Algorithms and Data Structures

(Search) Trees

Ulf Leser

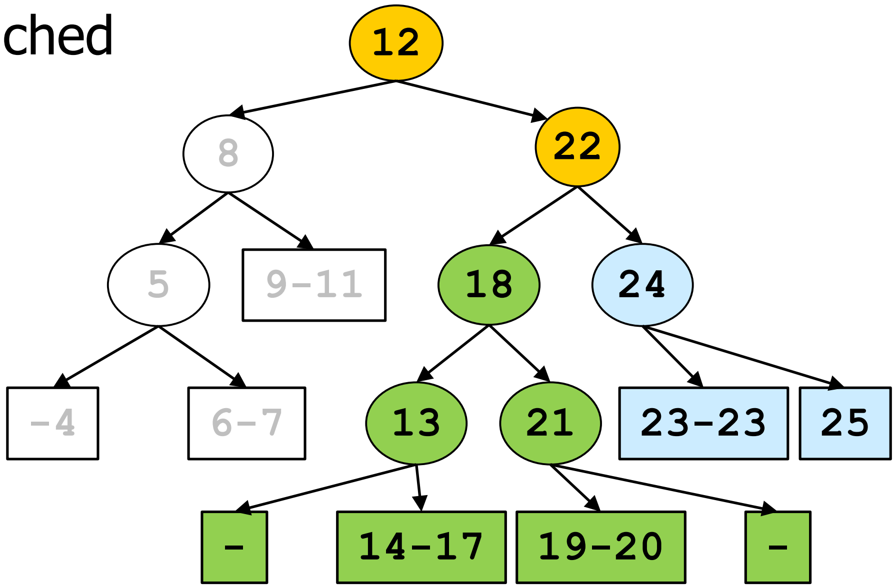
Source: whmsoft.net/

Content of this Lecture

- Trees
- Search Trees
- Natural Trees

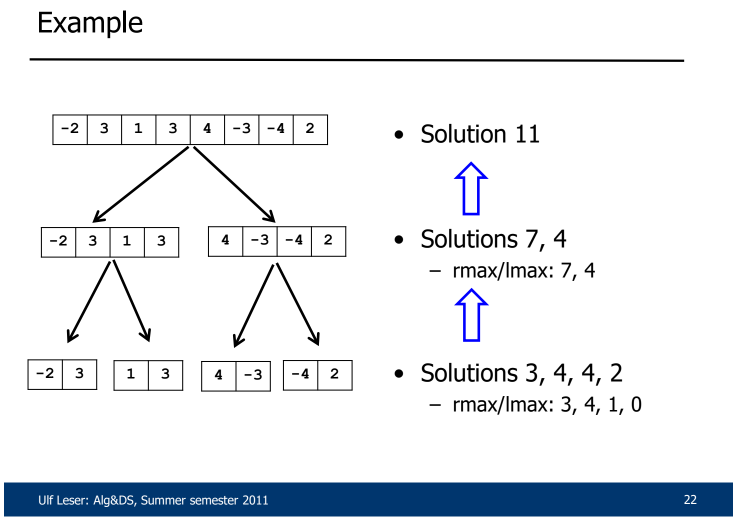
Motivation

- In a list, (almost) every element has one predecessor / successor
- In a tree, (almost) every element has one predecessor but many successors
- Elements create partitions of the set of all elements
  - Every node in a tree can be reached by only one path from root
    - I.e., path ~ element
  - Partitions: All nodes with the same path prefix
  - Prominent semantic split criterion: Order
    - Lower - left subtree,
    - Higher - right subtree

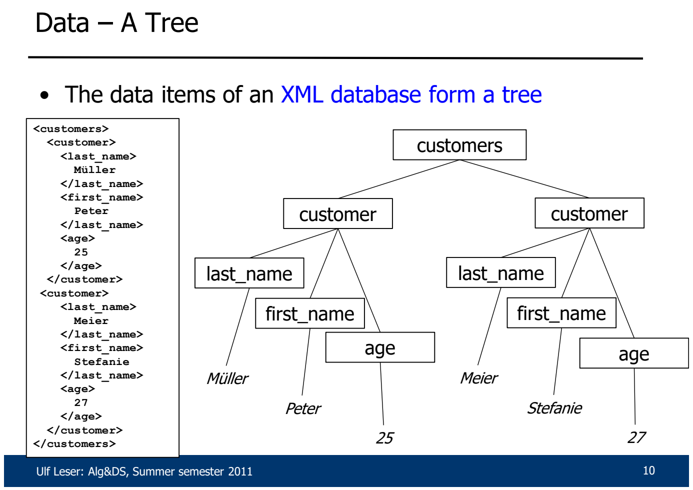


Trees are everywhere in computer science

- Divide-and-conquer partions
  - Max-subarray
  - Merge-Sort
  - QuickSort
  - …

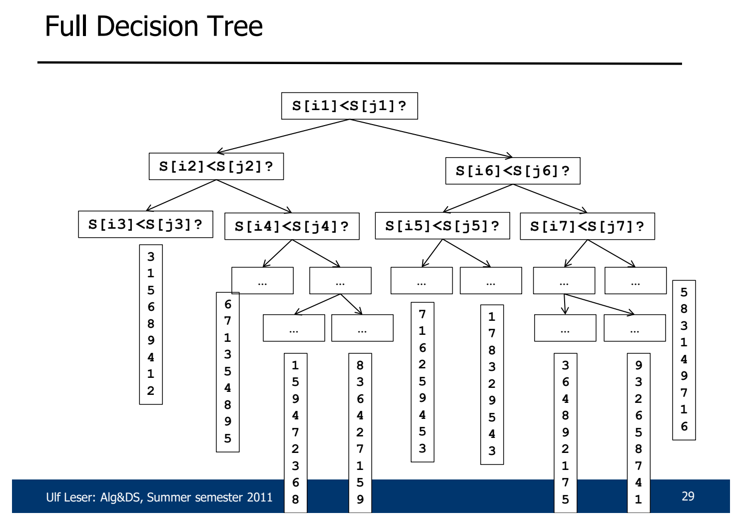


- XML
  - depth-first vs breadth-first traversal

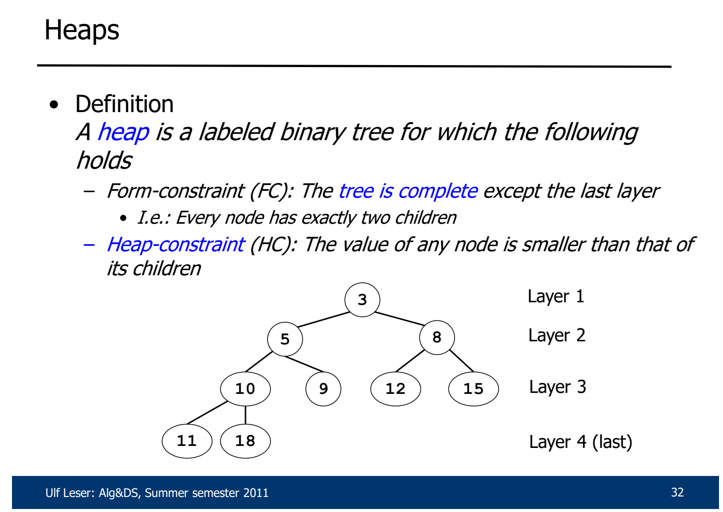


Already Seen

- Decision trees for proving the lower bound for sorting



- Heaps for priority queues



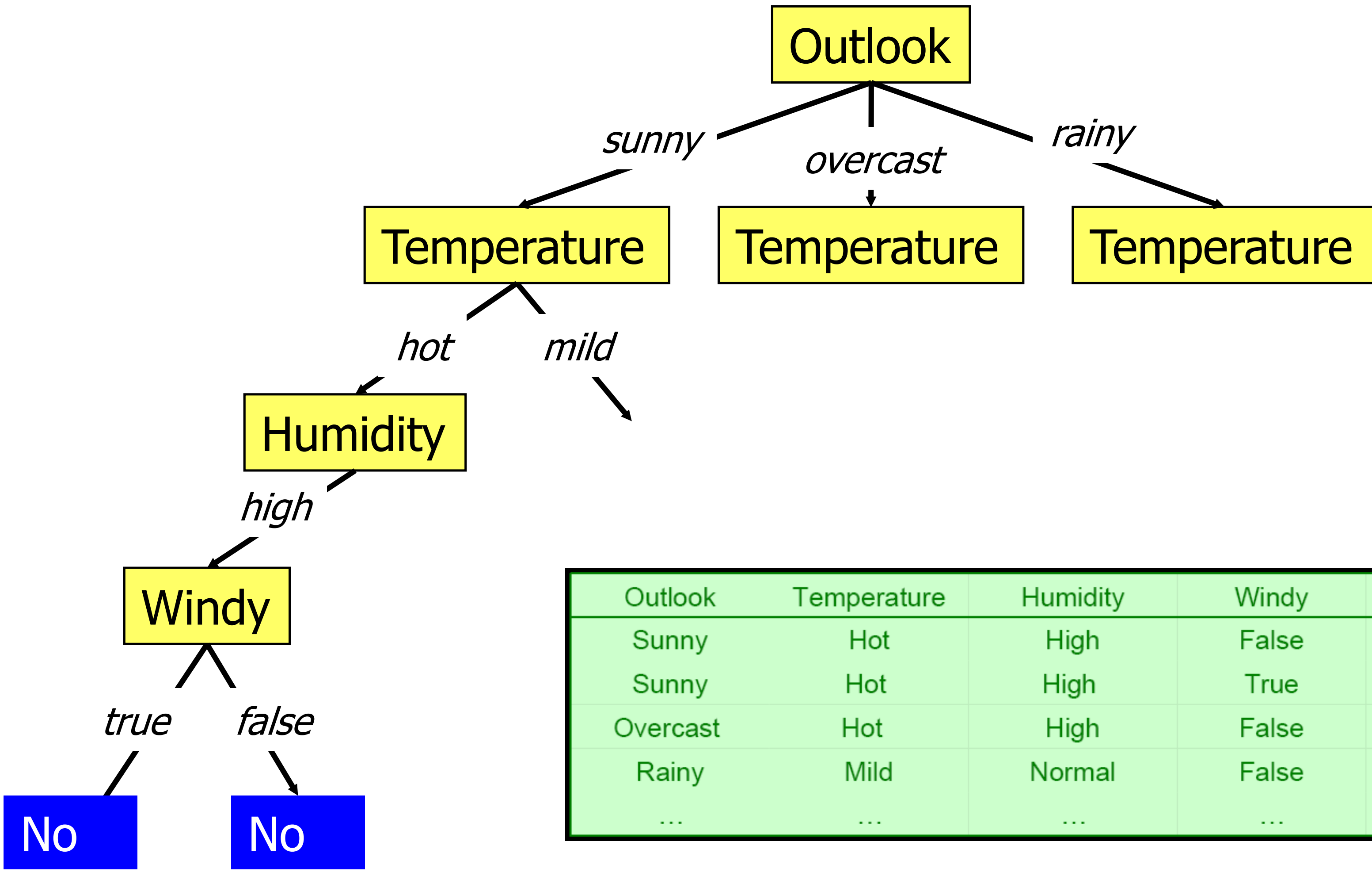
- …

Machine Learning

- Want to go to a football game?
- Might be canceled – depends on the whether
- Let‘s learn from examples

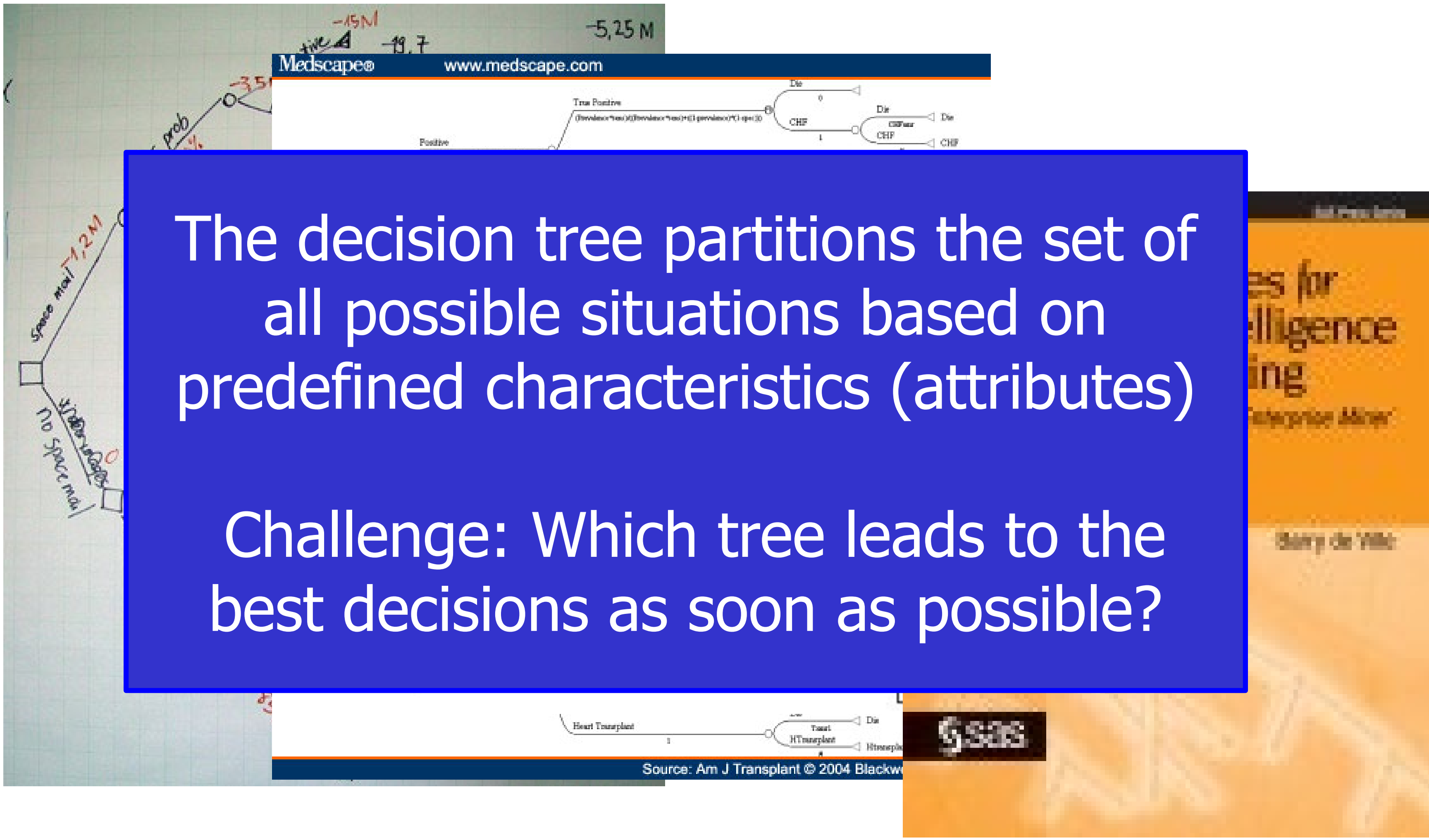
Outlook	Temperature	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	Normal	False	Yes
…	…	…	…	…

Decision Trees



Outlook	Temperature	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	Normal	False	Yes
…	…	…	…	…

Many Applications



The decision tree partitions the set of all possible situations based on predefined characteristics (attributes)

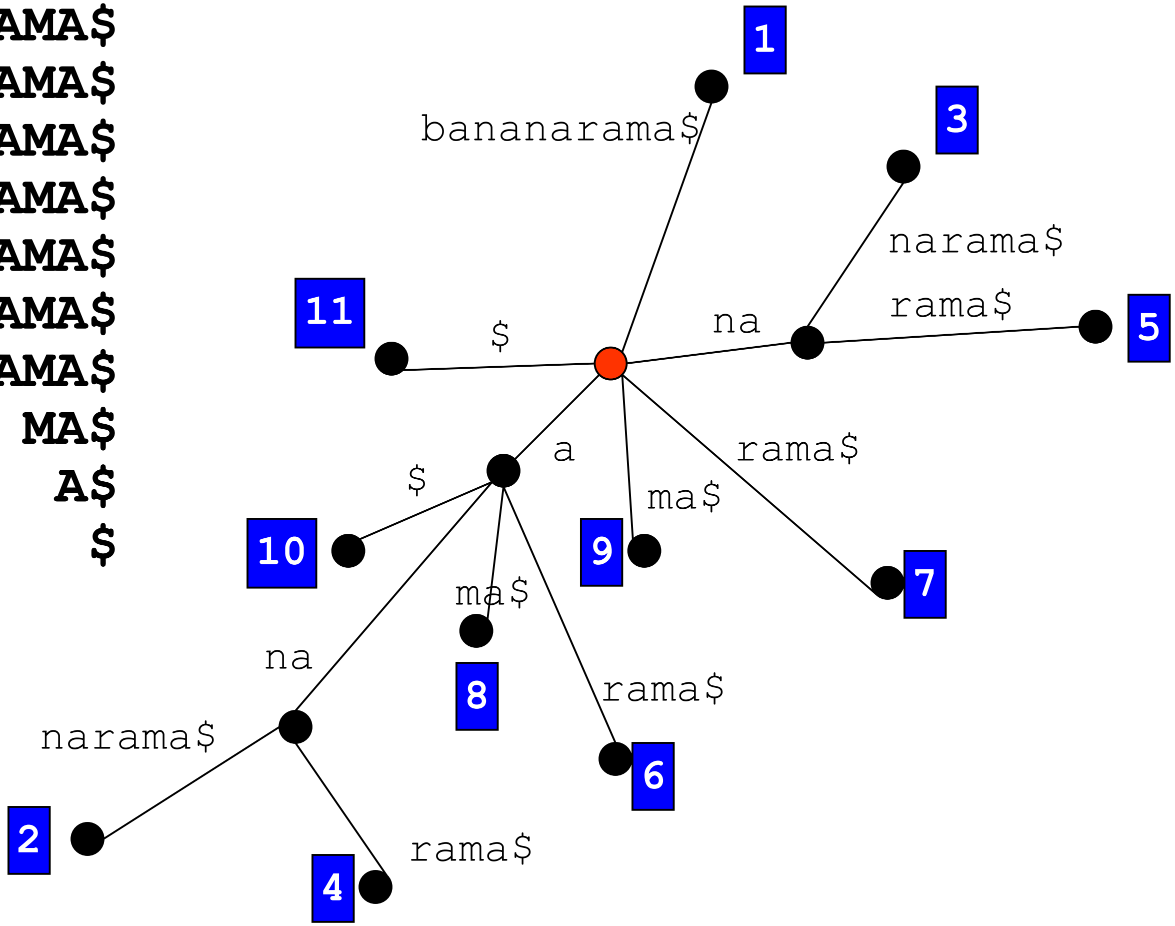
Challenge: Which tree leads to the best decisions as soon as possible?

Suffix-Trees

- Recall the problem to find all occurrences of a (short) string P in a (long) string T
- Fastest way (O(|P|)): Suffix Trees
  - Loot at all suffixes of T (there are |T| many)
  - Construct a tree
    - Every edge is labeled with a letter from T
    - All edges emitting from a node are labeled differently
    - Every path from root to a leaf is uniquely labeled
    - All suffixes of T are represented as leaves
- Every occurrence of P must be the prefix of a suffix of T
- Thus, every occurrence of P must map to a path starting at the root of the suffix tree

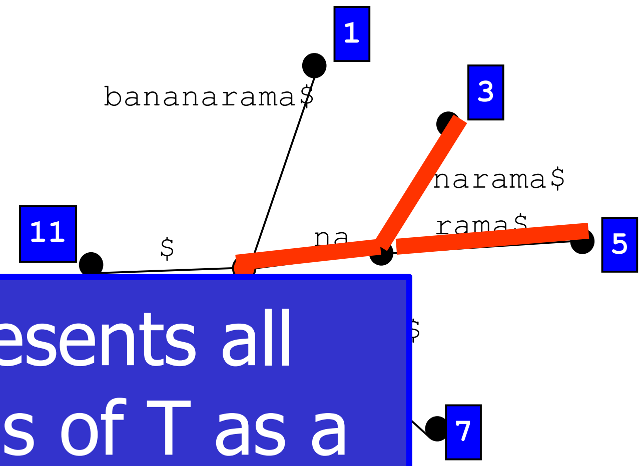
Example

```
12345678901
BANANARAMA$
 ANANARAMA$
  NANARAMA$
   ANARAMA$
    NARAMA$
     ARAMA$
      RAMA$
       AMA$
        MA$
         A$
          $
```



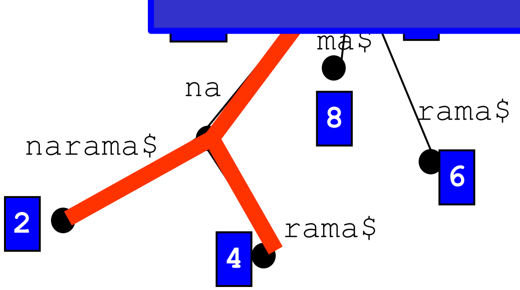
Searching in the Suffix Tree

P = „na“



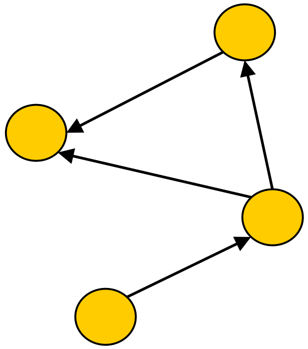
The suffix tree for T represents all common prefixes of suffixes of T as a unique path from root.

Challenge: Construction of a suffix tree in linear time.

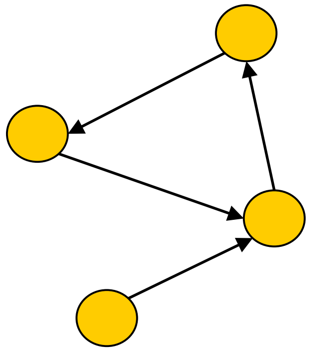


P = „an“

Not Trees

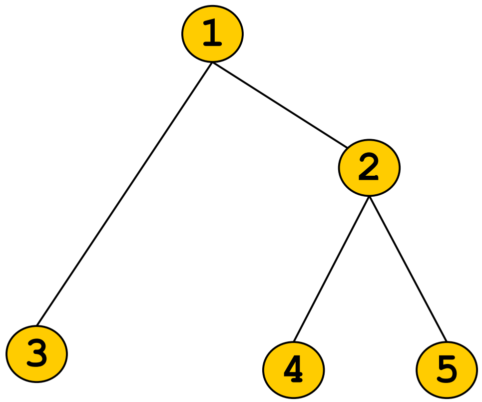


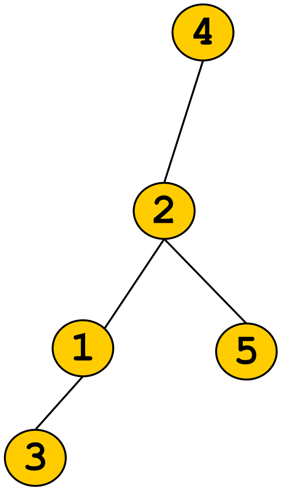
DAG: Directed, acyclic graph



General (directed) graph

Directed?





We sometimes draw undirected edges with root at the top and assume directed edges from root to leaves

This visual aid is necessary! Otherwise, root and leaves are indistinguishable

Graphs

- Definition

  A graph $G=(V, E)$ consists of a set V of vertices (nodes) and a set E of edges ($E \subseteq V \times V$).

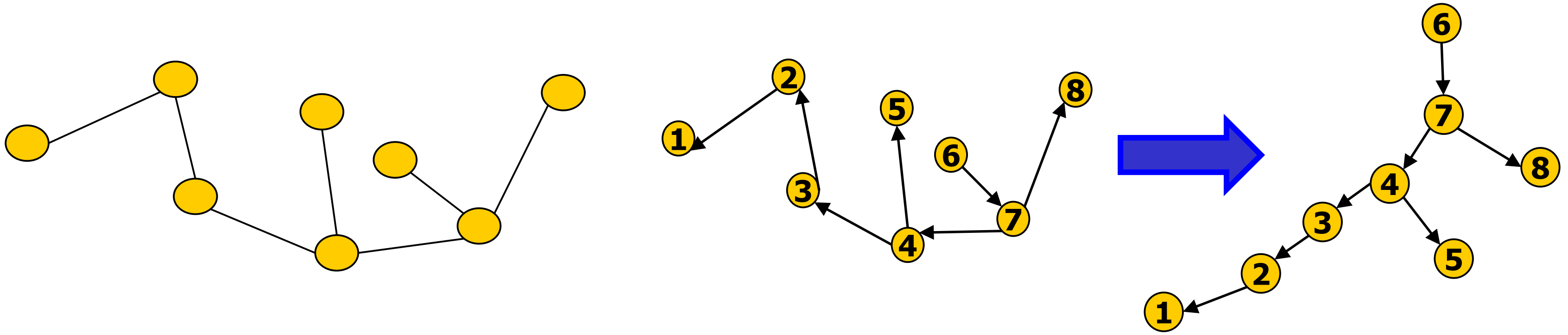
  - *A sequence of edges $e_1, e_2, .., e_n$ is called a path iff $\forall 1 \le i < n-1$: $e_i=(v', v)$ and $e_{i+1}=(v, v'')$*
  - *The length of a path $e_1, e_2, .., e_n$ is n*
  - *A path $(v_1,v_2), (v_2,v_3), \ldots, (v_{n-1},v_n)$ is acyclic iff all v_i are different*
  - *G is undirected, if $\forall (v,v') \in E \Rightarrow (v',v) \in E$. Otherwise G is directed*
  - *G is connected if every pair v_i, v_j is connected by at least one path*
  - *G is acyclic if it contains no cyclic path*

  Let $G=(V, E)$ be a directed graph and let $v,v' \in V$.

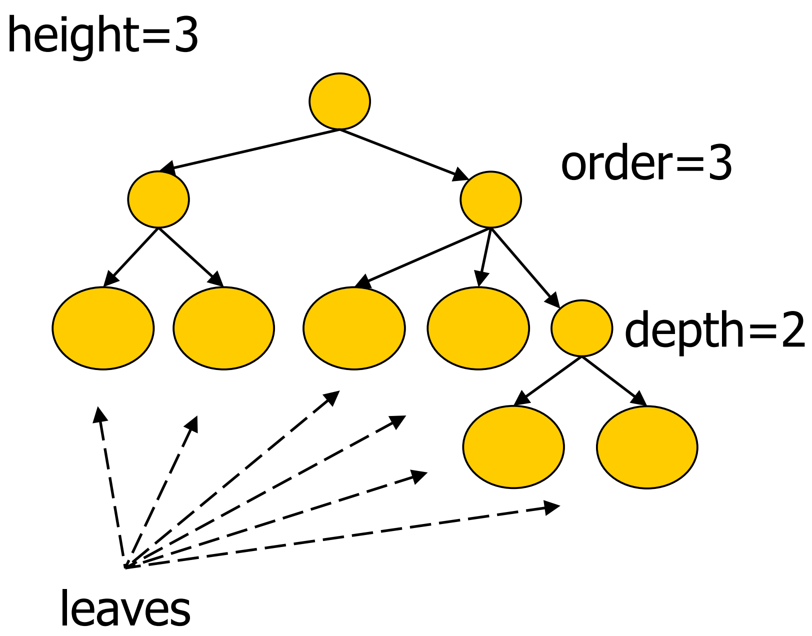
  - *Every edge $(v,v') \in E$ is called outgoing for v*
  - *Every edge $(v',v) \in E$ is called incoming for v*

Trees as Connected Graphs

- Definition
  - *A undirected connected acyclic graph is called a undirected tree*
  - *A directed acyclic graph in which all but one vertex have in-degree 1 and one vertex has in-degree 0 (the root) and there is a path from this node to every other node is called a directed rooted tree*
- From now on: "Tree" means "rooted directed tree"
- Lemma
  - *In a tree, there exists exactly one path between root and any other node*

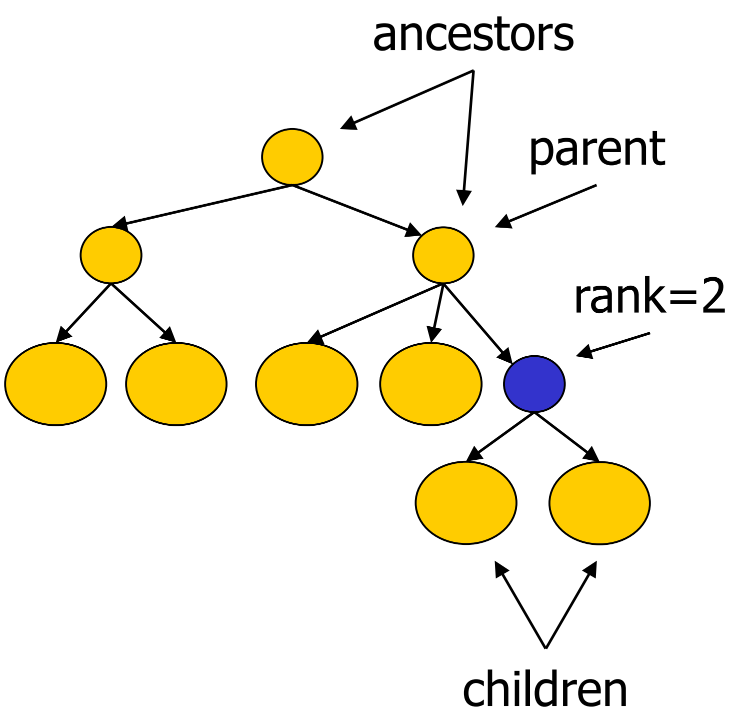


Terminology



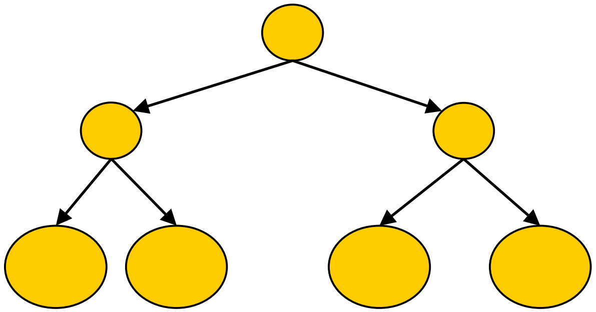
- Definition
  Let T be a tree. Then …
  - *A node with no outgoing edge is a leaf; other nodes are inner nodes*
  - *The depth of a node p is the length of the path from root to p*
  - *The height of T is the depth of its deepest leaf*
  - *The order of T is the maximal number of children of its nodes*
  - *"Level i" are all nodes at depth i*
  - *T is ordered if the children of inner nodes are ordered*

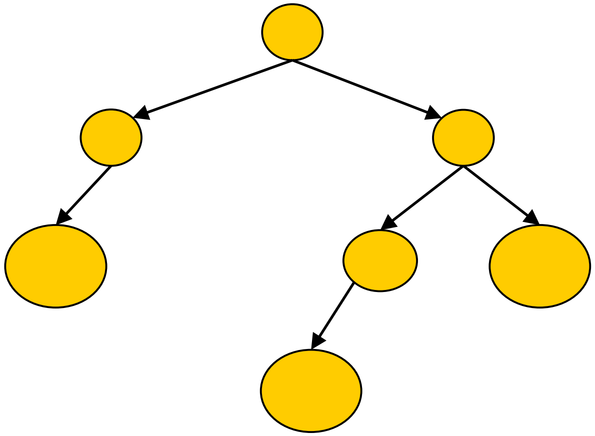
More Terminology



- Definition
  Let T be a tree and v a node.
  - *All nodes adjacent to an outgoing edge of v are v's children*
  - *v is called the parent of all its children*
  - *All nodes on the path from root to v without v are the ancestors of v*
  - *All nodes reachable from v are its successors*
  - *The rank of a node v is the number of its children*

Two More Concepts



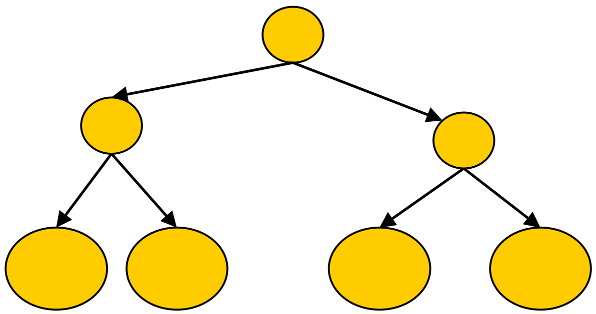


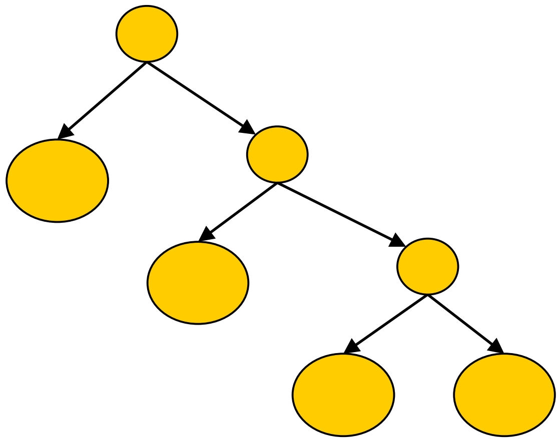
- Definition
  Let T be a directed tree of order k. T is complete if all its inner nodes have rank k and all leaves have the same depth
- In this lecture, we will mostly consider rooted ordered trees of order two (binary trees)

Recursive Definition of Trees

- We often traverse trees using recursive functions
- Definition
  A (binary) tree is a structure defined as follows:
  - *A single node is a tree with height 0*
  - *If T_1 and T_2 are trees, then the structure formed by a new node v and edges from v to the root of T_1 and from v to the root of T_2 is a tree*
    - *v is its root*
    - *The height of this tree is max(height(T_1), height(T_2))+1;*
  - *If T_1 is a tree, then the structure formed by a new node v and an edge from v to the root of T_1 is a tree*
    - *v is its root*
    - *The height of this tree is height(T_1)+1;*

Some Properties (without proofs)





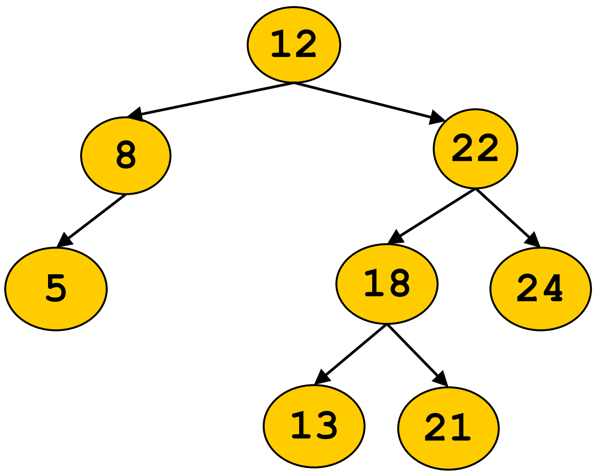
- Lemma
  Let T=(V, E) be a tree of order k. Then
  - *|V|=|E|+1*
  - *If T is complete, T has* $k^{height(T)}$ *leaves*
  - *If T is a complete binary tree, T has* $2^{height(T)+1}-1$ *nodes*
  - *If T is a binary tree with n leaves, height(T)* $\in$ *[floor(log(n)), n-1]*

Content of this Lecture

- Trees
- Search Trees
  - Definition
  - Searching
  - Inserting
  - Deleting
- Natural Trees

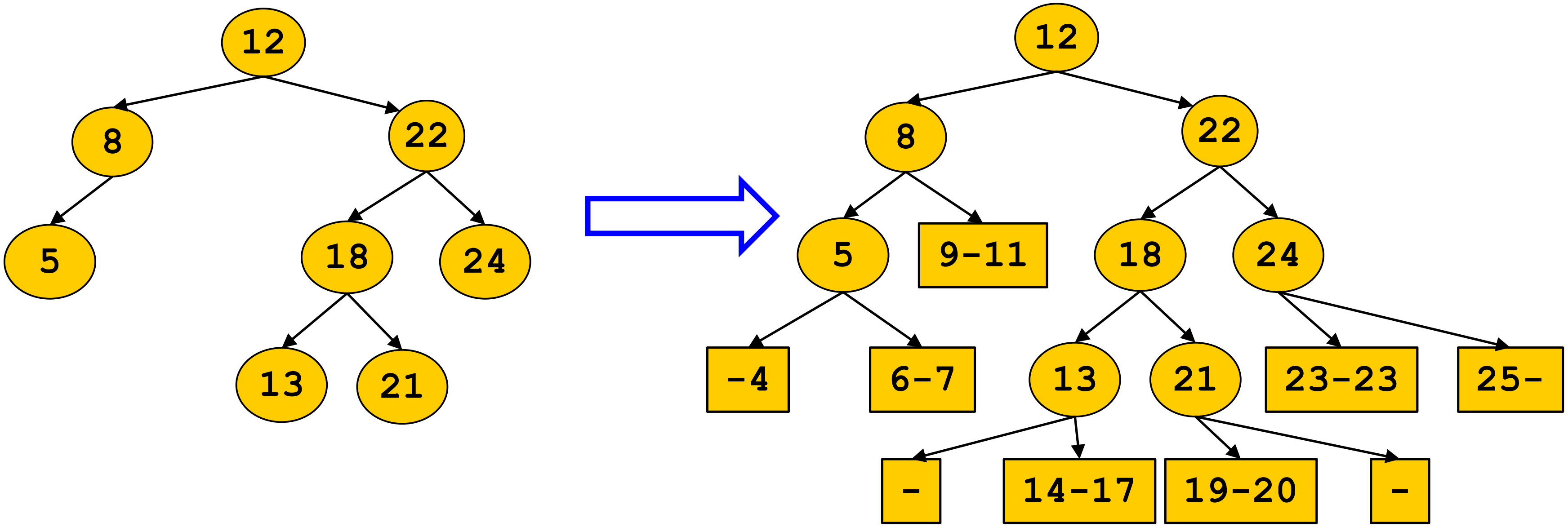
Search Trees

- Definition
  A search tree T=(V,E) for a set of n unique keys is a labeled binary tree with |V|=n and
  - *label(v)>max(label(left_child(v)), label(successors(left_child(v)))*
  - *label(v)<min(label(right_child(v)), label(successors(right_child(v)))*
- Remarks
  - For simplicity, we use integer labels
  - "node" ~ "label of a node"
  - We only consider search trees without duplicate keys (easy to change)
  - Search trees are used to manage and search a list of keys
  - Operations: search, insert, delete



Complete Trees

- Conceptually, we sometimes pad search trees to full rank in all nodes
  - "padded" leaves are usually neither drawn nor implemented (NULL)
- A "padded" leaf represents the interval of values that would be below this node



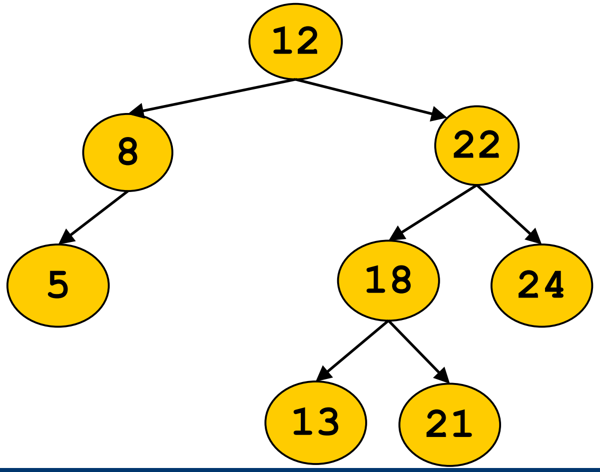
What For?

- For a search tree T=(V,E), we eventually will reach O(log(|V|)) for searching, inserting and deleting a key k
  - First: Average Case of natural trees
  - Next: Worst Case for AVL-Trees
- Compared to binsearch on arrays, search trees are a dynamically growing / shrinking data structure
  - But need to store pointers
  - Complete trees can be easily managed in arrays

Searching

- Searching a key k
  - Comparing k to a node determines whether we have to look further down the left or the right subtree
    - We stop if label(node)=k
  - If there is no child left, k∉T
- Complexity
  - In the worst case we need to traverse the longest path in T to show k∉T
  - Thus: O(|V|)
  - Wait a bit …

```
func node search( T search_tree,
                  k integer) {
  v := root(T);
  while v!=null do
    if label(v)>k then
      v := v.left_child();
    else if label(v)<k then
      v := v.right_child();
    else
      return v;
  end while;
  return null;
}
```

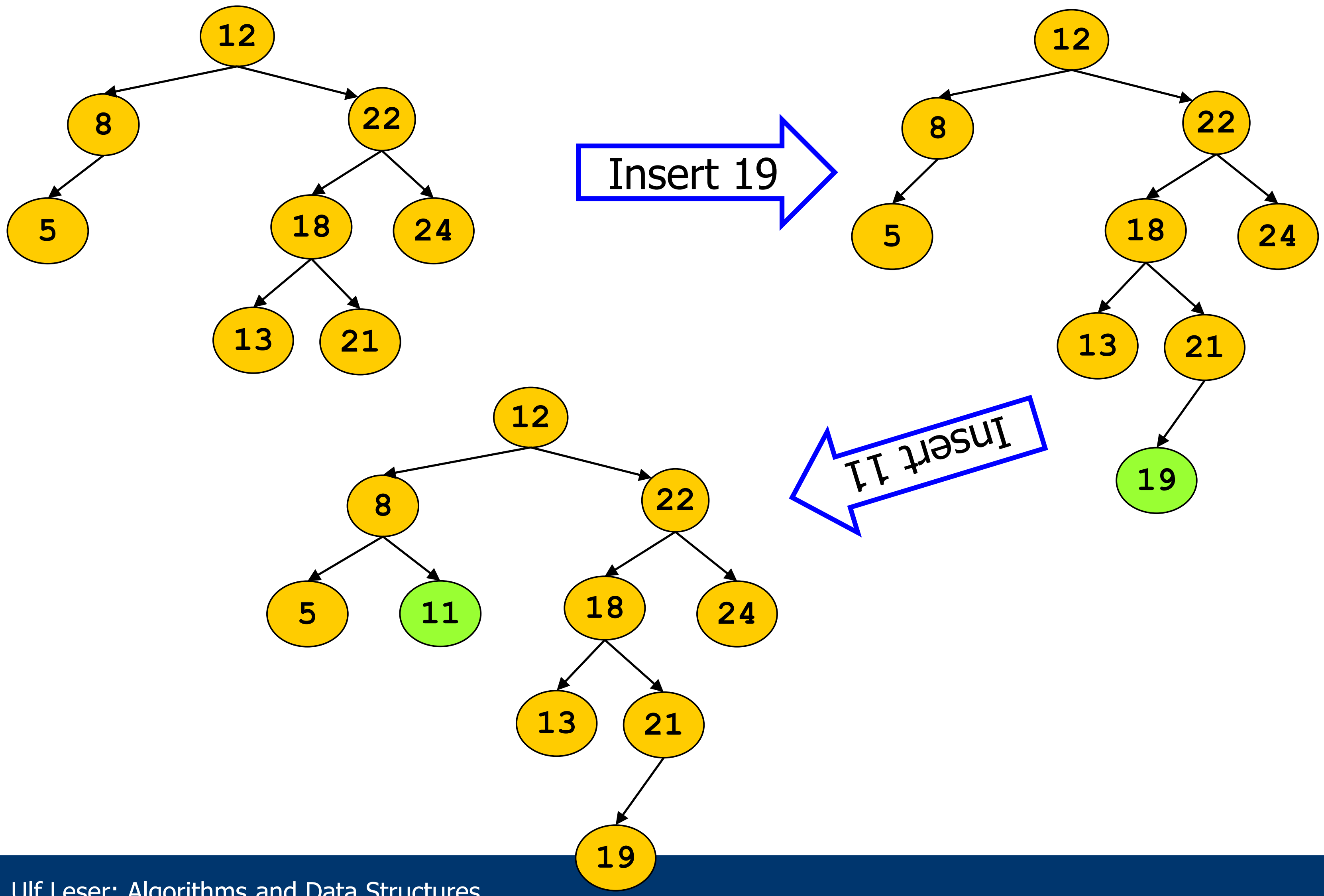


Insertion

```
func bool insert( T search_tree,
                  k integer) {
  v := root(T);
  while v!=null do
    p := v;
    if label(v)>k then
      v := v.left_child();
    else if label(v)<k then
      v := v.right_child();
    else
      return false;
  end while;
  if label(p)>k then
    p.left_child := new node(k);
  else
    p.right_child := new node(k);
  end if;
  return true;
}
```

- First search the new key k
  - If $k \in T$, we do nothing
  - If $k \notin T$, the search must finish at a null pointer in a node p
    - A “right pointer” if label(p)<k, otherwise a “left pointer”
- We replace the null with a pointer to a new node k
- Complexity: Same as search

Example

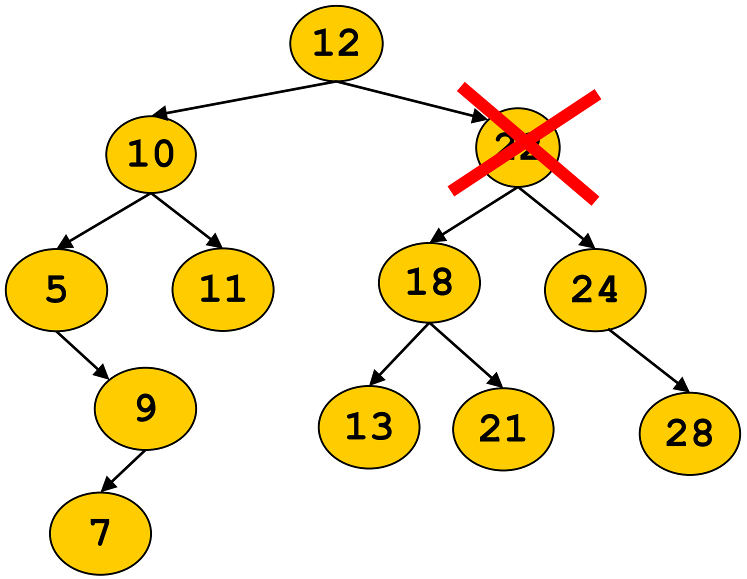

12
8
22
5
18
24
13
21
Insert 19
12
8
22
5
18
24
13
21
19
Insert 11
12
8
22
5
11
18
24
13
21
19

Deletion

- Again, we first search k
- If k∉T, we are done
- Assume k∈T. The following situations are possible
  - k is stored in a leaf. Then simply remove this leaf
  - k is stored in an inner node q with only one child. Then remove q and connect parent(q) to child(q)
  - k is stored in an inner node q with two children. Then …

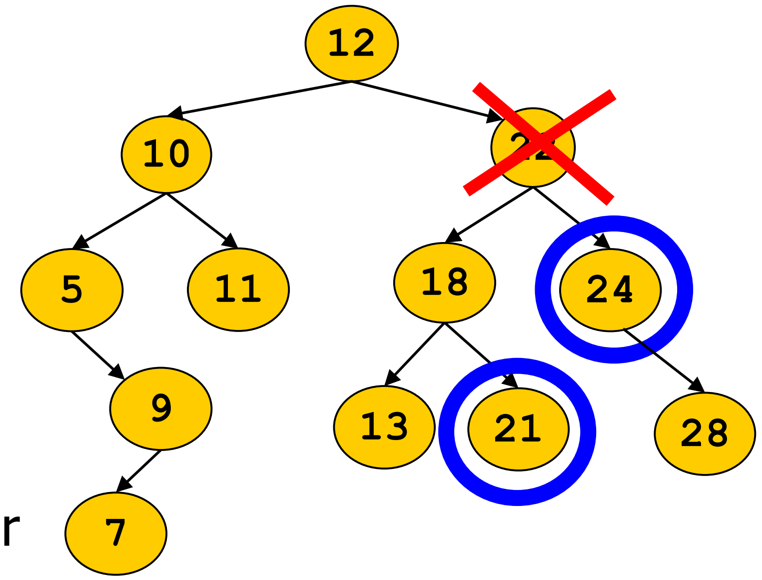
Observations

- We cannot remove q, but we can replace the label of q with another label - and remove this node
- We need a node q’ which can be removed and whose label k’ can replace k without hurting the search tree constraints
  - label(k’)>max(label(left_child(k’)), label(successors(left_child(k’)))
  - label(k’)<min(label(right_child(k’)), label(successors(right_child(k’)))



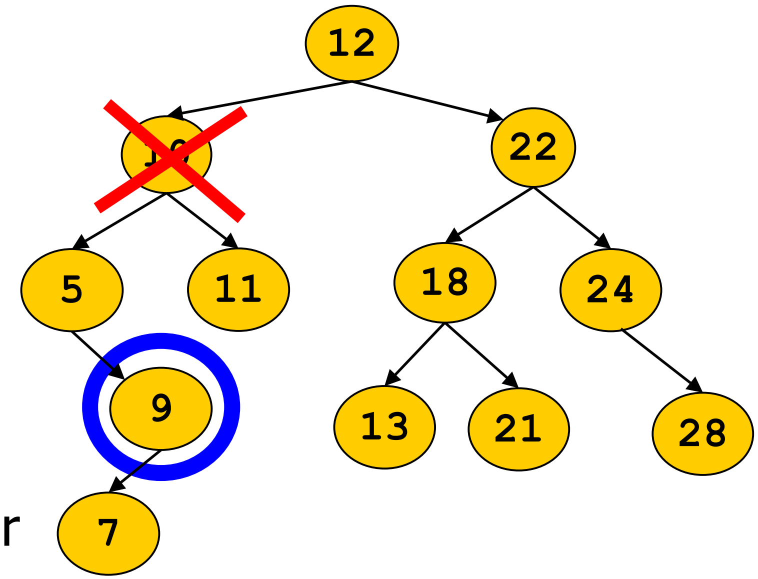
Observations

- Two candidates
  - Largest value in the left subtree (symmetric predecessor of k)
  - Smallest value in the right subtree (symmetric successor of k)
- We can choose any of those
  - Let's use the symmetric predecessor
  - This is either a leaf – no problem

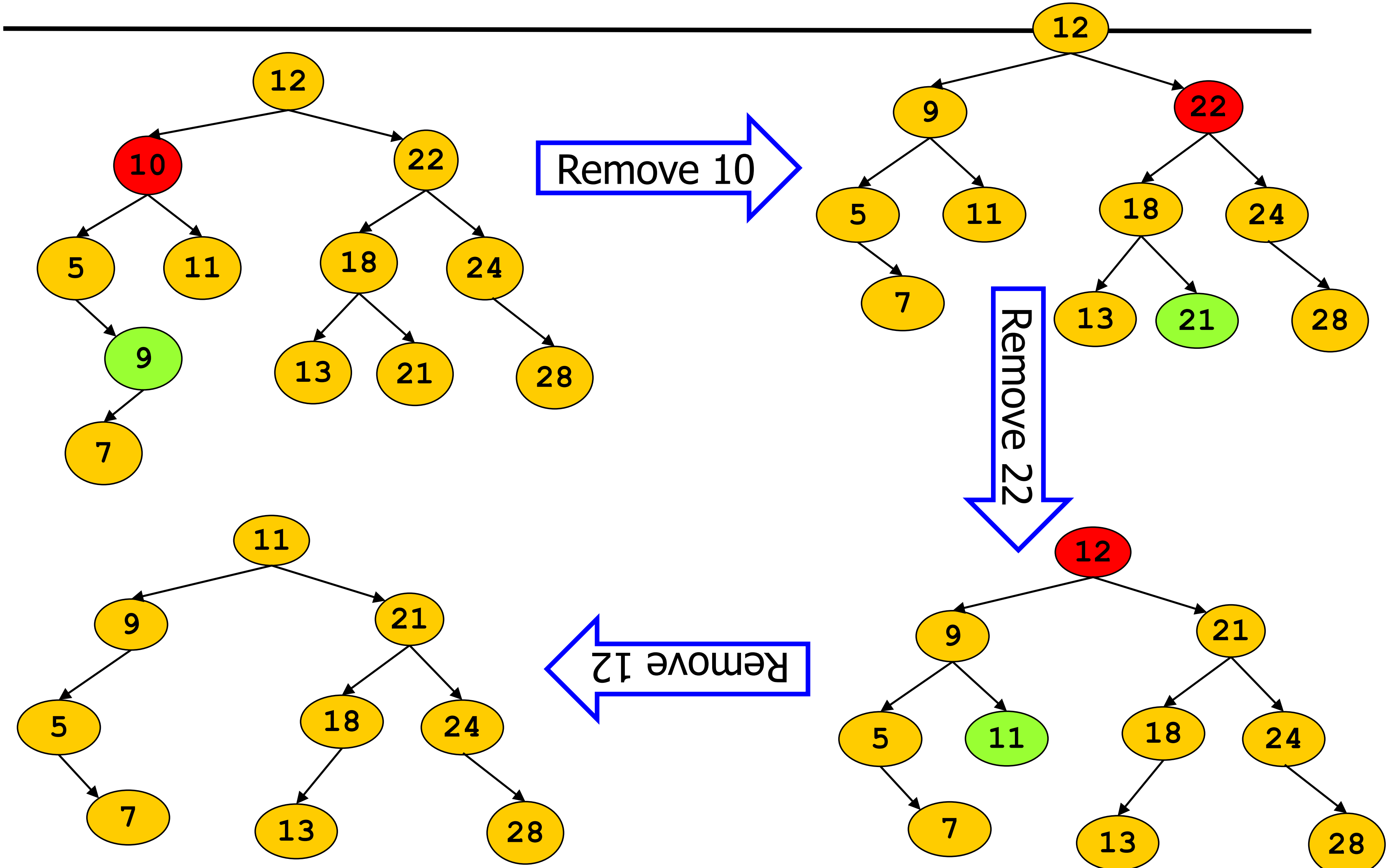


Observations

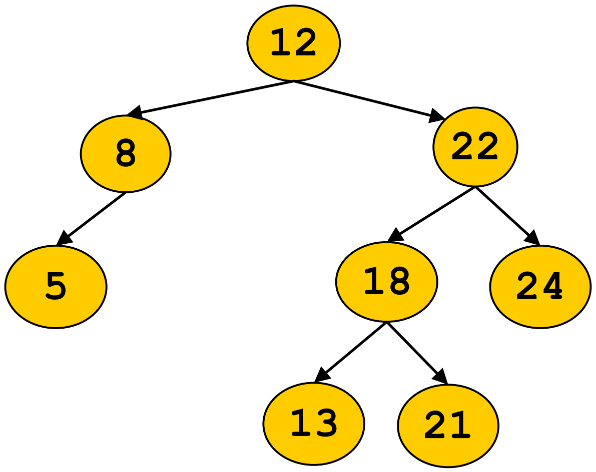
- Two candidates
  - Largest value in the left subtree (symmetric predecessor of k)
  - Smallest value in the right subtree (symmetric successor of k)
- We can choose any of those
  - Let's use the symmetric predecessor
  - This is either a leaf
  - Or an inner node; but since its label is larger than that of all other labels in the left subtree of q, it can only have a left child
  - Thus it is a node with one child - and can be removed easily

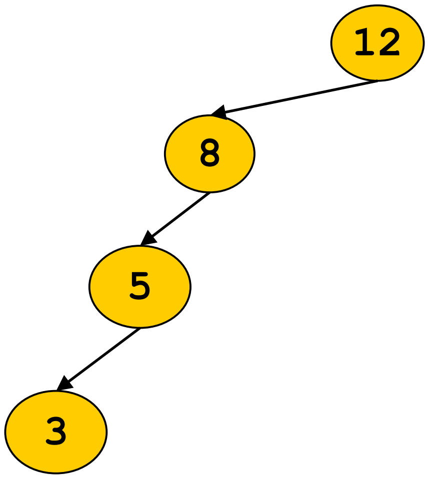


Example



Quiz





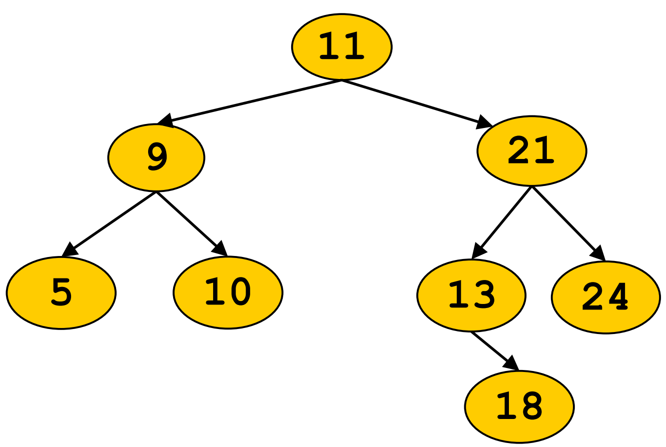
Content of this Lecture

- Trees
- Search Trees
  - Definition
  - Searching
  - Inserting
  - Deleting
- Natural Trees

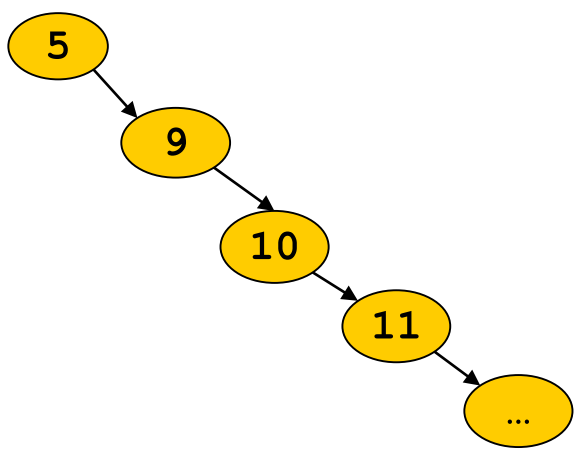
Natural Trees

- A search tree T created by inserting and deleting n keys in random order is called a natural tree
- As any binary tree, it has height(T)∈[n-1, log(n)]
- Height depends on the order in which keys were inserted
- Example

11,9,10,5,21,13,24,18



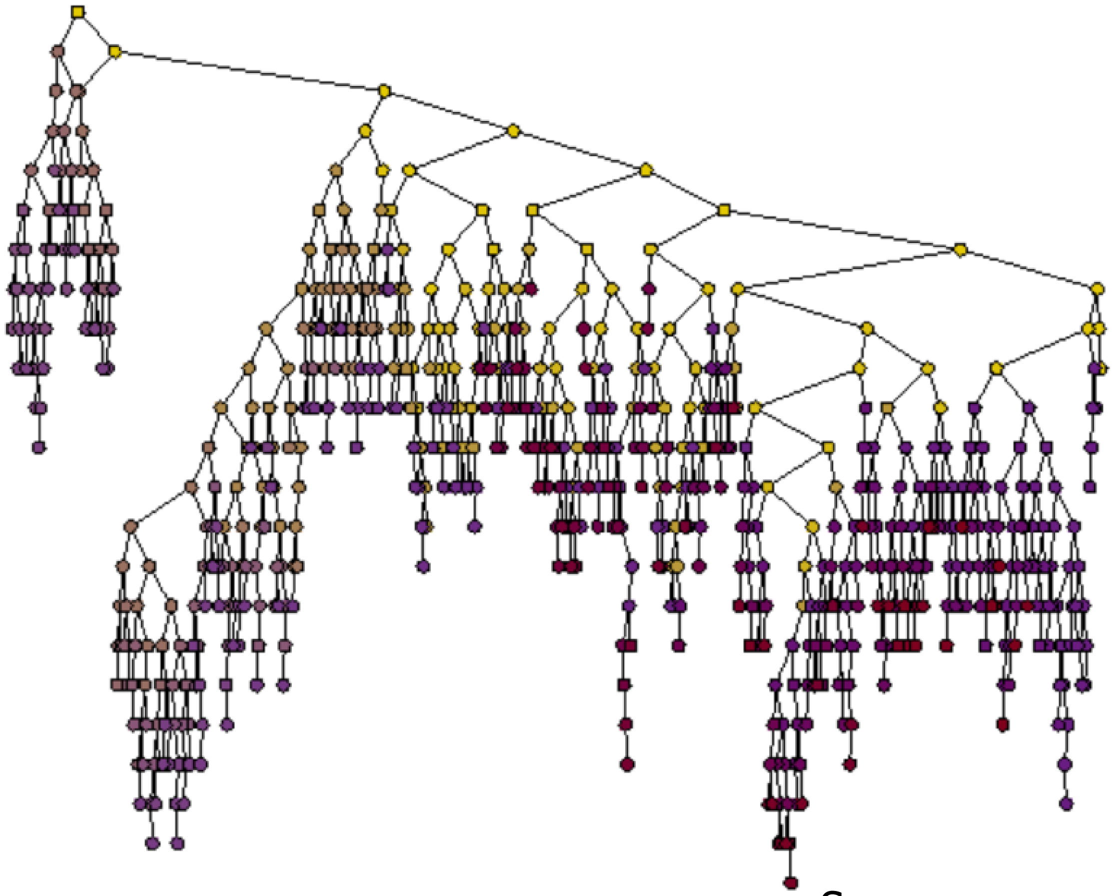
5,9,10,11,13,18,21,24



Average Case

- A natural tree with n nodes has maximal height n-1
- Thus, searching will need O(n) comparisons in worst-case
  - Same for inserting and deleting
- But: Natural trees are not bad on average
  - The average case is O(log(n))
  - More precisely, a natural tree is on average only ~1.4 times deeper than the optimal search tree (with height h~log(n))
  - We skip the proof (argue over all possible orders of inserting n keys), because balanced search trees (AVL trees) are O(log(n)) also in worst-case and are not much harder to implement

Example



Source: cg.scs.carleton.ca/

Exemplary Questions

- Construct a natural search tree from the following input, showing all intermediate steps (I: insert; D: delete): I5, I7, I3, I10, D7, I7, I13, I12, D5
- The worst case complexity for inserting/deleting a key into a search tree with n=|V| nodes is O(n). Give an order of the following operations such that this worst case happens for every operation: I5, I7, I3, I10, D7, I7, I13, I12, D5
- For deleting a given key k in a natural search tree, one may need to find the symmetric predecessor (SP) of a key. Define what a SP is, give an algorithm for finding it (starting from k), and analyze its complexity