Data Structures

Week 15 – Binary search tree, balance, heap; summary of the course

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

Review

- **Binary tree:** at most two children; linked and array representation
- **Traversals:** preorder, inorder, postorder (recursively), level order (with a queue)
- **Binary search** (week 2): O(log n) in a sorted array – but insertion is O(n)

Questions for today

- How can we make both search **and** insertion fast?
- Why does it matter that the tree is **balanced**?
- How does the **heap** give a fast priority queue?

Binary search tree

The binary search tree

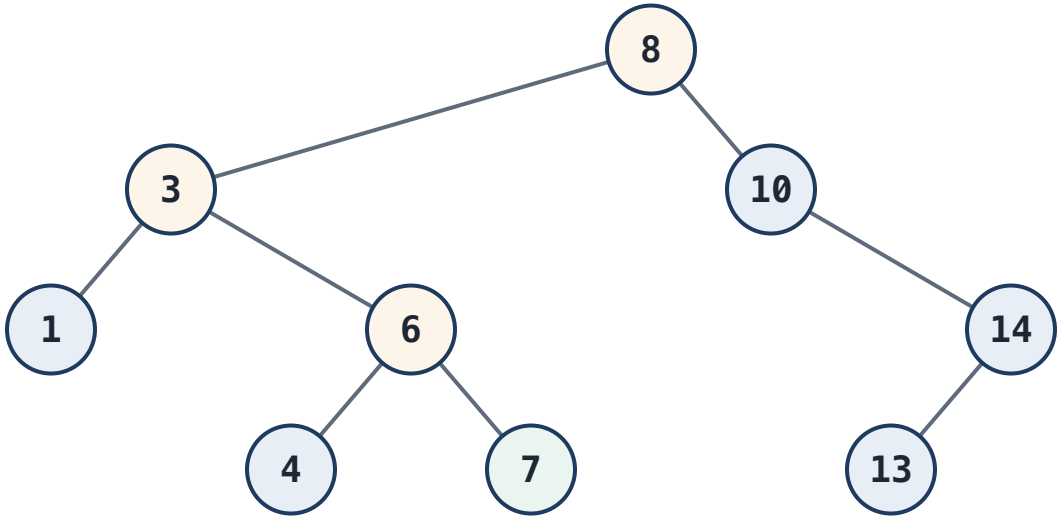
Binary search tree (BST): a binary tree in which for **every** node:

- every key in its **left** subtree is **smaller**,
- every key in its **right** subtree is **larger** than the key of the node.

Important consequence: the **inorder traversal** gives the keys in **increasing order**!
Example: after inserting 8, 3, 10, 1, 6, 14, 4, 7, 13 the inorder is: **1 3 4 6 7 8 10 13 14**

It is not enough that every node is larger/smaller than its **direct** children – the condition applies to the **whole** subtrees!

Search



Search: x = 7

```
7 < 8 → left
7 > 3 → right
7 > 6 → right
7 = 7 → found ✓
```

Every step goes one level down:

steps = O(height)

Search – algorithm

Iteratively

```
Function Search(Root, x)
  P ← Root
  While P ≠ Nil and Elem[P].Key ≠ x
    If x < Elem[P].Key then
      P ← Elem[P].Left
    else
      P ← Elem[P].Right
    End if
  End while
  Return P              // Nil if not found
End function
```

Minimum: always left, as long as possible

Maximum: always right, as long as possible

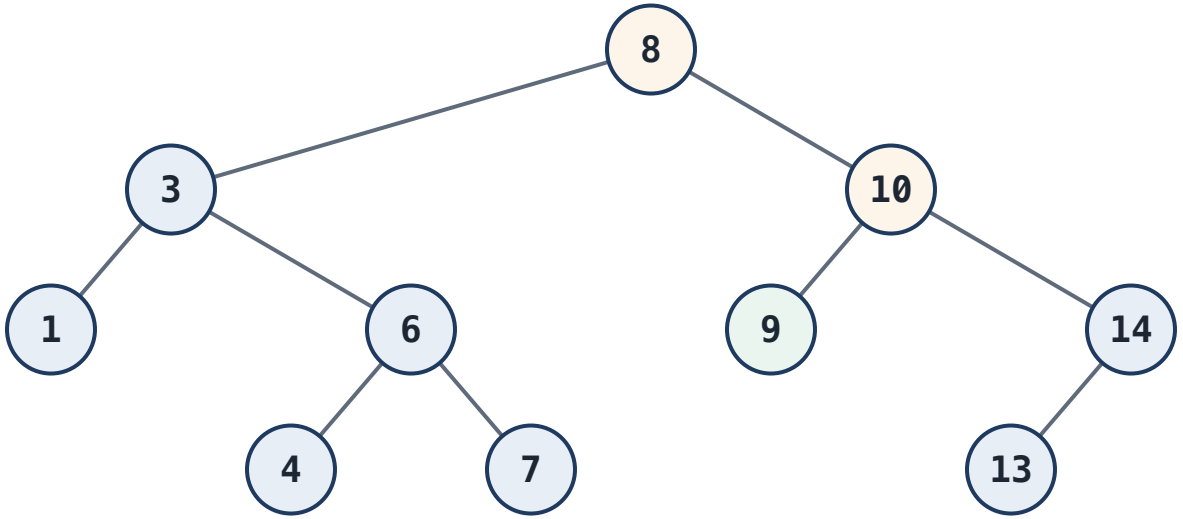
Java

```java
boolean contains(int x) {
    Node p = root;
    while (p != null && p.key != x)
        p = (x < p.key) ? p.left : p.right;
    return p != null;
}
```

Go

```go
func (t *BST) Contains(x int) bool {
    p := t.Root
    for p != nil && p.Key != x {
        if x < p.Key {
            p = p.Left
        } else {
            p = p.Right
        }
    }
    return p != nil
}
```

Insertion



Insert: x = 9

```
9 > 8 → right
9 < 10 → left
empty → here (new leaf)
```

The new element is always a leaf.

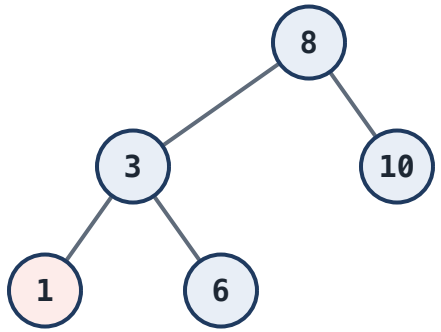
Insertion – algorithm

```
Function Insert(P, x)        // returns the new subtree
  If P = Nil then
    Reserve(Q) ; Elem[Q].Key ← x
    Elem[Q].Left ← Nil ; Elem[Q].Right ← Nil
    Return Q
  If x < Elem[P].Key then
    Elem[P].Left ← Insert(Elem[P].Left, x)
  else if x > Elem[P].Key then
    Elem[P].Right ← Insert(Elem[P].Right, x)
  Return P
End function
```

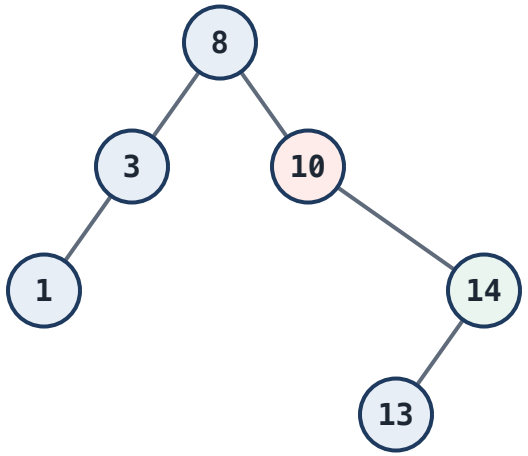
```
Node insert(Node p, int x) {
    if (p == null) return new Node(x);
    if (x < p.key)
        p.left = insert(p.left, x);
    else if (x > p.key)
        p.right = insert(p.right, x);
    return p;
}
```

Call: `root = insert(root, x);`

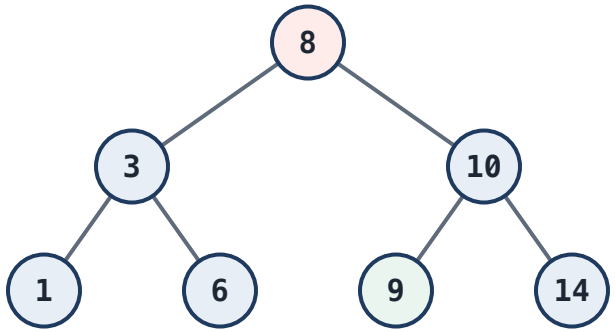
Deletion – three cases



1. leaf:
just remove it



2. one child:
the child takes its place



3. two children:
copy the in-order successor

- **Leaf:** simply deleted
- **One child:** the child (with its whole subtree) takes the place of the deleted node
- **Two children:** the **smallest** element of the right subtree (the **inorder successor**) is copied into its place, then that one is deleted – it has at most one child

Deletion – code

Java

```java
Node delete(Node p, int x) {
    if (p == null) return null;
    if (x < p.key) p.left = delete(p.left, x);
    else if (x > p.key) p.right = delete(p.right, x);
    else {
        if (p.left == null) return p.right;   // 0 or 1 child
        if (p.right == null) return p.left;
        Node s = p.right;                     // 2 children:
        while (s.left != null) s = s.left;    // inorder successor
        p.key = s.key;
        p.right = delete(p.right, s.key);
    }
    return p;
}
```

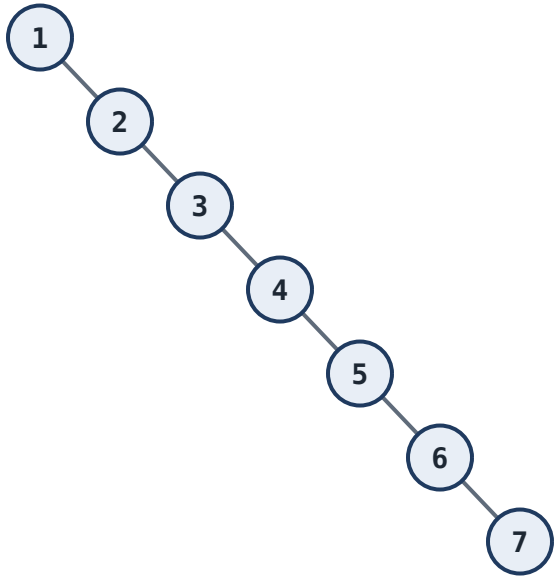
Go – the case of two children

```go
default:
    if p.Left == nil {
        return p.Right
    }
    if p.Right == nil {
        return p.Left
    }
    s := p.Right              // inorder successor
    for s.Left != nil {
        s = s.Left
    }
    p.Key = s.Key
    p.Right = del(p.Right, s.Key)
```

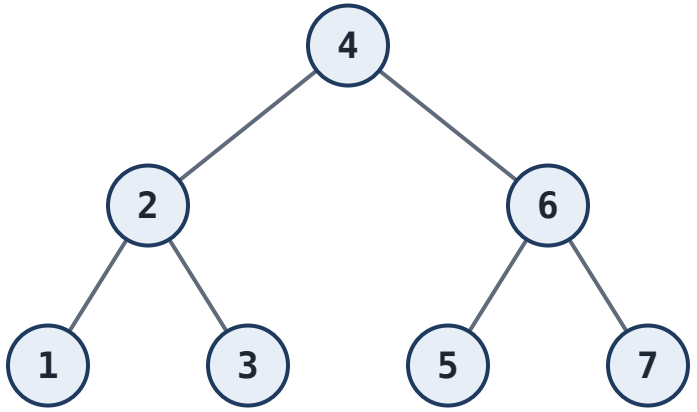
The full code: `w15/bst`

Balance

The efficiency of a search tree depends on its height



inserted as 1, 2, …, 7: height 6

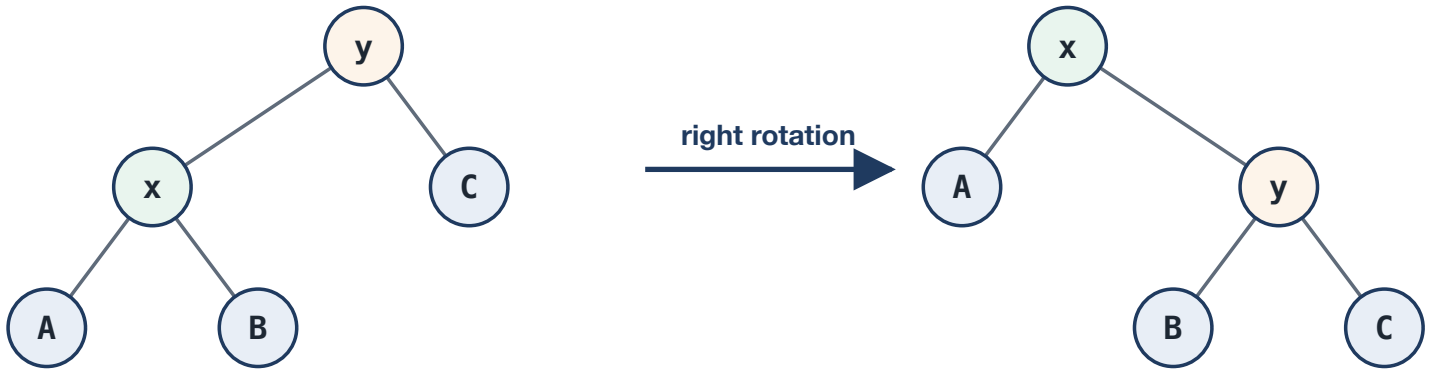


same elements, balanced: height 2

- Search, insertion, deletion: **O(h)**, where **h** is the height of the tree
- **Balanced** tree: $h \approx \log_2 n \rightarrow$ **O(log n)** · **degenerate** tree: $h = n - 1 \rightarrow$ **O(n)** (a list!)
- **Measurement:** 1000 elements in sorted order → height **999** · in random order → height **18**

Balanced search trees

Balanced tree: in every node the heights of the left and the right subtree differ **by at most 1** (AVL condition).



A < x < B < y < C – the in-order sequence (BST property) is preserved

- After insertion and deletion the tree is restored with **rotations** – one rotation is O(1)
- **AVL tree** (1962), **red-black tree** (Java `TreeMap`, `TreeSet`), **B-tree** (databases, file systems)
- Guaranteed **O(log n)** search, insertion, deletion

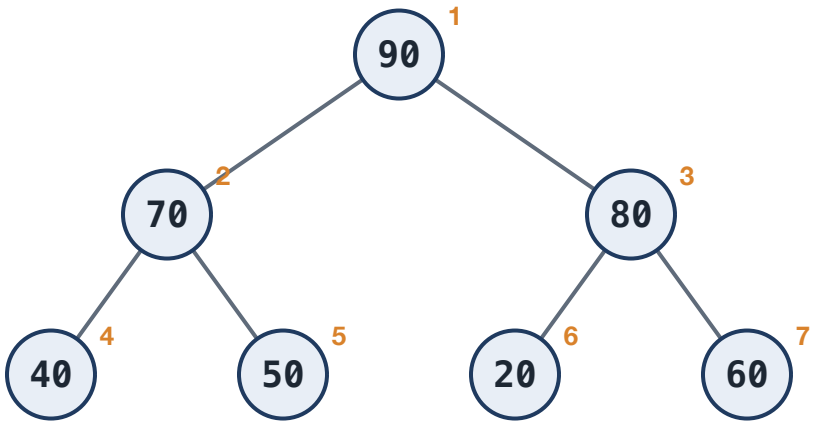
Search tree, sorted array or hash table?

Operation	Sorted array	Hash table	Balanced search tree
Search	O(log n)	**O(1)** on average	O(log n)
Insertion, deletion	O(n)	**O(1)** on average	O(log n)
Minimum, maximum	**O(1)**	O(n)	O(log n)
Sorted traversal	O(n)	O(n log n)	O(n)
Range [a, b]	O(log n + k)	O(n)	O(log n + k)

If we only need to search: **hash table**. If the **order** matters too (smallest, next, range): **search tree**.

Heap

The heap



A[1..7] – complete tree in an array (week 14)

90	70	80	40	50	20	60
1	2	3	4	5	6	7

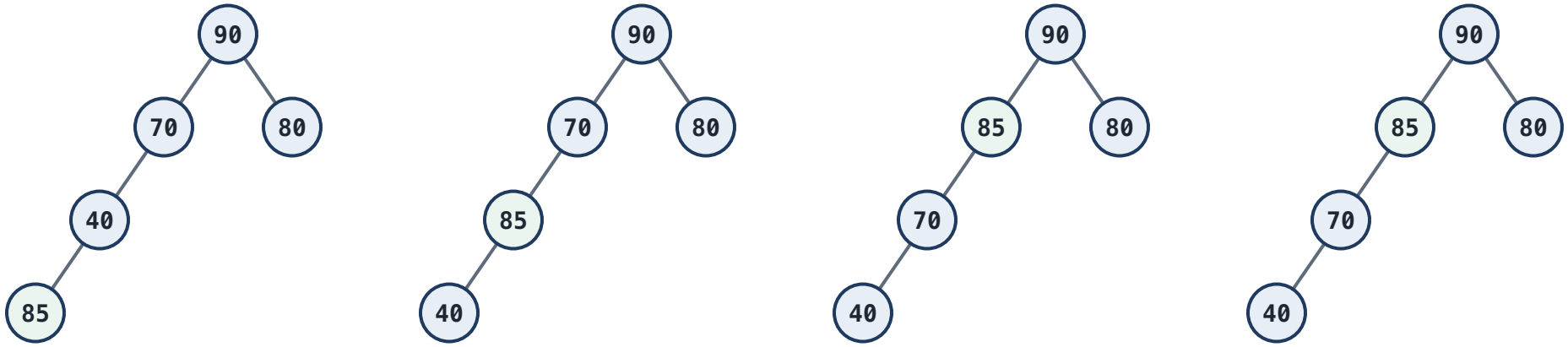
Max-heap: every parent ≥ its children

`A[i] ≥ A[2i] and A[i] ≥ A[2i + 1]`

the maximum is always at the root: A[1]

- A **complete** binary tree, stored **in an array** (week 14: children 2i and 2i + 1, parent i div 2)
- **Heap property:** every parent ≥ its children (max-heap) – or ≤ (min-heap)
- There is **no** order between siblings – a heap is **not** a search tree!

Insertion: sift up



85 at the end | 85 > 40 → swap | 85 > 70 → swap | 85 < 90 → done

- The new element goes to the **end** of the array (the next free place of the tree)
- While it is larger than its parent, we **swap** them → at most **h** = $\log_2 n$ swaps → **O(log n)**

Removing the maximum: sift down

```
Function RemoveMax(A, n)
  max ← A[1]
  A[1] ← A[n] ; n ← n − 1
  i ← 1
  While 2·i ≤ n
    c ← 2·i                          // left child
    If c + 1 ≤ n and A[c + 1] > A[c] then
      c ← c + 1                      // the larger child
    If A[i] ≥ A[c] then exit
    swap(A[i], A[c]) ; i ← c
  End while
  Return max
End function
```

1. The **maximum** is in the root – this is returned
2. The **last** element is put into the root
3. While it is smaller than its larger child, we **swap it downwards**

O(log n)

Priority queue	Insert	Remove max
unsorted array	O(1)	O(n)
sorted array	O(n)	O(1)
heap	**O(log n)**	**O(log n)**

Java `PriorityQueue` · Go `container/heap`

Heapsort

1. **Building the heap:** build a max-heap from the array in place – O(n)
2. Repeat **n** – 1 times: **swap** the root (the maximum) **with the last element**, consider the heap one smaller, and **sift down** the root

```
[40, 70, 20, 90, 50, 80, 60, 10]   start
[90, 70, 80, 40, 50, 20, 60, 10]   heap
[10, 20, 40, 50, 60, 70, 80, 90]   sorted
```

Property	Heapsort
Steps	**O(n log n)** – in every case
Extra space	**O(1)** – sorts in place
Stable	no

Summary of the course

Data structures – overview

Data structure	Access	Search	Insertion	Deletion	Week
array	**O(1)** by index	O(n) / O(log n) sorted	O(n)	O(n)	3
dynamic array	O(1)	O(n)	O(1)* at the end	O(n)	3
linked list	O(n)	O(n)	**O(1)** known place	**O(1)** known place	9–10
stack, queue	only the end(s)	–	O(1)	O(1)	7, 8, 11
set (boolean vector)	–	O(1)	O(1)	O(1)	6
hash table	–	**O(1)** avg.	**O(1)** avg.	**O(1)** avg.	11
search tree (balanced)	–	O(log n)	O(log n)	O(log n)	15
heap	max: O(1)	O(n)	O(log n)	max: O(log n)	15

* amortized

Algorithms – overview

Problem	Algorithm	Steps
search	linear · sentinel · binary	O(n) · O(n) · O(log n)
sorting (simple)	bubble · selection · insertion	$O(n^2)$
sorting (efficient)	merge · quick · heap	O(n log n)
set operations	merging sorted arrays	O(n + m)
expressions	postfix evaluation with a stack	O(n)
pattern search	naive	O(n · m)
tree traversal	pre-, in-, postorder · level order	O(n)

Preparing for week 16

Final test (60 minutes, on paper)

- True/false and multiple-choice questions about the concepts and efficiency
- Code tracing: what does a stack or a queue print?
- Completing pseudocode (missing expressions, relational operators): list, sentinel list and search tree operations
- Algorithms to execute on paper: sorting, binary search, traversal, search tree insertion

Practical grade (on the computer)

- Implementing and using a data structure in **Java or Go**

> Review the **review questions** and **exercises** of every week – the test is similar to them!

Summary

- **Binary search tree:** left < root < right; the inorder traversal is sorted
- Search, insertion, deletion **O(h) –** three cases for deletion (leaf, one child, two children)
- **Balanced** tree: h = O(log n); sorted insertion → degenerate tree
- AVL, red-black tree: keep the tree balanced with **rotations**
- **Heap:** complete tree in an array, parent ≥ child; insertion and removing the maximum O(log n)
- **Heapsort:** O(n log n), in place

Next week: final test and practical grade. Good luck!

Review questions

1. Insert into an empty search tree: 50, 30, 70, 20, 40, 60, 80, 35! Draw it and give the inorder traversal!
2. Delete 30 from the tree of question 1! What does the tree look like?
3. True or false: a binary search tree is always a balanced binary tree.
4. True or false: searching in a binary search tree can be as slow as in a linked list.
5. Insert into an empty max-heap: 15, 40, 30, 50, 10! Give the array after every step!
6. Why is a heap not suitable for quickly finding an arbitrary element?

Practice

✎ Demo programs

Program	Java	Go	What does it show?
Search tree	`BST.java`	`w15/bst`	search, insertion, the three cases of deletion, degenerate tree
Heap	`MaxHeap.java`	`w15/maxheap`	sift up, sift down, built-in priority queue
Heapsort	`HeapSort.java`	`w15/heapsort`	building the heap, sorting, timing

✎ Exercises

1. **Is it a search tree?** Decide whether a binary tree is a search tree! (Careful: checking the direct children is not enough!)
2. **k-th smallest:** find the **k**-th smallest key of a search tree!
3. **Range:** print the keys between a and b so that you enter only the necessary subtrees!
4. **Common ancestor:** find the lowest common ancestor of two keys in the search tree in O(h) steps!
5. **Floor:** find the largest key that is ≤ **x**!
6. **Top k:** find the **k** largest elements of an array with a min-heap of size **k**! What is the number of steps?

Solutions: `Exercises15.java`, `w15/exercises`

In the second half of the practice: **practice exercises for the final test** (based on the review questions of the weeks).

✎ Homework (preparation)

1. Extend the `BST` class with a `successor(x)` method that returns the smallest key larger than **x**!
2. Implement a **phone book** with a search tree (name → number) that can also print the entries in alphabetical order!
3. Implement the **min-heap**, and use it for a simple **task scheduler** (the task with the earliest deadline comes first)!
4. Review the review questions of all weeks, and solve the sample tasks of the old test on paper!