Informatics II

Tutorial Session 9

Wednesday, 27th of April 2022

Discussion of Exercises 7 and 8,
Stacks, Queues, Trees, Binary Search Trees

14.00 – 15.45

BIN 0.B.06

Agenda

- Review of Exercise 7
  - Stacks
  - Queues
- Review of Exercise 8
  - Binary Search Trees

Universität Zürich[UZH]

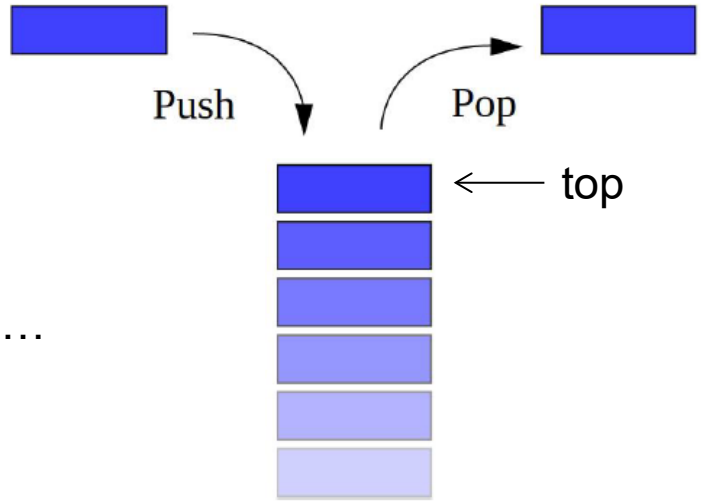
Institut für Informatik

Stacks and Queues

Overview: Stacks and Queues

Stack	Queue
	
LIFO	FIFO
– push(x) – pop()	– enqueue(x) – dequeue()

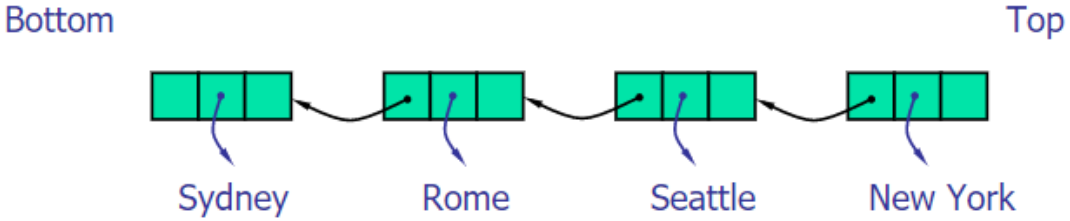
Stacks



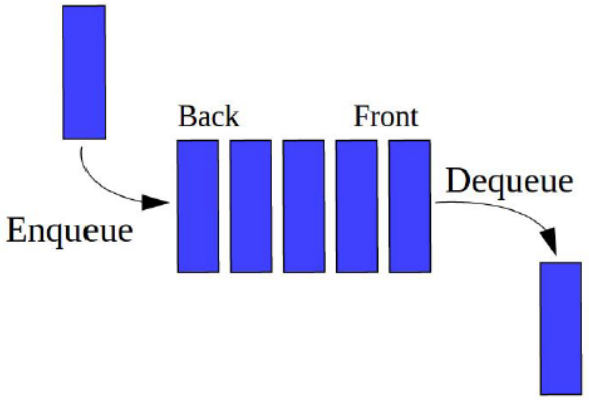
- Some use cases:
  - page-visited history in a web browser, undo sequence in a text editor, parentheses checking, iterative implementation of recursive algorithms, …
  - auxiliary data structure for many algorithms, e.g. depth-first search, …
- Typical operations: size(), isEmpty(), push(object), top(), pop(), peek()

Array-based implementation:

Implementation using linked list:



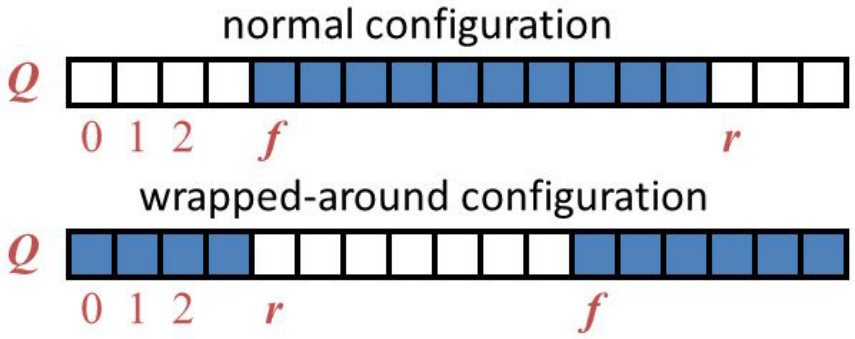
Queues



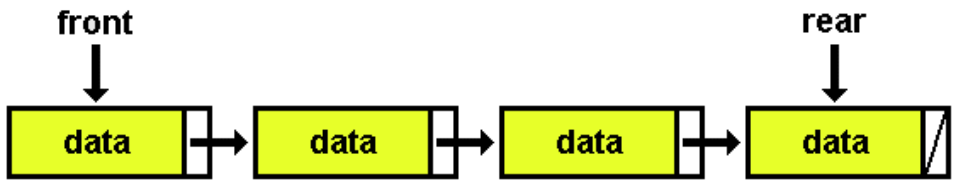
- Some use cases:
  - waiting lists, access to shared resources (e.g. round robin schedulers), …
  - auxiliary data structure for many algorithms, e.g. breath-first search, …

- Typical operations: size(), isEmpty(), enqueue(object), dequeue(), front(), back()

Array-based implementation:



Implementation using linked list:



Stack: Implementation in C using Dynamic Array

- In the `main` function, dynamically allocate memory for a struct of type `stack` on the heap segment of memory. (Don't forget to free memory at the end of the `main` function.)
- Use a preprocessor directive to define the initial size of the stack:

```
#define INITIAL_STACK_SIZE 5
```

- In the `initialize` function, dynamically allocate memory for an integer array of size `INITIAL_STACK_SIZE`. Assign the pointer to the location in dynamic memory to the field `elements`.

```
typedef struct stackADT {
    int *elements;
    int size;
    int count;
} stack;
```

(address of first element of an) integer array containing the contents of the stack

current size of the dynamical array `elements`

number of values currently stored in the stack

Code Example Stacks

Consider the following (partial) C code showing some stack operations. What will be the output?

```c
#include <stdlib.h>
#include <stdio.h>

typedef struct stackADT {
    int* elements;
    int size;
    int count;
} stack;

void initialize(stack* s);
int pop(stack* s);
int push(stack* s, int value);
int peek(stack* s);
int size(stack* s);

/* implementation of stack operations
   not shown here... */
```

```c
int main() {
    stack* myStack = malloc(sizeof(stack));
    initialize(myStack);
    push(myStack, 7);
    push(myStack, 10);
    printf("%d ", peek(myStack));
    printf("%d ", pop(myStack));
    push(myStack, 3);
    push(myStack, 5);
    printf("%d ", pop(myStack));
    printf("%d ", size(myStack));
    printf("%d ", peek(myStack));
    push(myStack, 8);
    printf("%d ", pop(myStack));
    printf("%d ", pop(myStack));
    free(myStack);
    return 0;
}
```

Solution: `10 10 5 2 3 8 3`

Code Example Queues

Consider the following (partial) C code showing some queue operations. What will be the output?

```c
#include <stdlib.h>
#include <stdio.h>
#define QUEUE_SIZE 16

typedef struct queueADT {
    int elements[QUEUE_SIZE];
    int head;
    int count;
} queue;

void initialize(queue* q);
int enqueue(queue* q, int value);
int dequeue(queue* q);
int size(queue* q);

/* implementation of queue operations
   not shown here... */
```

```c
int main() {
    queue* myQueue = malloc(sizeof(queue));
    initialize(myQueue);

    for (int i = 1; i <= 6; i++) {
        enqueue(myQueue, i);
    }
    for (int i = 0; i < size(myQueue); i++) {
        printf("%d ", dequeue(myQueue));
    }
    printf("%d ", size(myQueue));

    free(myQueue);
    return 0;
}
```

Solution: 1 2 3 3

Beware: One might get lured into answering «1 2 3 4 5 6 0» which is **wrong**. Note that the upper bound of the second `for` loop (`size(myQueue)`) is not a constant but continuously changes throughout the iterations of the `for` loop because items are removed from the queue within the loop.

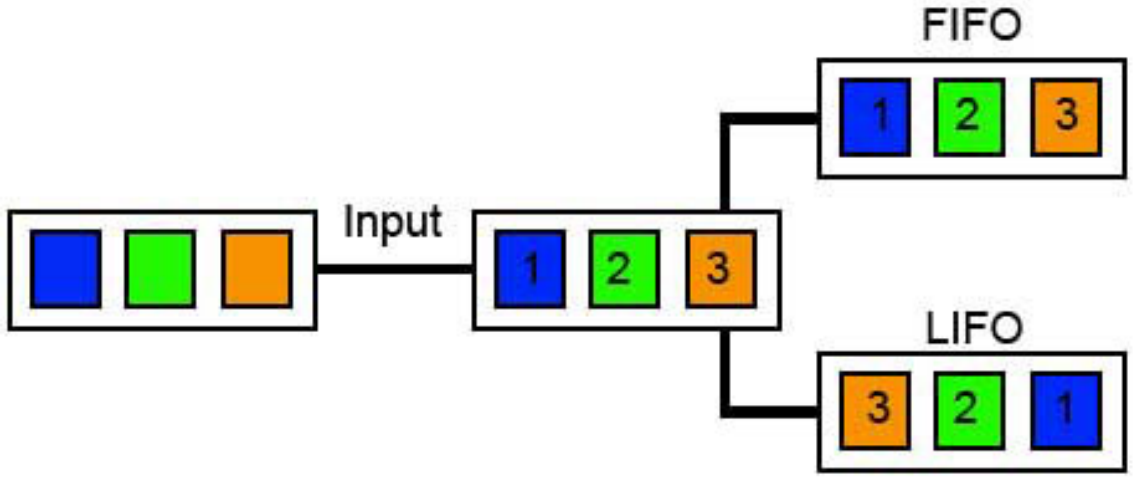
Conceptual Questions ADTs

– Why is it *not* a good idea to implement heapsort using a linked list?

– Which abstract data type is best suited to manage the current standings on the display in the finish area of a sports contest?

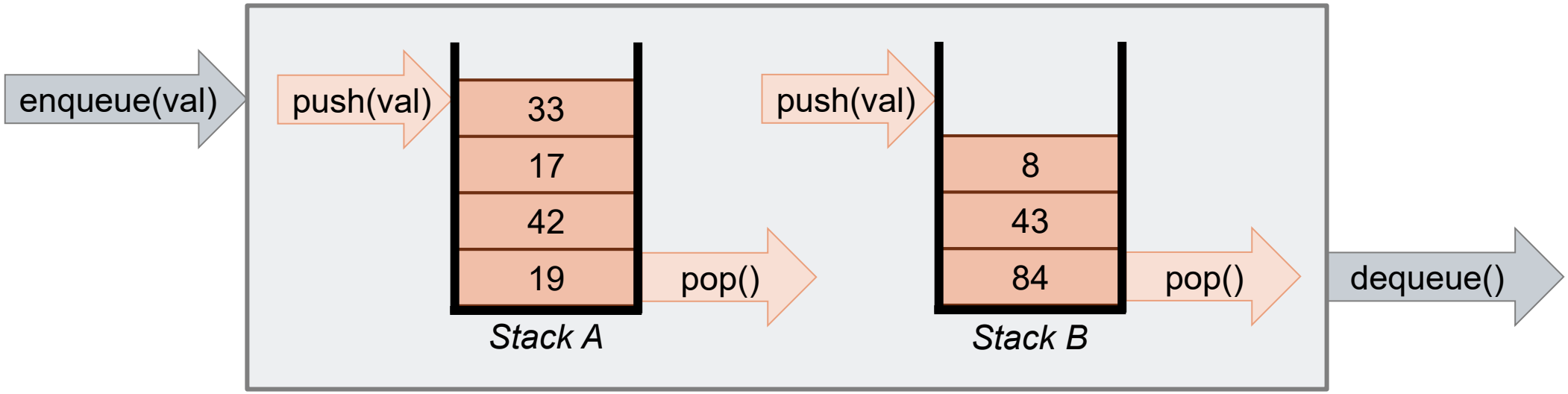
LIFO vs. FIFO

- *LIFO:* The output sequence is reverse of input sequence.
- *FIFO:* The output sequence is the same as the input sequence.



Additional Exercise: Implementing a Queue Using Two Stacks

Task: Create a queue which is based on two stacks (i.e. wrap a queue interface around a pair of stack interfaces).



Additional Exercise: Implementing a Stack Using Other ADTs

How would you implement a stack using…

a) …two queues?

b) …a (single) queue?

How would you implement a stack using a linked list?

How would you implement a stack using a binary heap?

Stacks and Queues: Additional Practice

Additional exercises for practice can be found here:

https://h5p.org/node/471454

Exercise 7, Task 1a

Consider the operations $push(S, x)$ which inserts item x into the stack S and the operation $pop(S)$ which removes the item at the top of the stack S and returns it. Further consider the operations $enqueue(Q, x)$ which inserts item x at the head of queue Q and the operation $dequeue(Q)$ which removes the item at the tail of Q and returns it.
The following sequence of operations are performed on two initially empty stacks S_1 and S_2 and two initially empty queues Q_1 and Q_2:

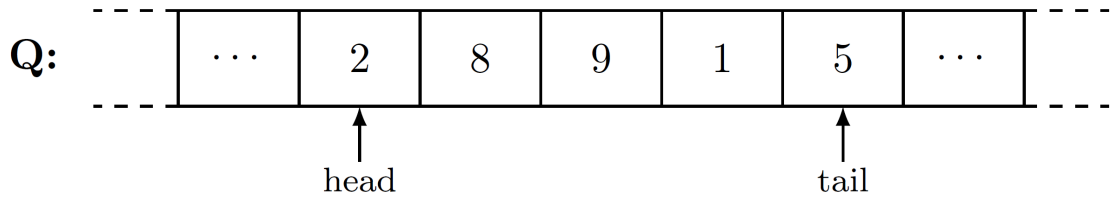
(1) $push(S_1, 5)$
(2) $push(S_2, 7)$
(3) $push(S_1, 3)$
(4) $enqueue(Q_1, 2)$
(5) $enqueue(Q_1, 4)$
(6) $enqueue(Q_2, 1)$
(7) $push(S_1, dequeue(Q_1))$
(8) $enqueue(Q_2, pop(S_2))$
(9) $push(S_2, 6)$
(10) $push(S_1, pop(S_1) + pop(S_2))$
(11) $enqueue(Q_2, pop(S_1) + dequeue(Q_2))$
(12) $push(S_2, dequeue(Q_2))$
(13) $dequeue(Q_2)$

→ 9

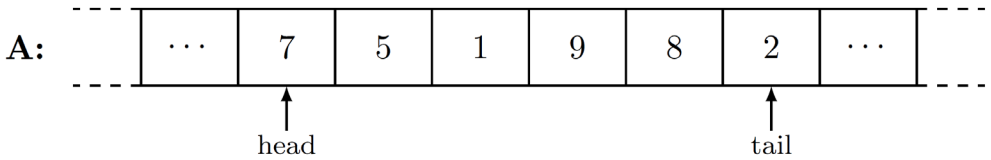
$S_1 = [3, 5]$ (where item 3 is at the top)
$S_2 = [7]$
$Q_1 = [4]$
$Q_2 = [\ ]$ (empty)

Exercise 7, Task 1b

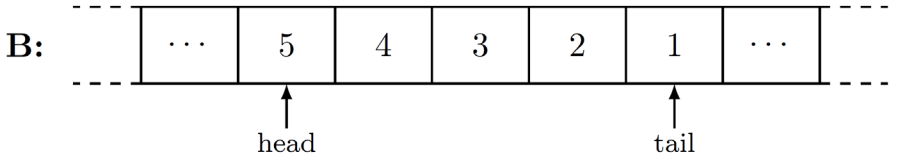
Consider the following queue Q:



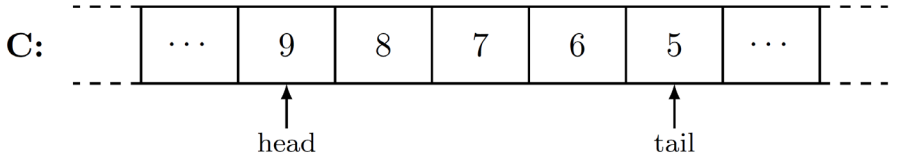
Which of the following queues could be transformed into queue Q *after less than ten* enqueue and / or dequeue operations? Any values are allowed in the enqueue operations.



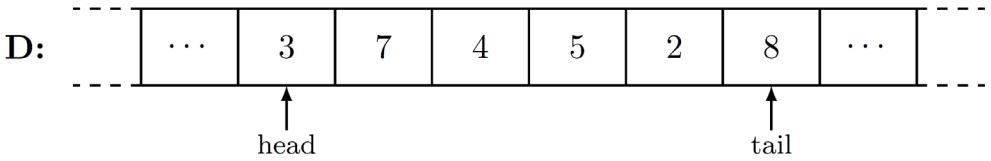
✓



✗



✗



✓

→ **A, D**

Exercise 7, Task 1c

Consider the following state of a stack S at some point in time (where the leftmost item is the top of the stack): $S = [7, 3, 5, 1]$
Which of the following are possible sequences of operations which were performed *immediately before* the situation shown above has arisen?

A: $pop(), pop(), push(2), pop(), push(7)$

B: $pop(), pop(), push(4), push(1), pop(), push(3), push(7)$

C: $push(7), pop(), push(5), pop(), push(3), pop()$

D: $push(5), push(7), push(3), pop(), pop()$

E: $push(pop()), push(pop()), push(pop()), push(pop())$

→ **A, C, E**

Exercise 7, Task 1d

Consider a queue Q which currently contains n distinct integers as items. Assume you want to remove the all items from Q which are divisible by 3. The other elements in the queue shall be in the exact same order as before after the removal of these elements and you may not use another storage for the items other than Q and a single helper variable.
How many enqueue and dequeue operations are minimally required in the best case and worst case to achieve this task?

Best case: n operations (if all items are divisible by 3)
Worst case: $2n$ operations (if no item is divisible by 3)

Algorithm: remove_divisible_three($queue, n$)

```
1 for i = 1 to n do
2     temp = dequeue(queue)
3     if temp mod 3 ≠ 0 then
4         enqueue(queue, temp)
```

Exercise 7, Task 1e

Consider a sequence of $m > 0$ stacks $(S_1, \ldots, S_m)$ and a sequence of $n > 0$ queues $(Q_1, \ldots, Q_n)$. All of the stacks and all of the queues have an identical size (capacity) of c. An input sequence of k mutually distinct integers is processed as items by these sets of m stacks and n queues in the following way:

(i) At first, any item from the input sequence is pushed to stack S_1.

(ii) Whenever a stack S_i is full, all its items are popped and then immediately pushed to the next stack S_{i+1} until the stack S_i is empty.

(iii) When the last stack in the sequence S_m is full, all its items are popped and then immediately enqueued to the first queue in the sequence Q_1 until the stack S_m is empty.

(iv) Whenever a queue Q_i is full, all its items are dequeued and then immediately enqueued to the next queue Q_{i+1} until the queue Q_i is empty.

(v) When the last queue in the sequence Q_n is full, all its items are dequeued and constitute, in the order that they were dequeued, the output sequence.

For which values of m, n, k and c will the output sequence be identical to the input sequence?

$m \geq 2,\ m \mod 2 = 0,\ n > 0,\ k = d \cdot c$ where d is a positive integer

Exercise 7, Task 2

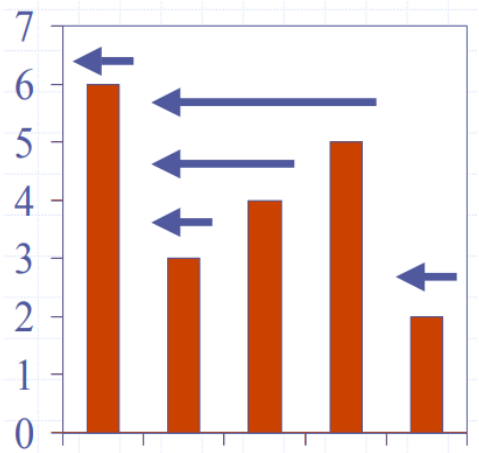
Consider the given code skeleton `task2_skeloton.c` which contains a starting point for the implementation of a stack in C. In particular, the skeleton contains a `struct` which shall be used for the implementation and is also shown below.

```
typedef struct Stack {
    unsigned int capacity;
    int* items;
  int top;
} Stack;
```

Expand the given skeleton and implement in C the following functions. Test your implementation with appropriate calls in the `main` function.

Exercise 7, Task 3

Consider an array $A[0..n-1]$ of n integers. The span $s(A,i)$ of array element $A[i]$ is the maximum number of consecutive elements $A[j]$ immediately preceding $A[i]$ and such that $A[j] \leq A[i]$. For example, array $A = [6,3,4,5,2]$ has the spans $s(A,0) = 1$, $s(A,1) = 1$, $s(A,2) = 2$, $s(A,3) = 3$ and $s(A,4) = 1$ which can be written as an array as follows: $s(A) = [1,1,2,3,1]$.



	0	1	2	3	4
X	6	3	4	5	2
S	1	1	2	3	1

Solution idea:

The indices of the elements which are “visible when looking back” are kept in a stack.

(*Application:* in financial analysis of stocks, e.g. to state that a particular stock is at its 10-week high)

Exercise 7, Task 4

Consider an initially empty stack S for which a number of n *push* operations and n *pop* operations is performed. Every time when an item is pushed to S this item is stored at the same time at the left-most (towards to the begining) of array In and every time when an item is popped from S this item is stored at the same time at the left-most of array Out.
The goal of this task is to devise and analyse an algorithm which takes two arrays $In[0..n-1]$ and $Out[0..n-1]$ as paramters and decides whether they could potentially be the result of such *push* and *pop* operations. For example, consider $In = [1, 2, 3, 4, 5]$ and $Out = [4, 5, 3, 2, 1]$. These two arrays can be the result of *push* and *pop* operations as follows:

(1) $push(1)$
(2) $push(2)$
(3) $push(3)$
(4) $push(4)$
(5) $pop() \rightarrow 4$
(6) $push(5)$
(7) $pop() \rightarrow 5$
(8) $pop() \rightarrow 3$
(9) $pop() \rightarrow 2$
(10) $pop() \rightarrow 1$

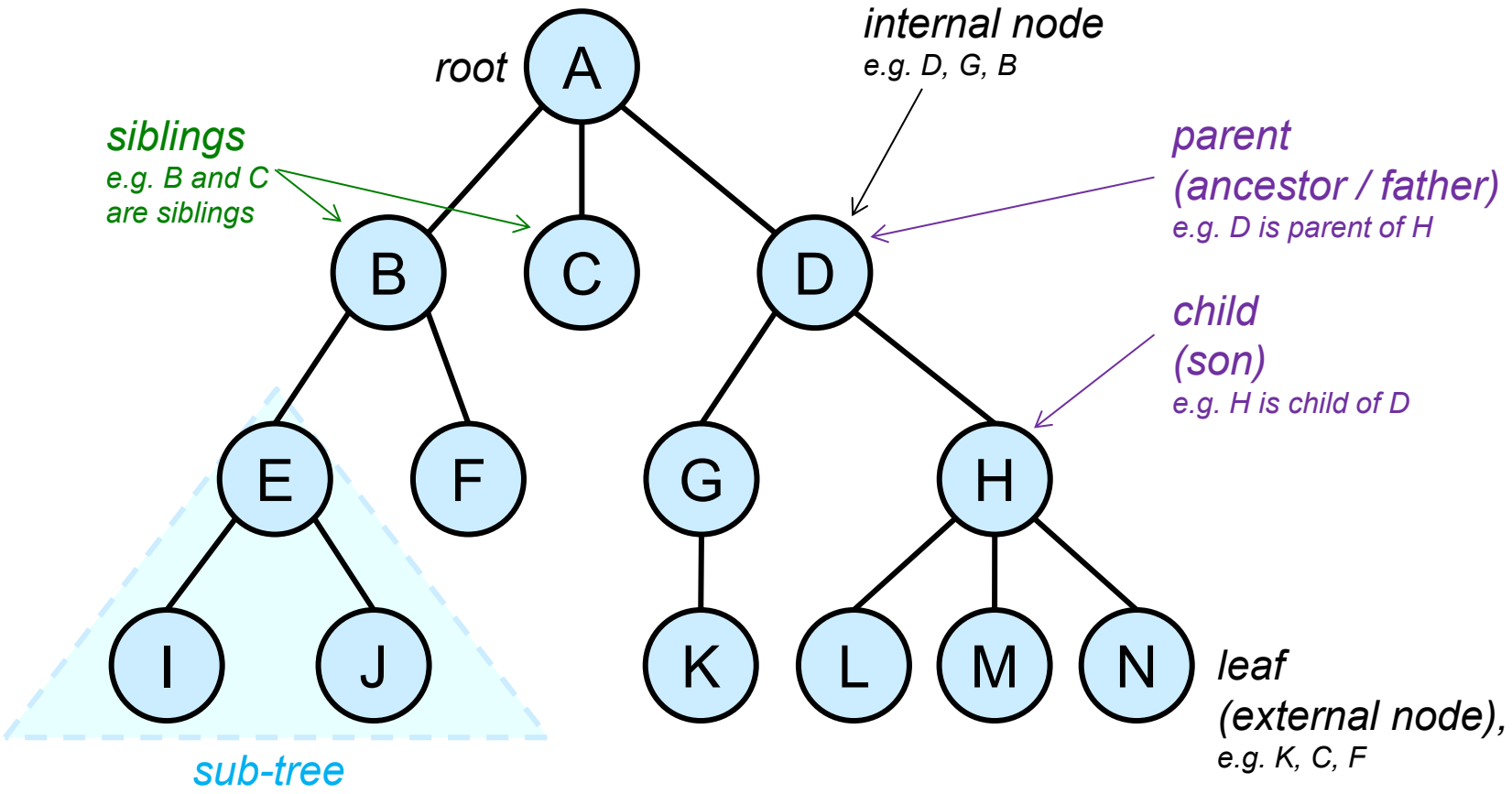
Universität Zürich[UZH]

Institut für Informatik

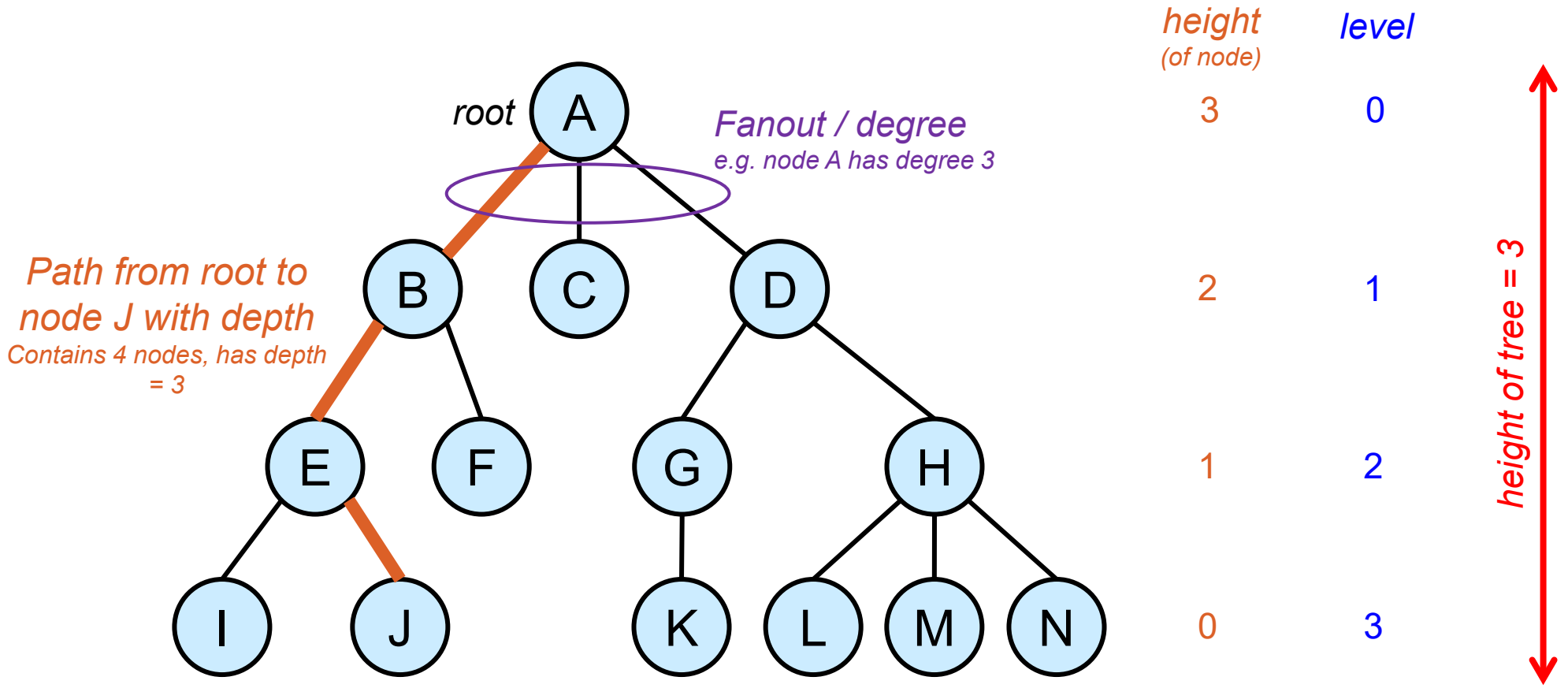
Trees

- Nomenclature
- Binary Trees
- Binary Tree Traversals
- Binary Search Trees

Trees: Nomenclature



Trees: Nomenclature

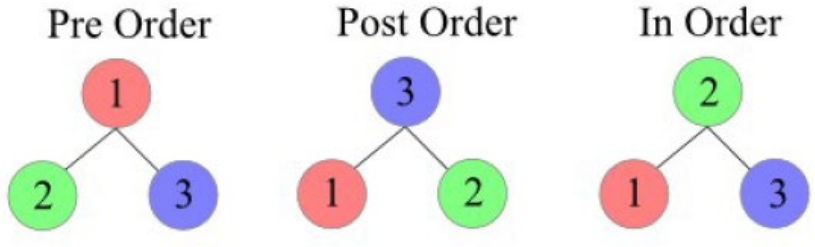


Binary Tree Traversals: Introduction and Overview

Tree traversals are rules on how to visit each node of a binary tree exactly once. There are three famous kinds of depth-first traversals and one kind of breadth-first traversal (visiting neighbors first) which will be discussed in more detail later in the lecture.

Depth-first traversals:

- Inorder: left – root – right
- Preorder: root – left – right
- Postorder: left – right – root



Breadth-first traversal:

- Levelorder: from left to right, from root level to leaf level

Binary Tree Traversals: Remarks

- Note that the sequence of nodes stemming from a particular traversal method in general does *not* allow to unambiguously reconstruct the original tree.
  - An exception is the preorder traversal which allows to reconstruct the original tree in an unambiguous and efficient manner; therefore, the preorder traversal can be used (and considered as) as representational structure for trees, e.g. in order to store a tree in a file.

Binary Tree Traversals

Recursively apply the principle stated before to all nodes of the tree by beginning at the root node.

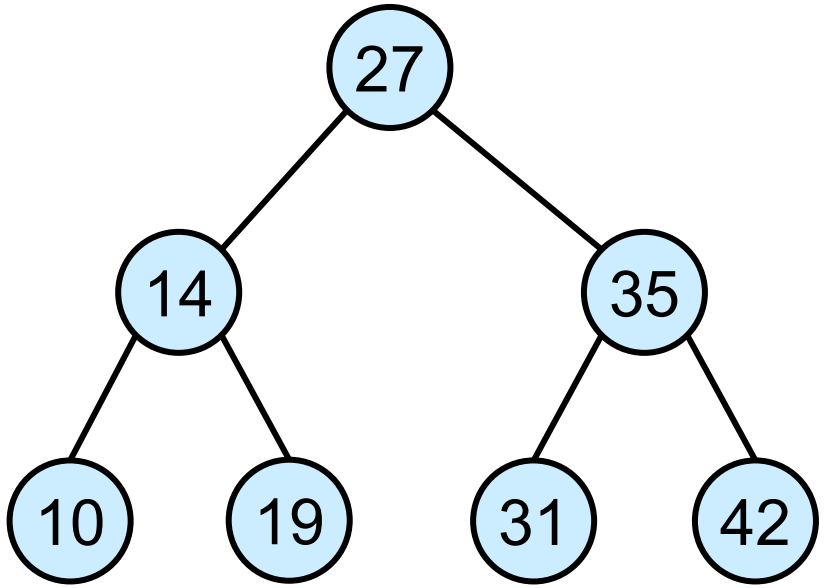
For example: inorder traversal: Left – Root – Right

Pseudo code:

```
inorder(tree T) {
    inorder(left_subtree(T));
    visit(root);
    inorder(right_subtree(T));
}
```

Binary Tree Traversals: Example

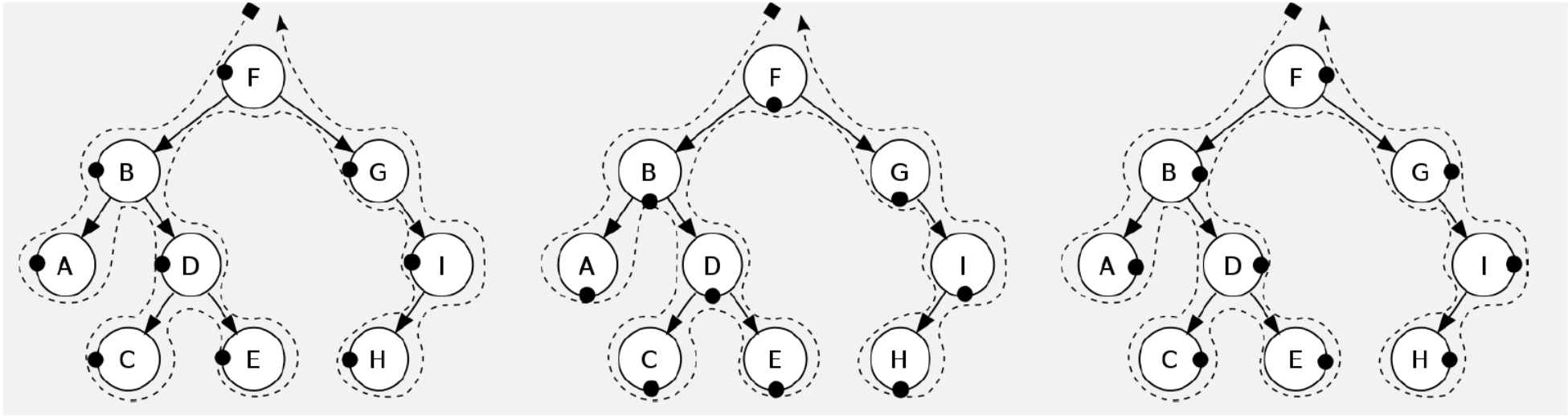
What are the inorder, preorder, postorder and levelorder traversals of the following binary tree?



Inorder: 10, 14, 19, 27, 31, 35, 42
Preorder: 27, 14, 10, 19, 35, 31, 42
Postorder: 10, 19, 14, 31, 42, 35, 27
Levelorder: 27, 14, 35, 10, 19, 31, 42

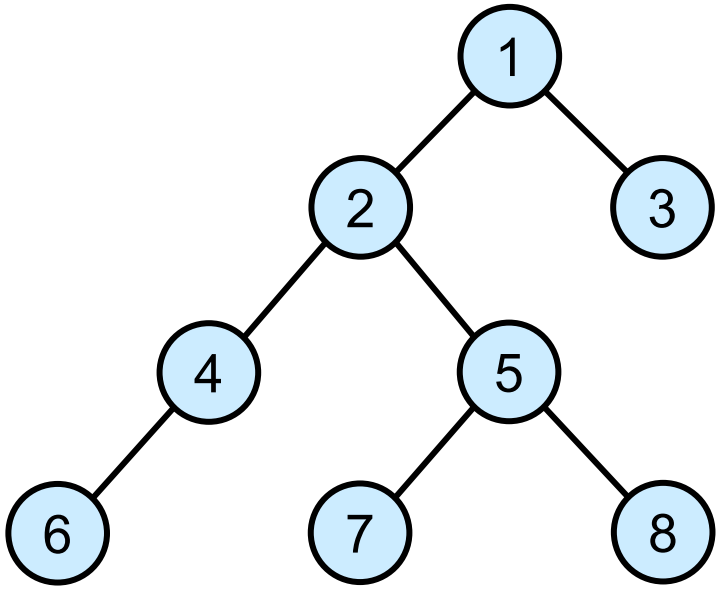
Binary Tree Traversals: Visual Help

preorder *inorder* *postorder*



Binary Tree Traversals: Iterative vs. Recursive Implementation

Instead of applying a recursive algorithm, the depth-first traversals (inorder, preorder, postorder) can also be implemented iteratively using a (explicit) stack. Similarly, a levelorder traversal can be implemented iteratively using a queue.



Step	0	1	2	3	4	5	6	7	8
Stack content	1	2 3	4 5 3	6 5 3	5 3	7 8 3	8 3	3	∅
Output of function «next»	-	1	2	4	6	5	7	8	3

Tree Traversals: Applications

What are these tree traversal strategies useful for?

- Preorder: easy and efficient way to store binary search trees with the possibility to reconstruct them unambigously
- Inorder: retrieving the node values of a binary search tree in ascending order
- Postorder: important for compiler design (in assembly: operand operand opcode)
- Levelorder: used in heapsort algorithm

Additional Exercises: Tree Walks as a Tool

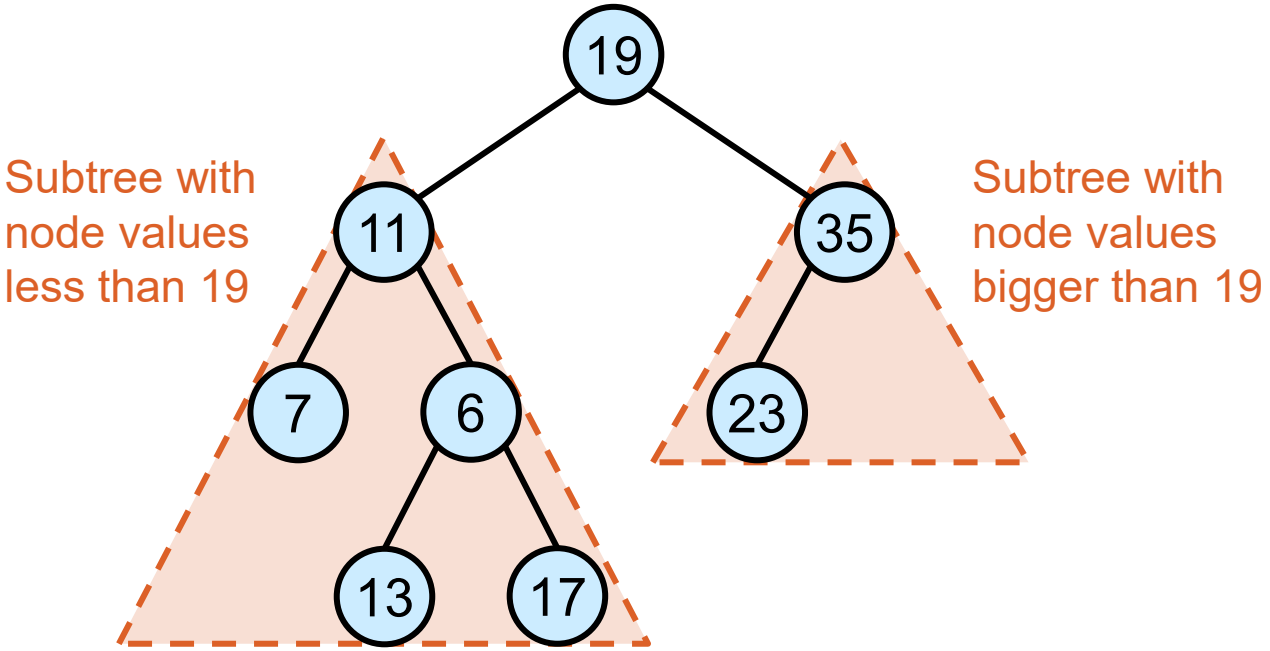
Write functions in C which calculate:

– the sum of the node values in a binary tree and
– the average of the node values in a binary tree.

Binary Search Trees

A binary search tree (BST) is a binary tree where all descendent nodes to the left are smaller or equal than the ancestor node and all descendants to the right are bigger or equal than the ancestor node.

Example:



Remark: The above example assumes that there are no duplicates in the BST (which is a common and oftentimes applicable assumption in use cases where BST make sense as an ADT. If there can be duplicates, this comes with a bunch of problems / potential ambiguities that need to be dealt with.

Binary Search Trees: Inserting Nodes / Building a Tree: Example

Insert the following key values into an initially empty binary search tree:

a) 67, 21, 57, 89, 12, 11, 7, 55

b) 42, 34, 29, 22, 17, 4

Binary Search Trees: Degeneration

If keys are inserted in sorted order into a BST, the tree will degenerate to a list.

It is said: the tree is unbalanced.

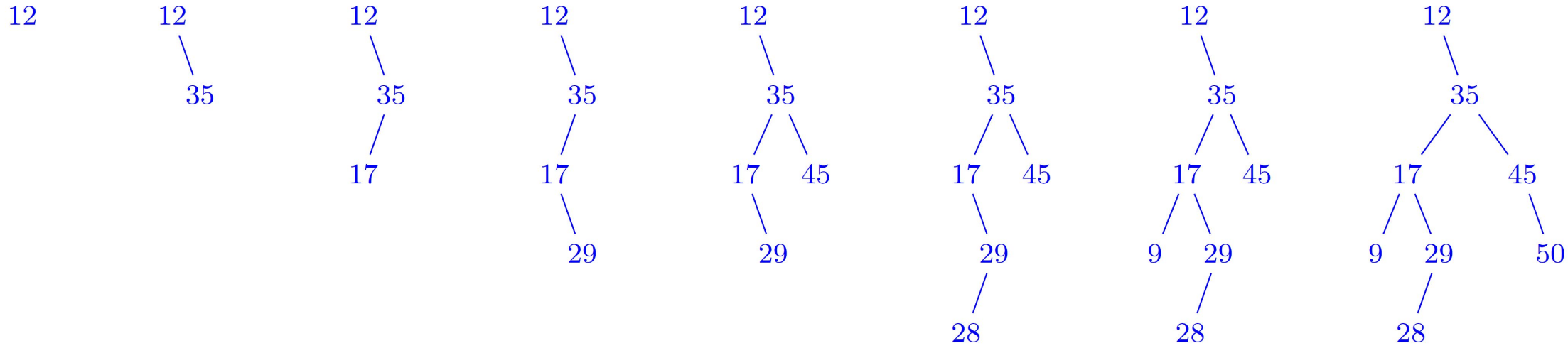
We will see techniques later which will ensure that this does not happen (red-black trees).

Additional Exercise: Degeneration Detector

Describe conceptually (or using pseudo code) how you would write an algorithm which checks whether a binary search tree has degenerated to a linked list and returns true if this is the case.

Exercise 8, Task 1a: Inserting Nodes into a BST

The following values are inserted in the given order into a previously empty BST: [12, 35, 17, 29, 45, 28, 9, 50]. Draw the tree structure when all values have been inserted.

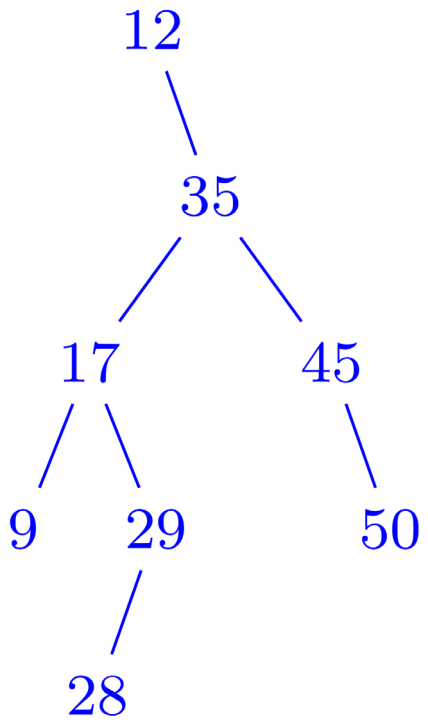


Exercise 8, Task 1b: Tree Nomenclature

In the resulting tree from task 1a determine the following properties of the tree and justify your results:

- height of the tree
- depth of node with value 45

Height of the tree: **4**
Depth of node with value 45: **2**



Exercise 8, Tasks 1c and 1d: Asymptotic Complexity of Operations

(c) Is it true that the time complexity of search in a binary search tree is O(1) in the best case? Identify when this best case is achieved.

True, complexity is O(1) if the node happens to be at the root.

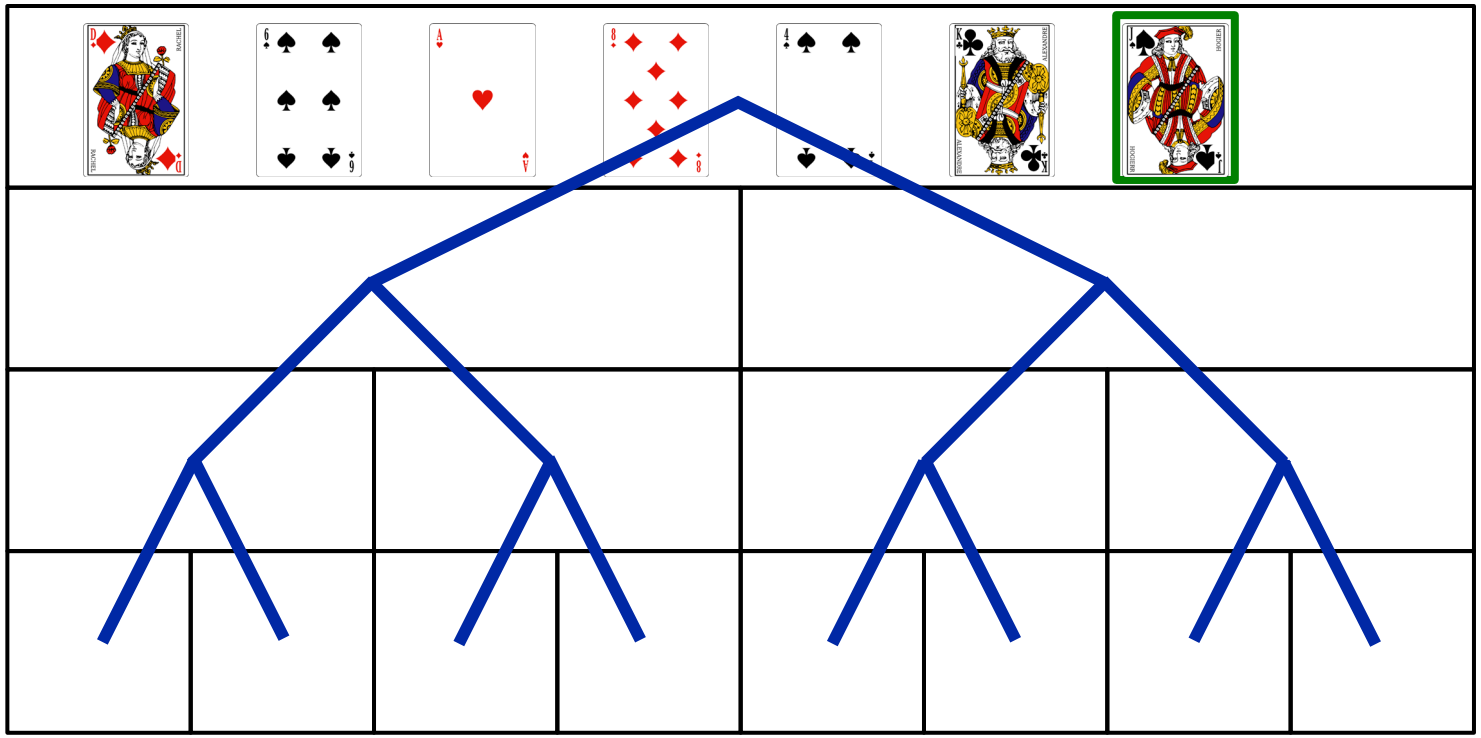
(d) Is it true that the time complexity of search in a binary search tree is O(log n) in the worst case where n is the number of nodes? Identify when this worst case is achieved.

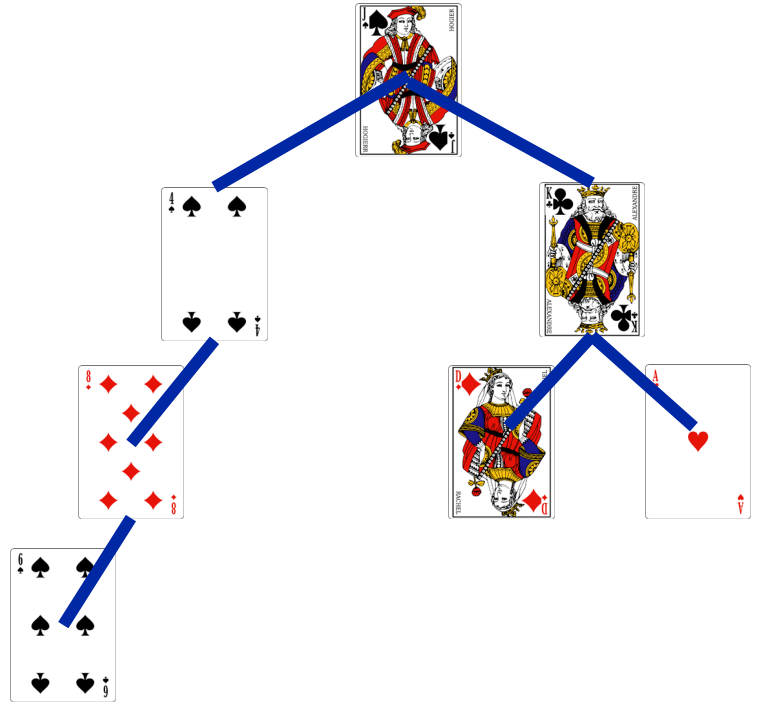
False, complexity will be **O(n)** if the tree is degenerate.

Binary Search Trees: Comprehension Question

How could a binary search tree be used to sort an array of integers? What will be the asymptotic complexity of this method?

Another Look at Quicksort

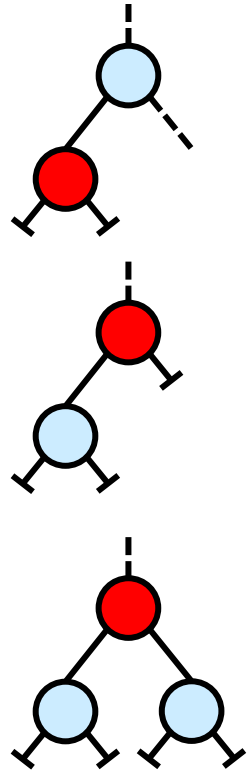




Binary Search Trees: Removing a Node

Three different cases are to be discerned:

- **A:** Node to be removed is a **leaf** (external node): just delete the leaf (the appropriate node of the parent is set to null and the node is deallocated)
- **B:** Node to be removed is **not a leaf** (internal node)
  - **B1:** Node to be removed has only **one child**: The node's parent can «adopt» the child; the node itself is then deallocated
  - **B2:** Node to be removed has **two children**: find maximum in left subtree and replace the root of the subtree with this value, than remove the maximum node of the subtree; since the maximum in the subtree can be a leaf node or an internal node of the subtree, therefore the removal has to be done using the techniques from cases A or B1 respectively (note that the maximum/minimum node cannot have two childs). Also note that this can also be done by taking the minimum in the right subtree.

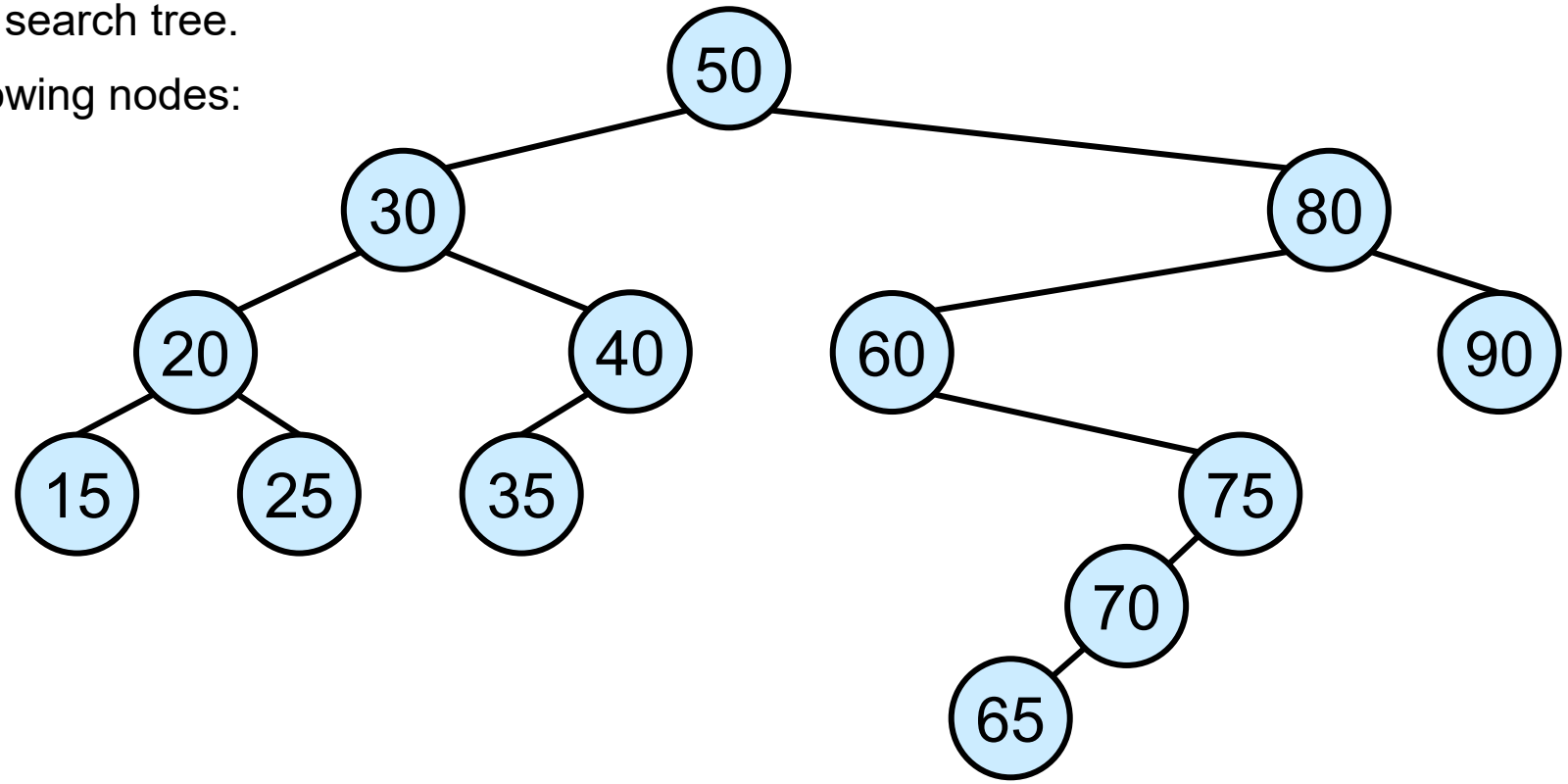


Binary Search Tree: Node Deletion Example

Consider the adjacent binary search tree.

Consecutively delete the following nodes:

35, 70, 80, 50



Additional Exercise Binary Search Trees: Asymptotic Complexity of Operations

Compose a table with the best case, average case and worst case asymptotic time complexities of the following operations applied to a binary search tree:

- insert a new node
- delete a node by its value
- search a node by its value

Additional Exercise: Binary Search Trees Traversal

Draw the binary search tree which will have a postorder traversal sequence as follows:

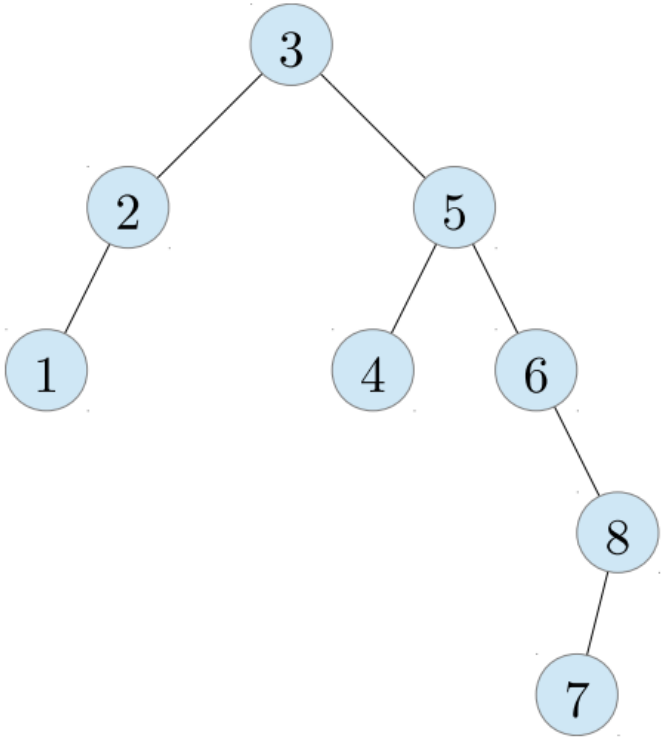
9, 5, 12, 14, 19, 18, 27, 21, 20, 10

Binary Search Trees / Tree Traversals: Additional Exercise

Draw a binary search tree which contains all integers from 1 to 8 and

- whose preorder traversal sequence starts with 3, 2, 1, 5, 4, and
- whose postorder traversal sequence ends with 7, 8, 6, 5, 3.

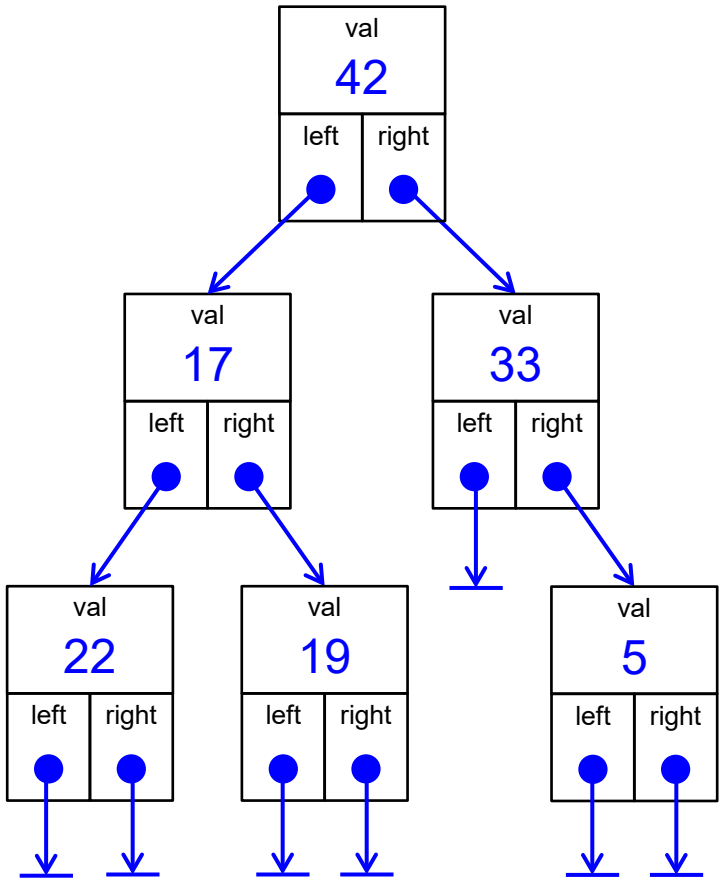
Is there more than one solution?



Binary Trees: Implementation

The nodes of a binary tree can be implemented in C using a struct as follows:

```
struct TreeNode {
        int val;
        struct TreeNode* left;
        struct TreeNode* right;
};
```



Optionally, a parent pointer may be added which allows to get the parent node of a node directly. This has advantages in some situations but can also have disadvantages in others (for example, this makes removal operations more difficult because there are more pointers which have to be changed correctly; also this needs more memory, obviously).

Exercise 8, Task 2: BST Implementation

Write a C program that contains the following functions:

(a) `struct TreeNode* insert(struct TreeNode* root, int val)`

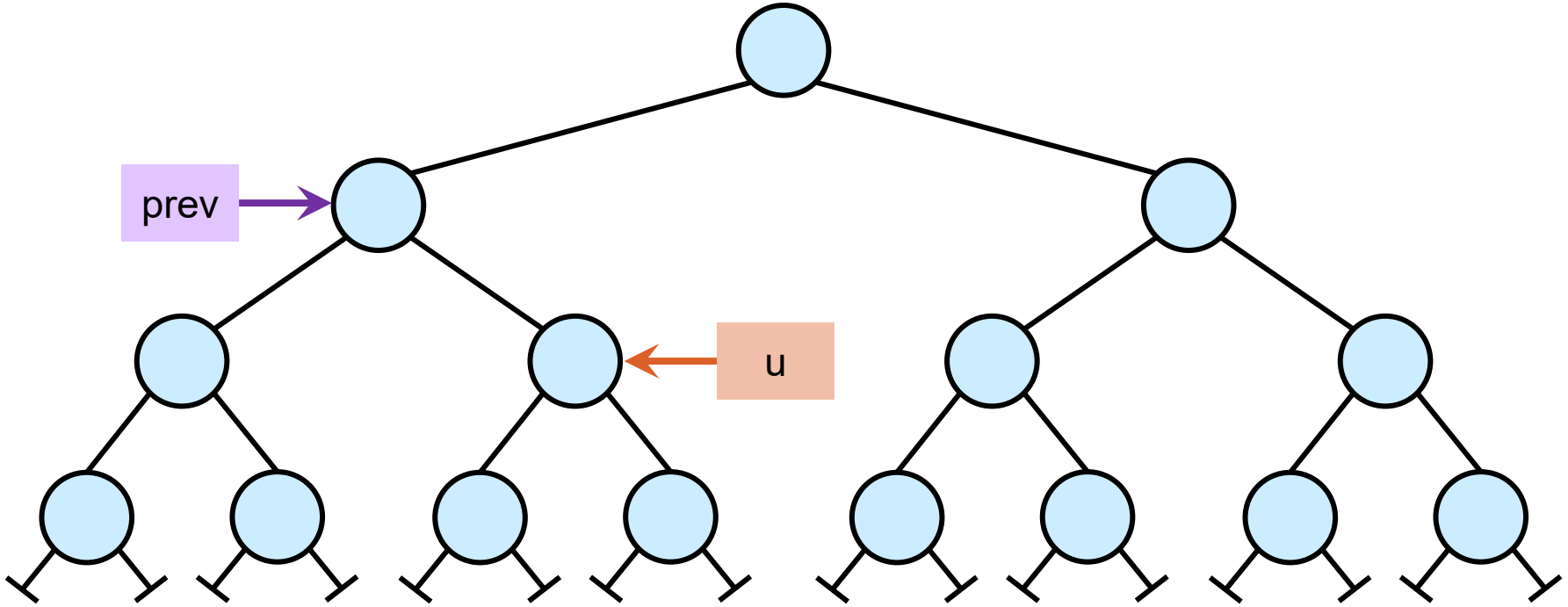
(b) `struct TreeNode* search(struct TreeNode* root, int val)`

(c) `struct TreeNode* delete(struct TreeNode* root, int val)`

(d) `void printTree(struct TreeNode* root)`

Exercise 8, Task 2c: Implementation of Deletion Operation

Idea: Maintain two pointers **u** and **prev** which point to nodes of the tree such that prev always points to the parent of u (except in the very beginning, when u points to the root and prev to NULL). Advance both these pointers until **u** points to the node which shall be deleted.



Exercise 8, Task 3: Balancing a Binary Search Tree

Exercise 8, Task 4: Range Query by Trimming a Binary Search Tree

Wrap-Up

- Summary
- Feedback
- Outlook
- Questions

Wrap-Up

– Summary

Outlook on Next Thursday's Lab Session

Coming up next: spring break

Next tutorial: Wednesday, 04.05.2022, 14.00 h, BIN 0.B.06

Topics:

- Review of Exercise 9
- Red-black Trees
- (Preview on Exercise 10)
- …
- … (your wishes)

Questions?

Thank you for your attention.