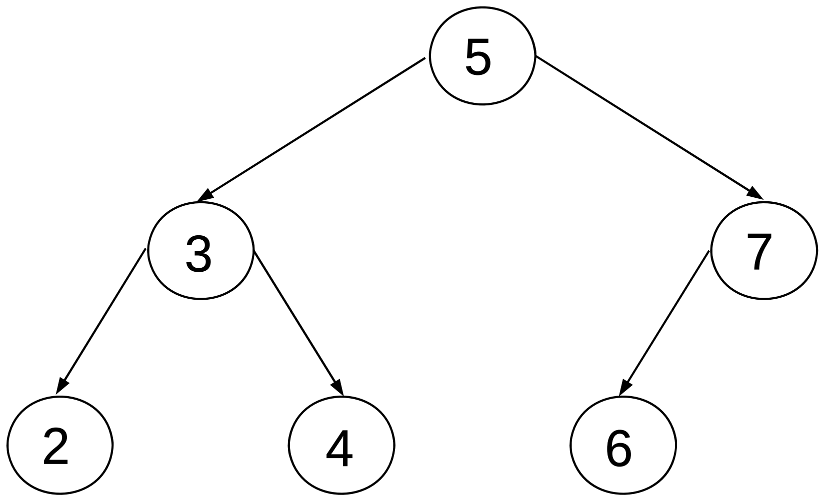
Tree Structures

Binary Search Trees (BST)

Binary Search Trees (BST)

- **Definition**: A **BT** where for each node **N**, the following invariant applies:

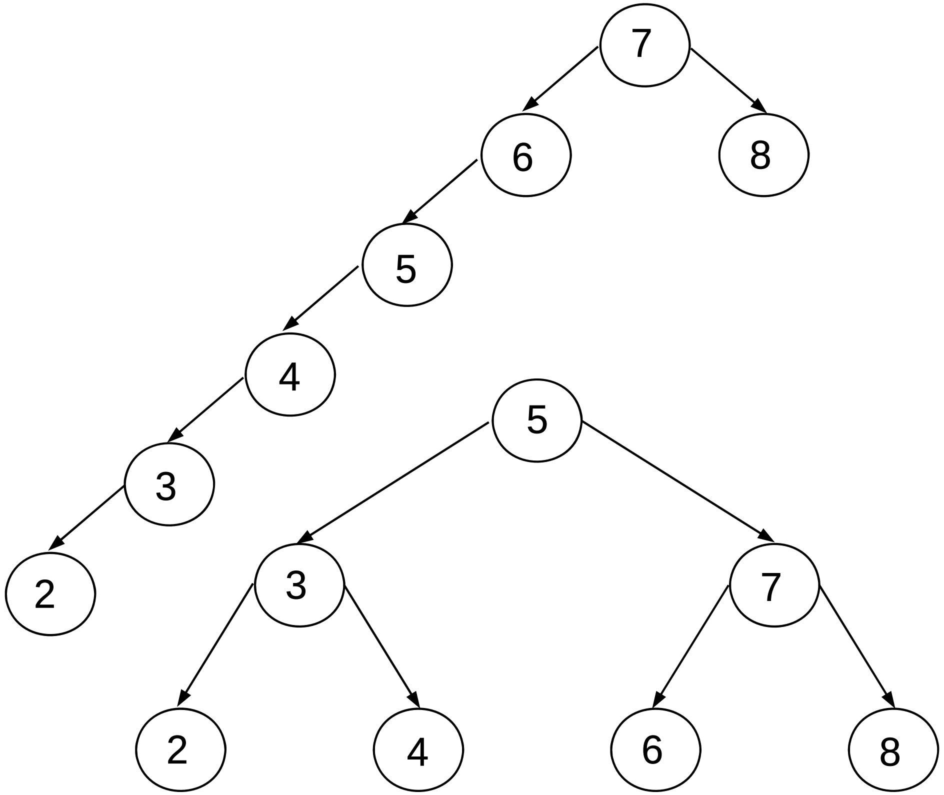
Value(Left(N))
< Value(N)
< Value(Right(N))



- The invariant applies before and after each operation.
- The invariant implies that a **BST is sorted.**

BST: Properties

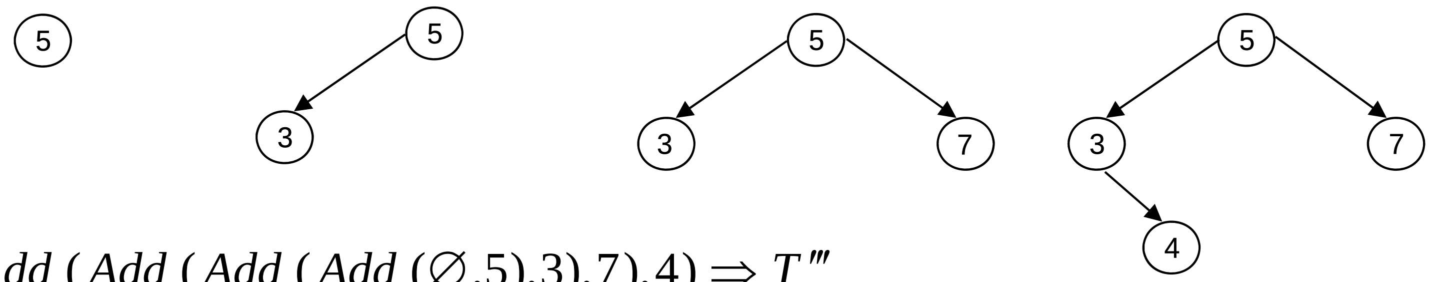
- Big-Oh Add / remove
  O(log(n)) or O(n)
  - The best case is where the tree is well balanced **O(log(n))**
  - The worst case is **O(n)**
  - **Searching** in a BST is fast – in a well balanced tree, searching is **O(log(n))**



BST – Add (NB the pattern for recursion ¤, <, >, =)

```
BST: Add(BST T, int v)
if   IsEmpty(T) then        return create_el(v)
if v < value(T) then        return cons(Add(left(T), v), T, right(T))
if v > value(T) then        return cons(left(T), T,  Add(right(T), v))
/* v =  value(T)  */        return T        // no duplicates!
end Add
```

$Add\,(\varnothing,5) \Rightarrow T$ $\quad Add\,(T,3) \Rightarrow T'$ $\quad Add\,(T',7) \Rightarrow T''$ $\quad Add\,(T'',4) \Rightarrow T'''$



$$Add\,(Add\,(Add\,(Add\,(\varnothing,5),3),7),4) \Rightarrow T'''$$

BST – Remove (NB the pattern for recursion ¤ < > =)

```
BST remove(BST T,  int v)
{
  if  IsEmpty(T) then                return T
  if v < value(T) then               return cons(remove(LC(T), v), T,  RC(T))
  if v > value(T) then               return cons(LC(T), T,  remove(RC(T), v))
  /* v =  value(T)  */               return remove_Root(T);
  }
```

LC, RC = return left and right child respectively

remove_Root(T) = return a BST with the (local) root removed

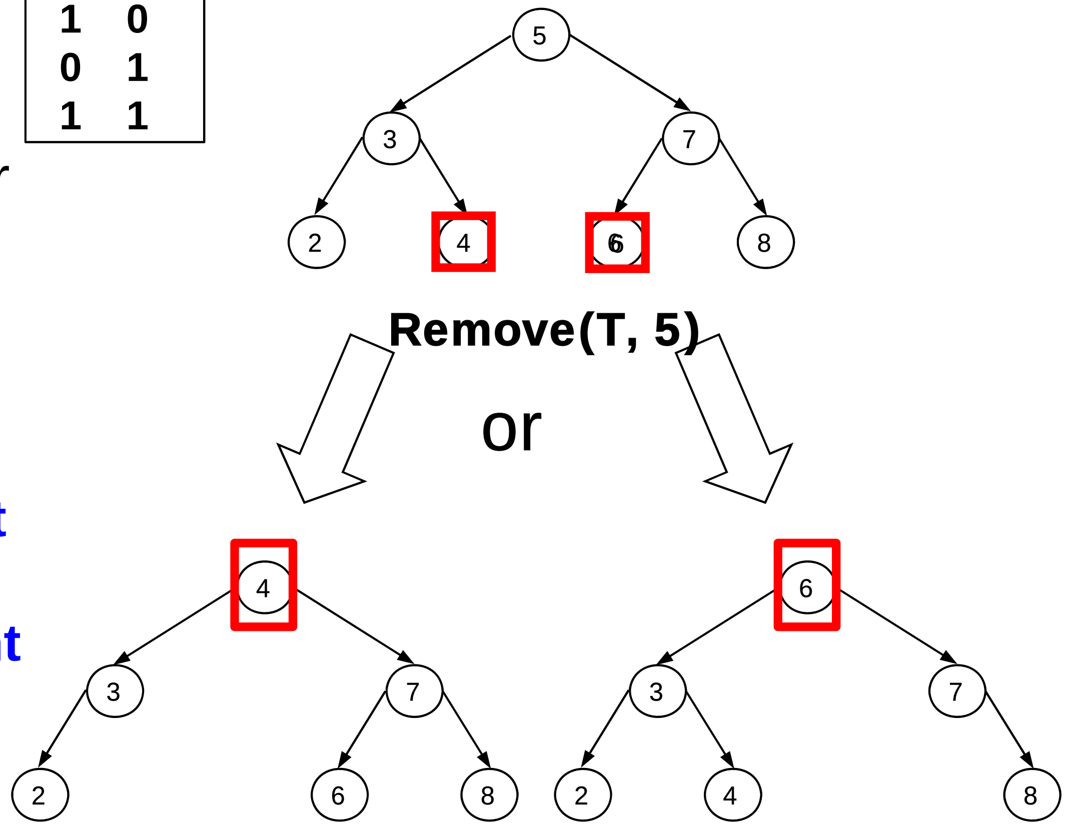
Divide the problem into the simpler cases first (¤ < >) then the more complicated case at the end (v = value(T) i.e. value of the root)

BST – Operations: remove root

LC	RC
0	0
1	0
0	1
1	1

Remove root – 4 cases

- If **leaf** – remove! **(0 0)**
- If the **left- or right sub-tree is empty** the (non-empty) right or left sub-tree is the result **(1 0) & (0 1)**
- T is **(LC N RC)** **(1 1)**
- N exchanged with either
  - **The highest value in the left sub-tree**
  - **The lowest value in the right sub-tree**



BST – Operations: remove root

```
BST remove_Root(BST T)
{
  ???        ☺                    // see description above (slide 6)
}
```

What is required?

- **If T is a leaf node remove the node –** **otherwise…**
- **If LC empty return RC OR RC empty return LC** **otherwise…**

LC & RC exist

- **find the maximum value in LC(T)** **OR**
- **find the minimum value in RC(T)**
- **Make the minimum/maximum value the root of a new tree – return**
- **Balance??? Think about this!**

Summary

- Properties
  - BST is a Binary tree with restrictions
  - **Value(Left(T)) < Value(T) < Value(Right(T))**
    - The invariant means that InOrder(T) is a sorted sequence
  - The operations must always maintain the invariant
    - Especially Remove:
      - When both left and right sub-trees exist, a particular strategy is required – the programmer has to choose.
      - The node to be removed is replaced by
        - The minimum value in the right sub-tree
        - The maximum value in the left sub-tree
        - **Or a recursive method may be used**