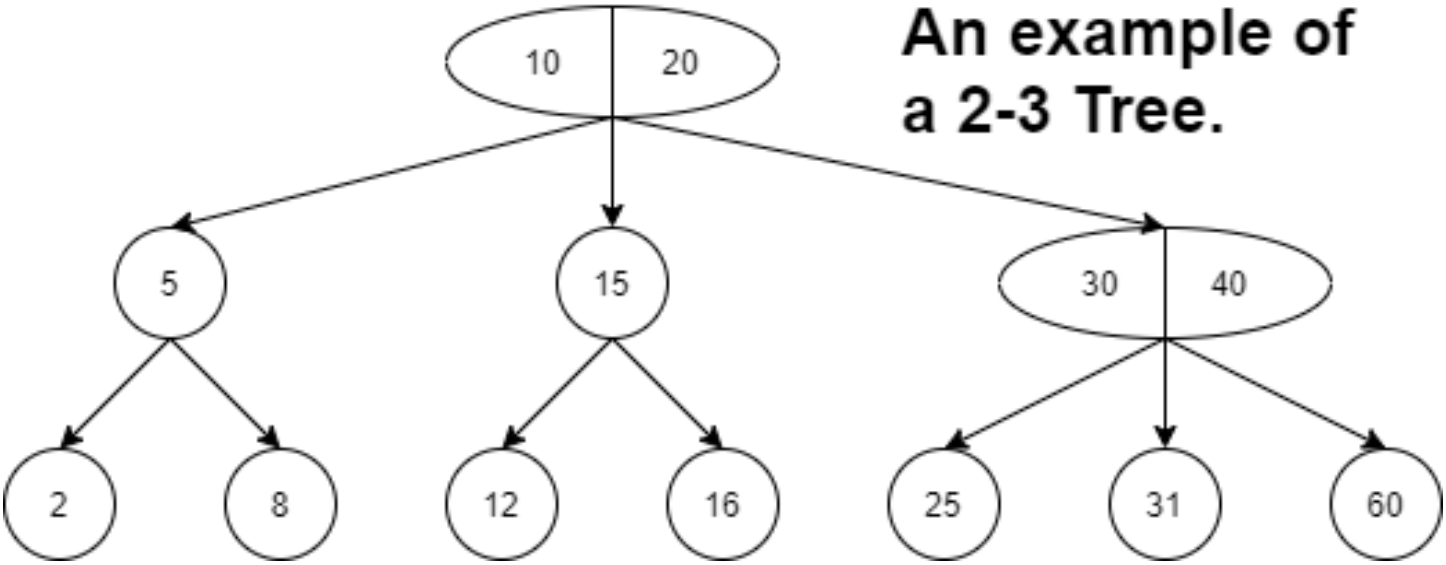
- Insertion O(log n)
- Search O(log n)



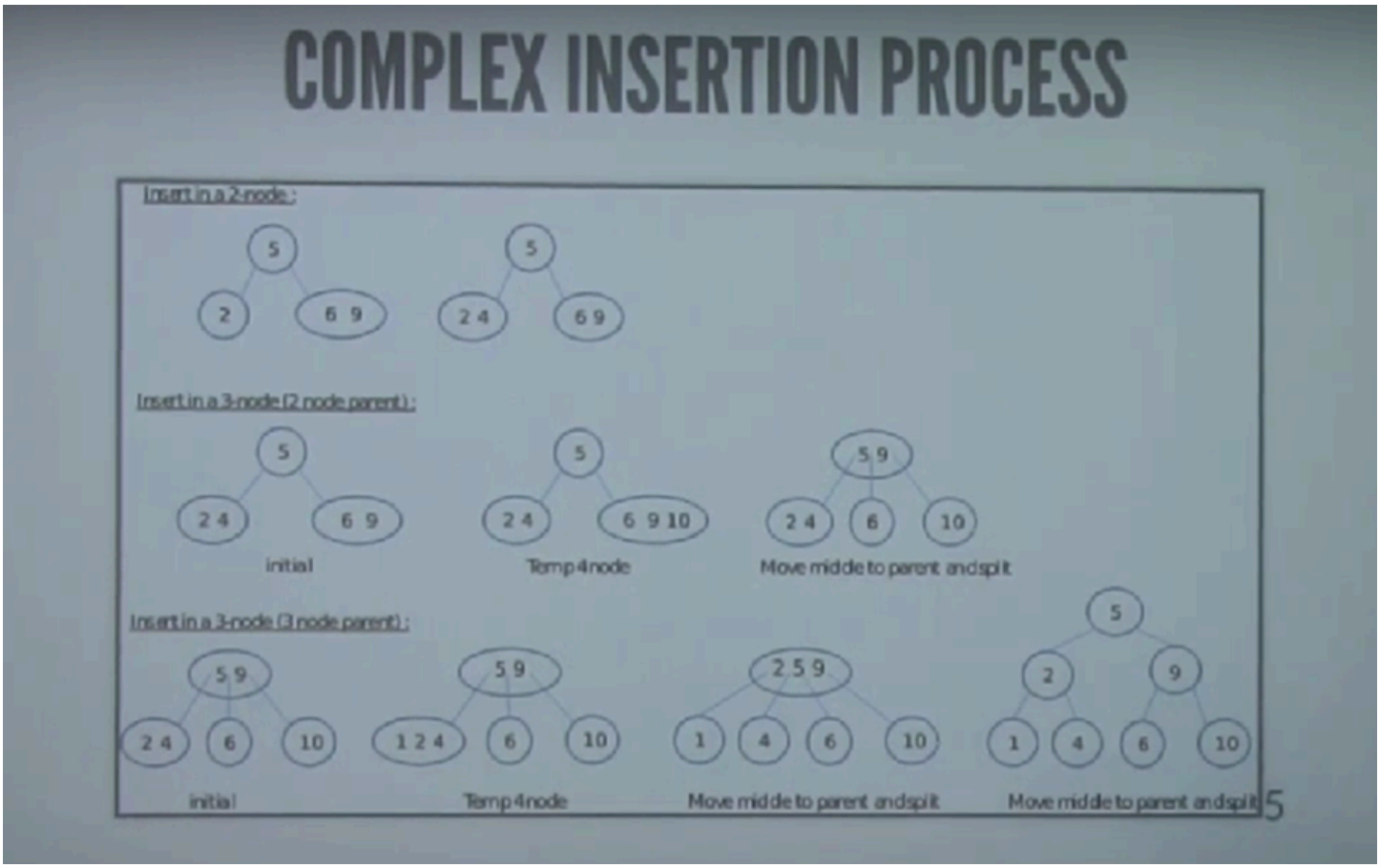
As we insert a new value, we are able to reuses subtrees

```
let tree1 = insert 10 tree
let tree2 = insert 5 tree1
let tree3 = insert 15 tree2
let tree4 = insert 7 tree3
let tree5 = insert 12 tree4
```

23 Tree




```haskell
data Tree k v = Leaf
              | Node2 (Tree k v) k v (Tree k v)
              | Node3 (Tree k v) k v (Tree k v) k v (Tree k v)
              deriving (Show, Read, Eq)
```




```haskell
insert :: (Ord k) => k -> v -> (Tree k v) -> (Tree k v)
insert k v t =
  let add k v Leaf = Pushed Leaf k v Leaf
      {- First we handle all the corner cases, when a visited node is empty -}
      add k v (Node2 Leaf k' v' Leaf)
        | k < k'    = Consumed (Node3 Leaf k v Leaf k' v' Leaf)
        | k == k'   = Consumed (Node2 Leaf k v Leaf)
        | otherwise = Consumed (Node3 Leaf k' v' Leaf k v Leaf)
      add k v (Node3 Leaf k' v' Leaf k'' v'' Leaf)
        | k < k' = Pushed (singleton k v) k' v' (singleton k'' v'')
        | k == k' = Consumed (Node3 Leaf k v Leaf k'' v'' Leaf)
        | k' < k && k < k'' = Pushed (singleton k' v') k v (singleton k'' v'')
        | k == k'' = Consumed (Node3 Leaf k' v' Leaf k v Leaf)
        | otherwise = Pushed (singleton k' v') k'' v'' (singleton k v)
      {- Typical cases, when a visited node is full -}
      add k v (Node2 l k' v' r)
        | k < k'    = case add k v l of
          Consumed newL -> Consumed (Node2 newL k' v' r)
          Pushed newL k'' v'' newR -> Consumed (Node3 newL k'' v'' newR k' v' r)
        | k == k' = Consumed (Node2 l k v r)
        | otherwise = case add k v r of
          Consumed newR -> Consumed (Node2 l k' v' newR)
          Pushed newL k'' v'' newR -> Consumed (Node3 l k' v' newL k'' v'' newR)
      add k v (Node3 l k' v' m k'' v'' r)
        | k < k'        = case add k v l of
          Consumed newL -> Consumed (Node3 newL k' v' m k'' v'' r)
          Pushed newL x y newR -> Pushed (Node2 newL x y newR) k' v' (Node2 m k'' v'' r)
        | k == k' = Consumed (Node3 l k v m k'' v'' r)
        | k' < k && k < k'' = case add k v m of
          Consumed newM -> Consumed (Node3 l k' v' newM k'' v'' r)
          Pushed newL x y newR -> Pushed (Node2 l k' v' newL) x y (Node2 newR k'' v'' r)
        | k == k'' = Consumed (Node3 l k' v' m k v r)
        | otherwise     = case add k v r of
          Consumed newR -> Consumed (Node3 l k' v' m k'' v'' newR)
          Pushed newL x y newR -> Pushed (Node2 l k' v' m) k'' v'' (Node2 newL x y newR)
    {- If the two subtrees have been pushed whole way up the tree,
       we create a new Node2 root with these subtrees as children. -}
    in case add k v t of
      Consumed newT -> newT
      Pushed newL x y newR -> Node2 newL x y newR
```

23Tree Benchmarking

Operation	Time (Mean)
Insert/Small Tree	471.3 ns
Insert/Large Tree	5.883 µs
Lookup/Small Tree	613.2 ns
Lookup/Large Tree	5.996 µs
fromList/Small List	1.489 µs
fromList/Large List	63.03 ms
toList/Small Tree	737.2 ns
toList/Large Tree	2.278 µs

Lists Benchmarking

Benchmark Group	Benchmark Name	Time (Mean)
insert	`insert at start`	819.3 ns
	`insert at middle`	1.193 µs
	`insert at end`	1.746 µs
listLookup	`lookup 2 in smallList`	377.9 ns
	`lookup 100000 in largeList`	15.68 ms

WHAT WE CAN NOTICE:

23 Tree have a more complex insertion,

But are more efficient at lookup than lists

23Tree- Python vs Haskell

The Python code uses a mutable object and modifies the tree in place, whereas the Haskell code creates new trees for every insertion due to immutability.

Haskell's style is more declarative, and its use of pattern matching makes it more concise,

After benchmarking, 23tree is better than normal linked lists for lookups even if insertion is more complex (5 micro sec VS 1.5 micro sec)

Haskell Std Libraries

Data.List:

- Sequential traversal (O(n)).
- Prepending (:) is O(1)
- Appending (++) is O(n).
- Indexing (!!) is O(n).

It is Lightweighted and has a simpler memory layout.

Data.Sequence:

- Prepending (<|) and appending (|>) O(log n).
- Indexing and updates are O(log n)

It is A FINGER TREE structure that uses more memory but reduces redundant computations.

	type	[a]	Seq a
head	:: Seq a -> a	O(1)	O(1)
tail	:: Seq a -> Seq a	O(1)	O(log n)
cons	:: a -> Seq a -> Seq a	O(1)	O(log n)
last	:: Seq a -> a	O(n)	O(1)
init	:: Seq a -> Seq a	O(n)	O(log n)
snoc	:: Seq a -> a -> Seq a	O(n)	O(log n)
(++)	:: Seq a -> Seq a -> Seq a	O(n)	O(log n)
(!!)	:: Seq a -> Int -> a	O(n)	O(log n)

link vid

ACCESSING DATA

```
benchmarking Access/List Access
time                 7.802 µs   (7.737 µs .. 7.859 µs)
                     0.999 R²   (0.999 R² .. 1.000 R²)
mean                 7.865 µs   (7.802 µs .. 7.959 µs)
std dev              273.8 ns   (198.7 ns .. 414.9 ns)
variance introduced by outliers: 43% (moderately inflated)

benchmarking Access/Seq Access
time                 149.1 ns   (147.5 ns .. 150.9 ns)
                     0.999 R²   (0.998 R² .. 0.999 R²)
mean                 149.5 ns   (148.0 ns .. 151.3 ns)
std dev              5.405 ns   (4.414 ns .. 7.226 ns)
variance introduced by outliers: 55% (severely inflated)
```

Appending DATA

```
benchmarking Append/List Append
time                    76.59 ns   (74.60 ns .. 78.35 ns)
                        0.997 R²   (0.996 R² .. 0.998 R²)
mean                    74.70 ns   (73.50 ns .. 75.91 ns)
std dev                 4.018 ns   (3.388 ns .. 4.703 ns)
variance introduced by outliers: 74% (severely inflated)

benchmarking Append/Seq Append
time                    273.7 ns   (263.4 ns .. 285.5 ns)
                        0.990 R²   (0.986 R² .. 0.994 R²)
mean                    281.7 ns   (274.6 ns .. 291.4 ns)
std dev                 27.54 ns   (20.54 ns .. 43.08 ns)
variance introduced by outliers: 90% (severely inflated)
```

Weird right? Why in our benchmark Appending in SEQ is taking longer than append in LIST?

We just said that appending should be more efficient!

=>Amortized O(log n) means that most of the time, the operation will be fast, but there are occasional, more expensive operations due to the tree rebalancing that happens

Finger Tree

A finger tree is a more specialized and optimized type of tree. It is a functional data structure used in functional programming (like Haskell) to represent sequences efficiently.

Finger trees keep fingers to both ends of the sequence to allow efficient access. These are like "shortcut" references to the first and last parts of the sequence

A finger is a point where one can access part of a data structure; in imperative languages, this is called a pointer.

data FingerTree a = Empty | Single a | Deep (FingerTree (Node a)) (Node a) (FingerTree (Node a))

Deep: A deep node with two parts:

- A left subtree (finger reference pointing to the left end),
- A middle node that holds data,
- A right subtree (finger reference pointing to the right end).

```haskell
data FingerTree a
  = Empty
  | Single a
  | Deep (FingerTree a) [a] (FingerTree a)

-- Access the first element
first :: FingerTree a -> Maybe a
first Empty = Nothing
first (Single x) = Just x
first (Deep _ (x:_) _) = Just x

-- Access the last element
last :: FingerTree a -> Maybe a
last Empty = Nothing
last (Single x) = Just x
last (Deep _ _ y) = last y
```

Example: Deep (Node [1,2]) (Node [3,4]) (Node [5,6])

Finger references: The left and right subtrees of the tree maintain fingers to the leftmost and rightmost nodes respectively.

Accessing First and Last Elements:

- First element: To get the first element, you start at the leftmost chunk (Node [1,2]) and access its leftmost element (here 1).
- Last element: To get the last element, you start at the rightmost chunk (Node [4,5]) and access its rightmost element (here 5).

Zipper

The Zipper

The **Zipper** is a way to traverse and modify a tree **without rebuilding the entire structure** every time. It achieves this by keeping track of the current position in the tree (the focus) and a trail of breadcrumbs showing how to get back to the root.

Components:

1. **Crumbs (Breadcrumbs)**: These keep track of the path taken to reach the current focus:

2. `data Crumb a = LeftCrumb a (Tree a) | RightCrumb a (Tree a) deriving (Show)`

    - `LeftCrumb` means you came from the left child of a node and stores:
        - The value of the parent node (`a`).
        - The right subtree of the parent (the part you didn't go down).
    - `RightCrumb` is similar but indicates coming from the right child.

1. **Zipper Type**:

4. `type Zipper a = (Tree a, Breadcrumbs a)`

    - The first part is the current focus (`Tree a`).
    - The second part is the trail (`Breadcrumbs a`) to retrace steps back to the root.

Tree Traversal and Modifications

The code provides operations to move around the tree, modify nodes, and attach new subtrees:

1. **Move Left (`goLeft`)**: Moves focus to the left child of the current node and updates the breadcrumbs.
2. **Move Right (`goRight`)**: Moves focus to the right child similarly.
3. **Go Back (`goUp`)**: Moves the focus back to the parent node using breadcrumbs.
4. **Modify a Node (`modify`)**: Changes the value at the current focus (e.g., applying a function like `(+1)` to increment).
5. **Attach a New Subtree (`attach`)**: Replaces the current focus with a new subtree.

Why Zipper?

The Zipper allows us to:

- Traverse a tree efficiently without rebuilding it.
- Modify parts of a tree while maintaining immutability (a key property in functional programming).

For example:

- Without a zipper, modifying a node would require reconstructing the entire path back to the root.
- With a zipper, we can focus on one node at a time, apply changes, and seamlessly move around.

REFERENCE: LearnYouAHaskell Chapter 14 Zippers

```haskell
-- Binary tree data structure
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)

-- Breadcrumbs for navigation in the tree
data Crumb a = LeftCrumb a (Tree a) | RightCrumb a (Tree a) deriving (Show)
type Breadcrumbs a = [Crumb a]

-- Zipper type combining the current focus and breadcrumbs
type Zipper a = (Tree a, Breadcrumbs a)
```

```python
lst = [1, 2, 3, 4]
zipper = ListZipper([], lst[0], lst[1:])
print(zipper)  # Left: [], Focus: 1, Right: [2, 3, 4]
zipper = zipper.move_right()
print(zipper)  # Left: [1], Focus: 2, Right: [3, 4]
zipper = zipper.update_focus(5)
print(zipper)  # Left: [1], Focus: 5, Right: [3, 4]
print(zipper.to_list())  # [1, 5, 3, 4]
```

Zippers Implementation

Operation	Time (Mean)
Zipper Operations/goLeft	135.1 ns
Zipper Operations/goRight	136.1 ns
Zipper Operations/goUp	116.5 ns
Zipper Operations/modify	131.0 ns

Binomial Heap

A binomial heap is a collection of binomial trees satisfying the min-heap property, where the key of a parent node is always less than or equal to the keys of its children. It is a type of priority queue data structure that supports efficient insertion, deletion, and finding the minimum element.

Merge (Union): Merging two binomial heaps is one of the most efficient operations and takes O(log n) time.

Binomial Tree Basics

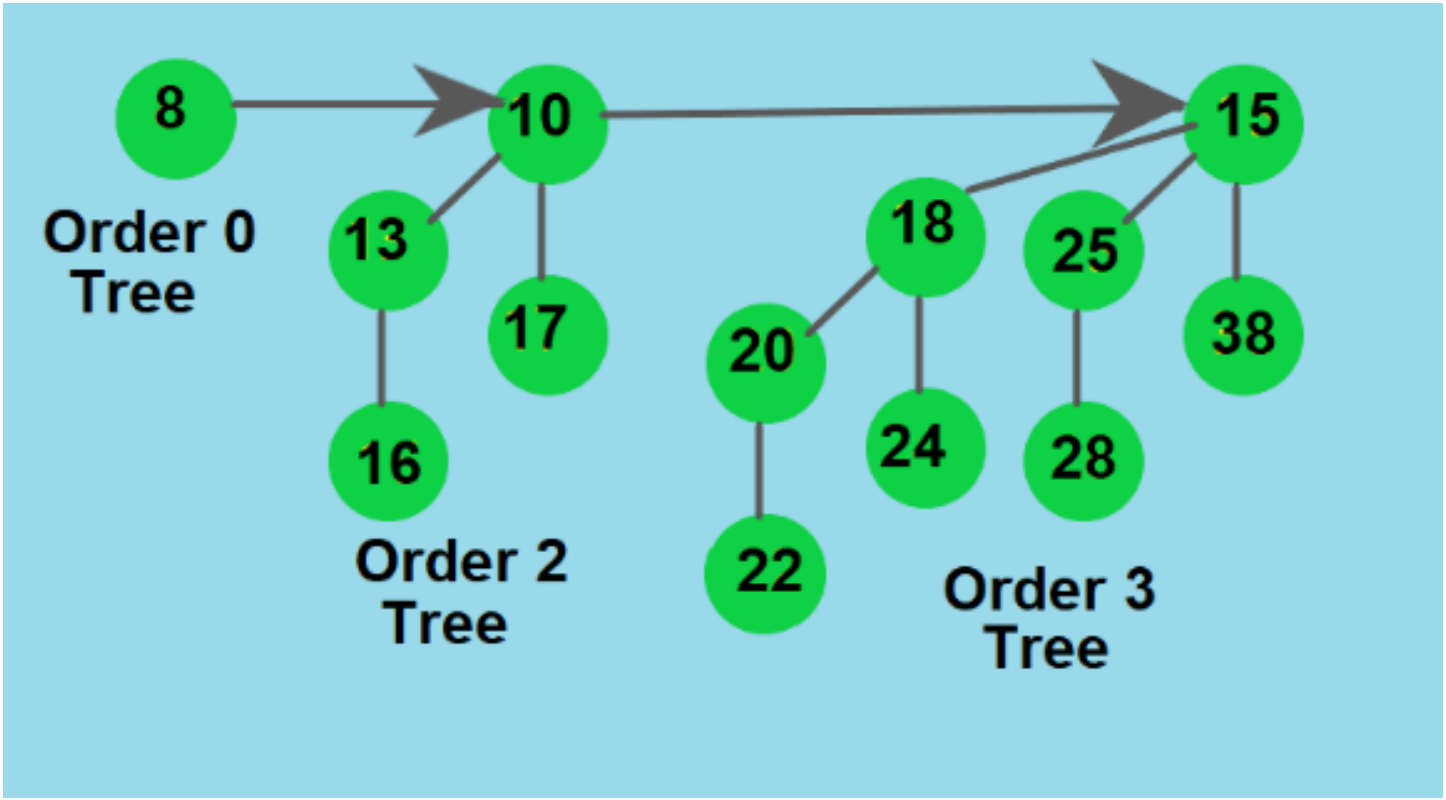
1. **Binomial Tree**:
   - A binomial tree of rank r is recursively defined:
     - A binomial tree of rank 0 is a single node.
     - A binomial tree of rank $r+1$ is formed by linking two binomial trees of rank r, where the smaller root becomes the parent.
   - Example of a binomial tree of rank 2:

1

/ \

2 3

1. **Heap Property**:
   - Every parent node has a value less than or equal to its children (min-heap).




```
benchmarking BinomialHeap Operations/insert
time                    142.4 ns   (137.2 ns .. 147.4 ns)
                        0.992 R²   (0.990 R² .. 0.997 R²)
mean                    139.7 ns   (136.4 ns .. 143.7 ns)
std dev                 11.48 ns   (9.090 ns .. 14.20 ns)
variance introduced by outliers: 87% (severely inflated)

benchmarking BinomialHeap Operations/merge
time                    99.30 ns   (97.99 ns .. 100.7 ns)
                        0.998 R²   (0.997 R² .. 0.999 R²)
mean                    100.9 ns   (99.55 ns .. 102.9 ns)
std dev                 5.272 ns   (4.386 ns .. 6.953 ns)
variance introduced by outliers: 72% (severely inflated)

benchmarking BinomialHeap Operations/deleteMin
time                    589.4 ns   (582.8 ns .. 596.6 ns)
                        0.999 R²   (0.998 R² .. 0.999 R²)
mean                    588.9 ns   (581.8 ns .. 599.0 ns)
std dev                 26.96 ns   (19.45 ns .. 35.68 ns)
variance introduced by outliers: 63% (severely inflated)
```

Use cases of binomial Heaps

- Merging Efficiency: Binomial heaps allow merging of two priority queues in O(logn)O(\log n) time, while normal priority queues typically require O(n)O(n) time for merging.
- Insertion Time: Both binomial heaps and normal priority queues offer O(logn)O(\log n) insertion time, but binomial heaps handle merging operations more efficiently.
- Memory Structure: Binomial heaps are more memory-efficient than normal priority queues when handling large numbers of elements due to their tree-based structure, while normal priority queues often use arrays or linked lists. (The memory for these trees is more compact because each tree is linked in a way that reduces redundant storage, using pointers to represent relationships between nodes.)

Code Explanation

We implemented a binomial heap in both Haskell and Python. Now, let's dive into the Haskell implementation to explain its structure and functionality.

BinomialHeap.hs

Implementation Details

Data Structures

1.Tree

data Tree a = Node Int a [Tree a]

- Represents a binomial tree.
  - Int: Rank of the tree.
  - a: Value of the root element.
  - [Tree a]: List of child trees.

2.Heap

type Heap a = [Tree a]

- Represents a binomial heap as a list of binomial trees.

3.Element Typeclass

class Ord a => Element a where
leq :: a -> a -> Bool

- Defines a custom comparison method (leq) for elements in the heap.
- Ensures flexibility in handling different data types.

Heap Operations

1.Initialization

- empty: Represents an empty heap.

  empty :: Heap a
- isEmpty: Checks whether a heap is empty.

  isEmpty :: Heap a -> Bool

2.Tree Manipulation

- rank: Retrieves the rank of a tree.
  rank :: Tree a -> Int
- root: Retrieves the root element of a tree.
  root :: Tree a -> a
- link: Combines two trees of the same rank into one, preserving the heap property.
  link :: (Element a) => Tree a -> Tree a -> Tree a

3.Insertion

- insTree: Inserts a tree into a heap, maintaining structural properties.
  insTree :: (Element a) => Tree a -> Heap a -> Heap a
- insert: Adds a new element to the heap.
  insert :: (Element a) => a -> Heap a -> Heap a

4.Merging

- merge: Combines two heaps into one while preserving the heap property.
  merge :: (Element a) => Heap a -> Heap a -> Heap a

5.Finding and Deleting the Minimum

- removeMinTree: Removes the tree with the smallest root value.
  removeMinTree :: (Element a) => Heap a -> (Tree a, Heap a)
- findMin: Retrieves the minimum element from the heap.
  findMin :: (Element a) => Heap a -> a
- deleteMin: Deletes the minimum element and rebalances the heap.
  deleteMin :: (Element a) => Heap a -> Heap a

Benchmarking

The Criterion library is used for performance measurement

Memory Profiling

The printMemoryUsage function reports memory usage using RTS statistics

Operation	*Time Complexity*
insert	$O(\log n)$
merge	$O(\log n)$
findMin	$O(\log n)$
deleteMin	$O(\log n)$

binomialHeap.py

Class Definitions

Node Class:

- Represents a node in a Binomial Tree.
- Contains attributes for:
  - data: Value of the node.
  - degree: Number of children (subtrees).
  - child: Pointer to the leftmost child.
  - parent: Pointer to the parent node.
  - sibling: Pointer to the next sibling node.

Key Functions

1. mergeBinomialTrees(b1, b2)

- Merges two Binomial Trees of the same degree.
- Ensures the smaller data tree becomes the root.
- Adjusts the parent and sibling pointers.

Objective: Fundamental operation for building a Binomial Heap.
Time Complexity: O(1).

2. unionBionomialHeap(l1, l2)

- Merges two binomial heaps to one, keeping sorted by their degree.
  Steps:
  - This function merges the two sorted lists of Binomial Trees
  - Returns combined list of trees.

Objective: preparation for adjustment in heap
Time Complexity: O(log n).

3. adjust(_heap)

- Ensures the Binomial Heap maintains its properties:
  - Increasing order of tree degrees.
  - No two trees have the same degree.
- Merges trees with the same degree using mergeBinomialTrees.

Time Complexity: $O(\log n)$.

4. insertATreeInHeap(_heap, tree)

- Adds a single Binomial Tree to the heap.
- Calls unionBionomialHeap and adjust.

Objective: Allows insertion of nodes into the heap.

5. removeMinFromTreeReturnBHeap(tree)

- Removes the root (minimum element) of a Binomial Tree.
- Converts its children into a new Binomial Heap.

Objective: extract the minimum element.
Time Complexity: $O(\log n)$.

6. insert(_head, key)

- Creates a new Binomial Tree with the key.
- Inserts it into the existing heap using insertATreeInHeap.

Time Complexity: $O(\log n)$.

7. getMin(_heap)

- Finds the node with the smallest value in the heap.
- Traverses all root nodes to find the minimum.

Time Complexity: $O(\log n)$.

8. extractMin(_heap)

- Finds and removes the minimum element in the heap.
- Merges the remaining heap with the heap formed by the children of the minimum element.
- Calls unionBionomialHeap and adjust to restore heap properties.

Objective: Extracts the smallest value in the heap.
Time Complexity: $O(\log n)$.

9. printTree(h) and printHeap(_heap)

- Prints the Binomial Tree and the entire heap.
- Recursively traverses and displays nodes.

Code Comparison with Haskell

Feature	*Python*	*Haskell*
Language Paradigm	Object-Oriented (Classes and Methods).	*Functional Programming (Pure Functions and Immutable Data).*
Implementation	*Uses classes (Node) and dynamic data structures (lists) for heap storage.*	*Defines recursive types for nodes and trees; relies on immutability and recursion.*
State Management	*Explicit tracking of parent, sibling, and child relationships through pointers.*	*Recursive pattern matching to traverse and manipulate data.*
Performance Profiling	*Profiling uses Python libraries like memory_profiler and time to track execution time and memory usage.*	*Profiling relies on GHC tools like Criterion for benchmarking; no explicit profiling in code.*
Debugging	*Python's syntax make debugging more intuitive.*	*Haskell's declarative style and lazy evaluation can make debugging challenging but concise.*

Conclusion

Purely functional data structures are ideal when immutability, safety, and concurrency are priorities.

However, they may come with performance trade-offs, particularly in memory

Where are we using FUNCTIONAL data structures nowadays?

- In environments that require high concurrency, such as multi-core processors or distributed systems (immutability)
- Blockchain and Cryptocurrencies
- Many NoSQL databases use immutable data structures to support features like distributed transactions, and versioned data.
- In real-time systems (e.g., embedded systems, robotics), where consistency, safety, and predictable behavior are critical

REFERENCES:

- LearnYouAHaskell Chapter 14 Zippers
- https://youtu.be/uMWnOkm6zIQ?si=T5-57mEGn4JUixAV video about Functional Data Structures
- https://www.bing.com/videos/riverview/relatedvideo?&q=finger+tree+data+structure&&mid=67C0565D74C25E4C83C767C0565D74C25E4C83C7&&FORM=VRDGAR Video about Haskell Standard Libraries