Welcome to Programming Fundamentals 3

University of Luxembourg

2025/2026

Bachelor in Computer Sciences (BICS)
Semester 3

Lecture - Week #8

Dr. Afonso DELERUE ARRIAGA afonso.delerue@uni.lu

Important: Please remember to register your attendance in Moodle!

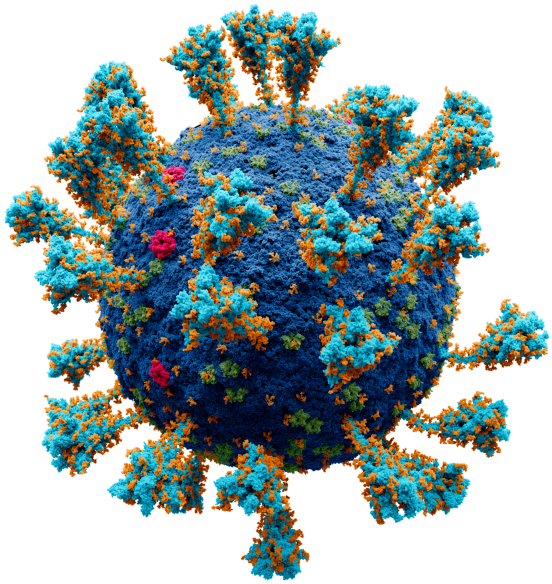
Intro

In a post-apocalyptic world, you're tasked with identifying an infected individual. 🦠

1000 survivors, 1 might be the carrier of a deadly virus.

You dispose of a state-of-the-art equipment capable of detecting the virus in blood samples, but it can only process one sample at the time. Results are 100% accurate and take 24 hours to materialize.

How long to find the carrier?



Whiteboard

WELCOME TO PF3 WEEK #8

SEARCH OVER A SORTED ARRAY

LIST = [2 , 3 , 10 , 17 , 23 , 99]

(10: MIDDLE; 23: FOUND)

L = POS 0

R = POS LENGTH LIST - 1

is 23 ∈ LIST ?

LIST !! 0 = 2

④ AT THE BEGINNING WE HAVE THE WHOLE ARRAY TO SEARCH. WE SET L = 0 AND R = LEN - 1

② FIND THE MIDDLE ELEMENT THAT SEATS BETWEEN L AND R.

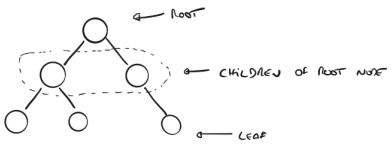
③ IF x > MIDDLE : SEARCH BETWEEN MIDDLE + 1 AND R

IF x < MIDDLE : SEARCH BETWEEN L AND MIDDLE - 1

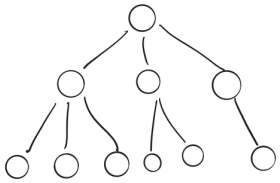
IF x == MIDDLE : FOUND IT!

IF L > R : x IS NOT IN LIST.

TREES

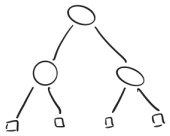


ROSE TREES



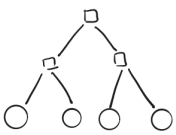
```
DATA TREE a = NODE a [TREE a]
```

BINARY TREE WITH LABELED NODES



```
DATA TREE a = LEAF | NODE (TREE a) (TREE a)
```

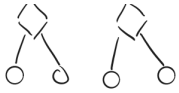
BINARY TREE WITH LABELED LEAVES

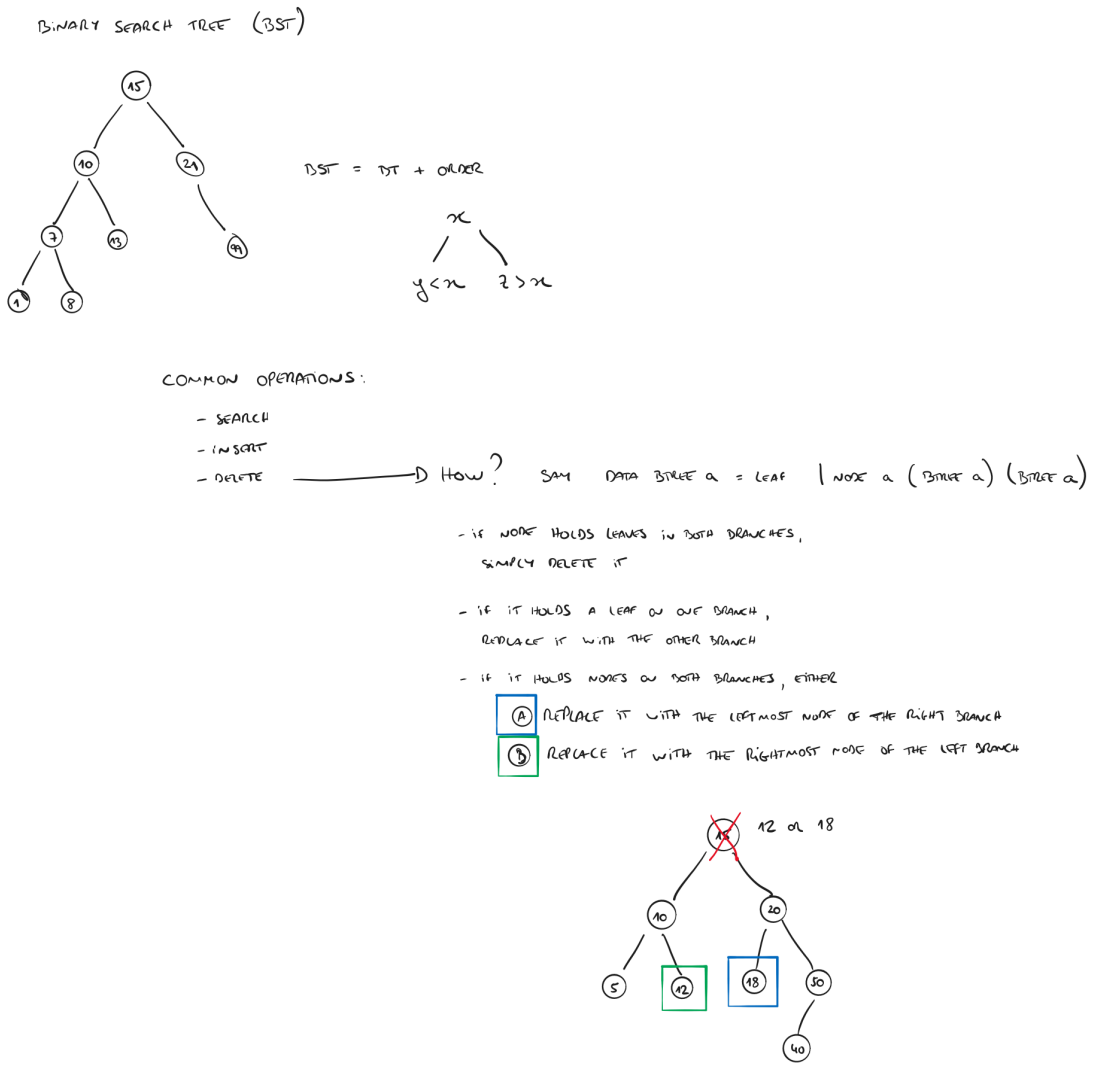


```
DATA TREE a = LEAF a | NODE (TREE a) (TREE a)
```

BINARY TREE WITH LABELED NODES AND LEAVES

```
DATA TREE a b = LEAF a | NODE b (TREE a b) (TREE a b)
```





Warning: Please note that the code presented in these slides was created during the class and has not undergone extensive testing. If you come across any issues or bugs, please report them.

Binary search over sorted lists

```
search :: Ord a => a -> [a] -> Bool
search x xs = binarySearch x xs 0 (length xs -1)

binarySearch :: Ord a => a -> [a] -> Int -> Int -> Bool
binarySearch x list l r | l > r = False
                        | x > list !! middle = binarySearch x
list (middle+1) r
                        | x < list !! middle = binarySearch x
list l (middle-1)
                        | otherwise = True
```

```
  where
    middle = (r+l) `div` 2
```

- **It's important to note that random access to a position in an array is a common and efficient operation in most imperative programming languages, but in Haskell, (!!) is inefficient because it has to traverse the list linearly. Using !! operator for binary search would not provide the expected efficiency gains of a true binary search algorithm.**
- If you want to perform a binary search efficiently in Haskell, you should use a different data structure that allows for logarithmic-time random access, such as an array or Data.Array, rather than a simple Haskell list.
- Arrays are simple data structures but arrays are also rather static.
- In a sorted array, inserting or deleting an element often requires shifting all elements to maintain the sorted order, which can be inefficient.
- Binary search trees are dynamic data structures, which means you can easily insert and delete elements without the need to reorganize the entire structure.
- Binary trees are also friendly data structures for functional programming languages like Haskell.

Trees 🌳

- Trees are common data structures, widely used in computer science.
- A picture is worth a 1000 words (see whiteboard). Root node has child nodes, and those child nodes have child nodes of their own, and so on. If a node has no children, it's called a *leaf*.
- Trees are sets of connected nodes, without loops.
- Trees come in multiple forms. We've seen a few definitions back in Lecture 5:

```
-- trees may have many branches, these trees are called rose
trees
data Tree a = Node a [Tree a]

-- binary trees with unlabelled nodes
data Tree a = Leaf a | Node (Tree a) (Tree a)

-- binary trees with labelled nodes but unlabbeled leaves
data Tree a = Leaf | Node (Tree a) a (Tree a)

-- binary trees with leaves and nodes possibly labelled with
different types
data Tree a b = Leaf a | Node (Tree a b) b (Tree a b)
```

- Very often, when we talk about trees, we mean *binary trees*. A binary tree means that each node has 2 child nodes (or leaves).
- Take this alternative representation of a binary tree with labelled nodes and unlabbeled leaves. Each node has a *left* tree and a *right* tree. Leaves carry no data, so we just use

the constructor with no argument. It's common to leaf nodes `Empty`, `Nil` or `Leaf`.

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

- Trees have many applications in Computer Science:
  - File systems (and directory structure);
  - Document Object Models, such as HTML and XML;
  - Hierarchical organizational structures;
  - Priority queues (Heaps);
  - Search (BST);
  - etc.

Binary Search Trees (BST)

- To be a binary search tree, for every node in the BST:
  1. Its left child node has value lesser (`<`)than the parent node;
  2. Its right child node has value greater (`>`) than the parent node;
  3. Both its children are BST trees.
- Finding a node in a BST is very eficient because we can quickly digure out where the node might be.
- Common operations:
  1. `search`
  2. `insert`
  3. `delete`

```
data BTree a = Leaf | Node a (BTree a) (BTree a)
  deriving (Eq,Show)

search :: Ord a => a -> BTree a -> Bool
search x Leaf = False
search x (Node y left right)
    | x < y = search x left
    | x > y = search x right
    | otherwise = True

insert :: Ord a => a -> BTree a -> BTree a
insert x Leaf = Node x Leaf Leaf
insert x t@(Node y left right)
    | x < y = Node y (insert x left) right
    | x > y = Node y left (insert x right)
    | otherwise = t

delete :: Ord a => a -> BTree a -> BTree a
delete x Leaf = error "cannot delete element" -- in the
definition of BTree a, leaves are not labeled
delete x (Node y left right)
    | x < y = Node y (delete x left) right
    | x > y = Node y left (delete x right)
```

```
    | left == Leaf = right -- at this point we have that x == y
    | right == Leaf = left
    | otherwise = let z = leftmost right -- alternative, fetch
the rightmost node of the left branch
                  in Node z left (delete z right)
  where
    leftmost :: BTree a -> a
    leftmost Leaf = error "cannot be here"
    leftmost (Node n Leaf _) = n
    leftmost (Node n l r) = leftmost l
```

- How to delete a node?
  - If it's a leaf, simply delete it.
  - If it's a node without a left branch or a right branch, replace the node with its non-empty branch.
  - If it's a node with both branches, replace the node with either:
    1. the rightmost node of the left branch, or
    2. the leftmost node of the right branch.
- If the BST is fully filled, search operations takes `O(log n)` steps, where `n` is the number of nodes in the tree.
- However, if the BST is unbalanced, search takes `O(n)`.
- Try inserting some random nodes:

```
t1 :: BTree Int
t1 = foldl (flip insert) Leaf [5,4,3,7,1,10,8]
```

- See what happens if nodes are inserted sorted:

```
t2 :: BTree Int
t2 = foldl (flip insert) Leaf [1..10]
```

Tree traversals

There are 3 common ways to visit every in a tree:

1. `inorder`: left-root-right, which produces an ordered list of the nodes
2. `preorder`: root-left-right
3. `postorder`: left-right-root

For a Binary Search Tree (BST), the `inorder` traversal visits nodes *in order*. Storing the nodes of an inorder traversal in a list results in a sorted list.

Write the 3 traversal algorithms.

```
-- tree traversals
-- inorder (left, root, right)
inorder :: BTree a -> [a]
inorder Leaf = []
```

```
inorder (Node x left right) = inorder left ++ [x] ++ inorder
right

-- preorder traversal (root, left, right)
preorder :: BTree a -> [a]
preorder Leaf = []
preorder (Node x left right) = [x] ++ preorder left ++ preorder
right

-- postorder traversal (left, right, root)
postorder :: BTree a -> [a]
postorder Leaf = []
postorder (Node x left right) = postorder left ++ postorder
right ++ [x]
```