Algorithms and Data Structures

(Search) Trees

Marius Kloft

Content of this Lecture

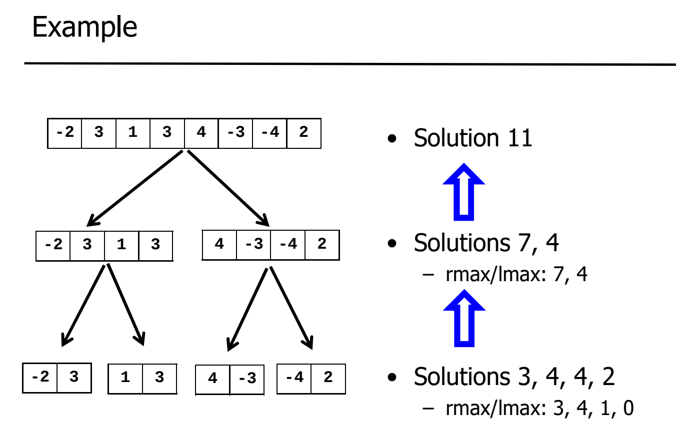
- Trees
- Search Trees
- Natural Trees

Motivation

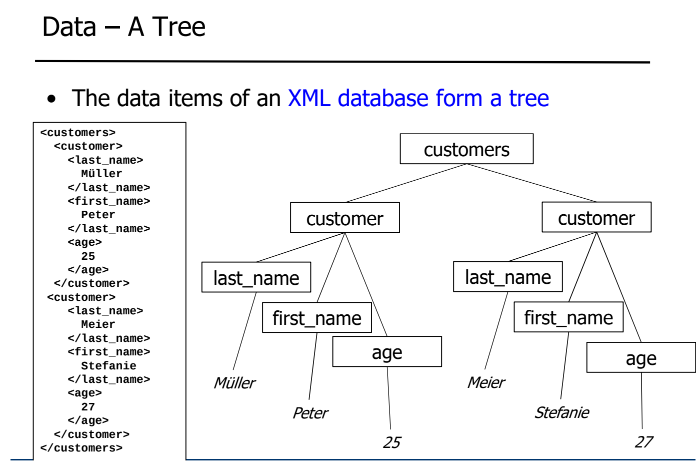
- In a list, (almost) every element has one predecessor / successor
- In a tree, (almost) every element has one predecessor but many successors
- The different successors partition the set of all elements in the subtree
  - Partitions of sets by characteristics of their elements
  - Partitions of sets by order of their elements
  - …
- Trees are everywhere in computer science

Already Seen

- Divide-and-conquer call stack in max-subarray
  - Also: Merge-Sort, QuickSort



- XML
  - depth-first vs breadth-first traversal

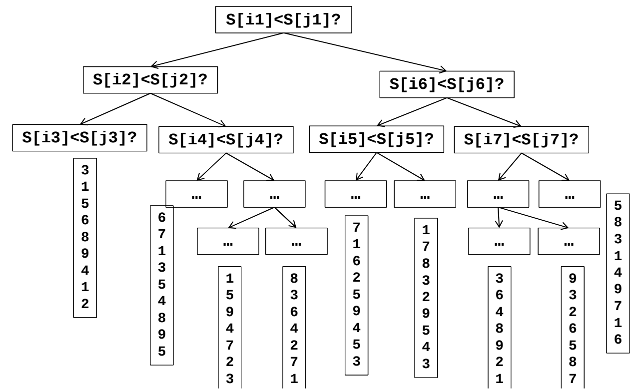


Already Seen?

- Decision trees for proving the lower bound for sorting

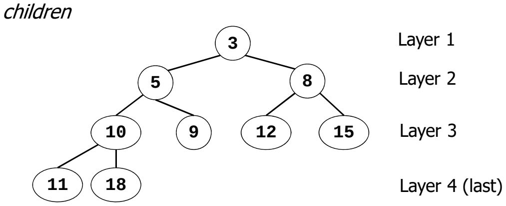
- Heaps for priority queues

Full Decision Tree



Heaps

- Definition
  A heap is a labeled binary tree for which the following holds
  - *Form-constraint (FC): The tree is complete except the last layer*
    - *I.e.: Every node has exactly two children*
  - *Heap-constraint (HC): The value of any node is smaller than that of its children*

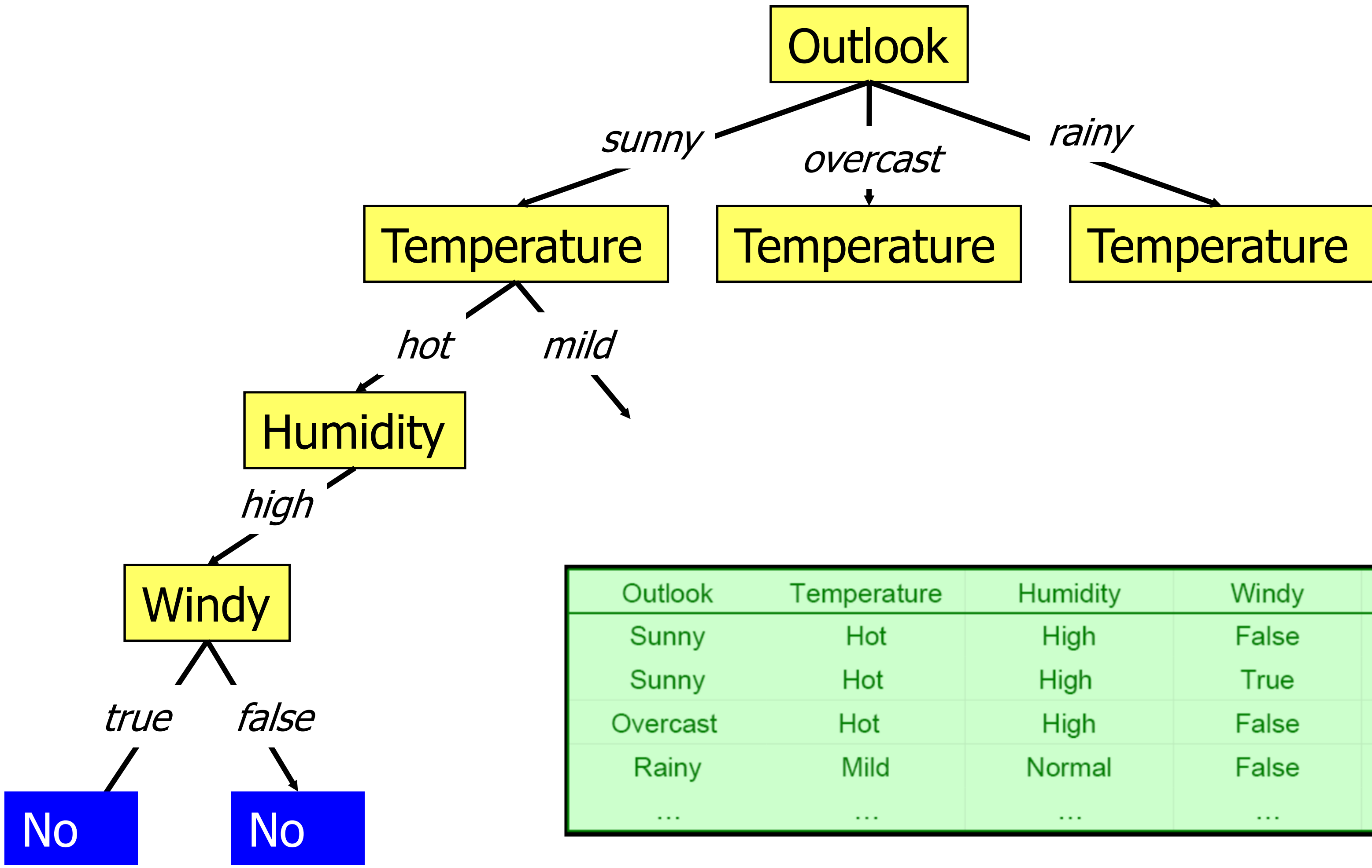


Machine Learning

- Want to go play football?
- Might be canceled – depends on the weather
- Let‵s learn from examples (supervised learning)

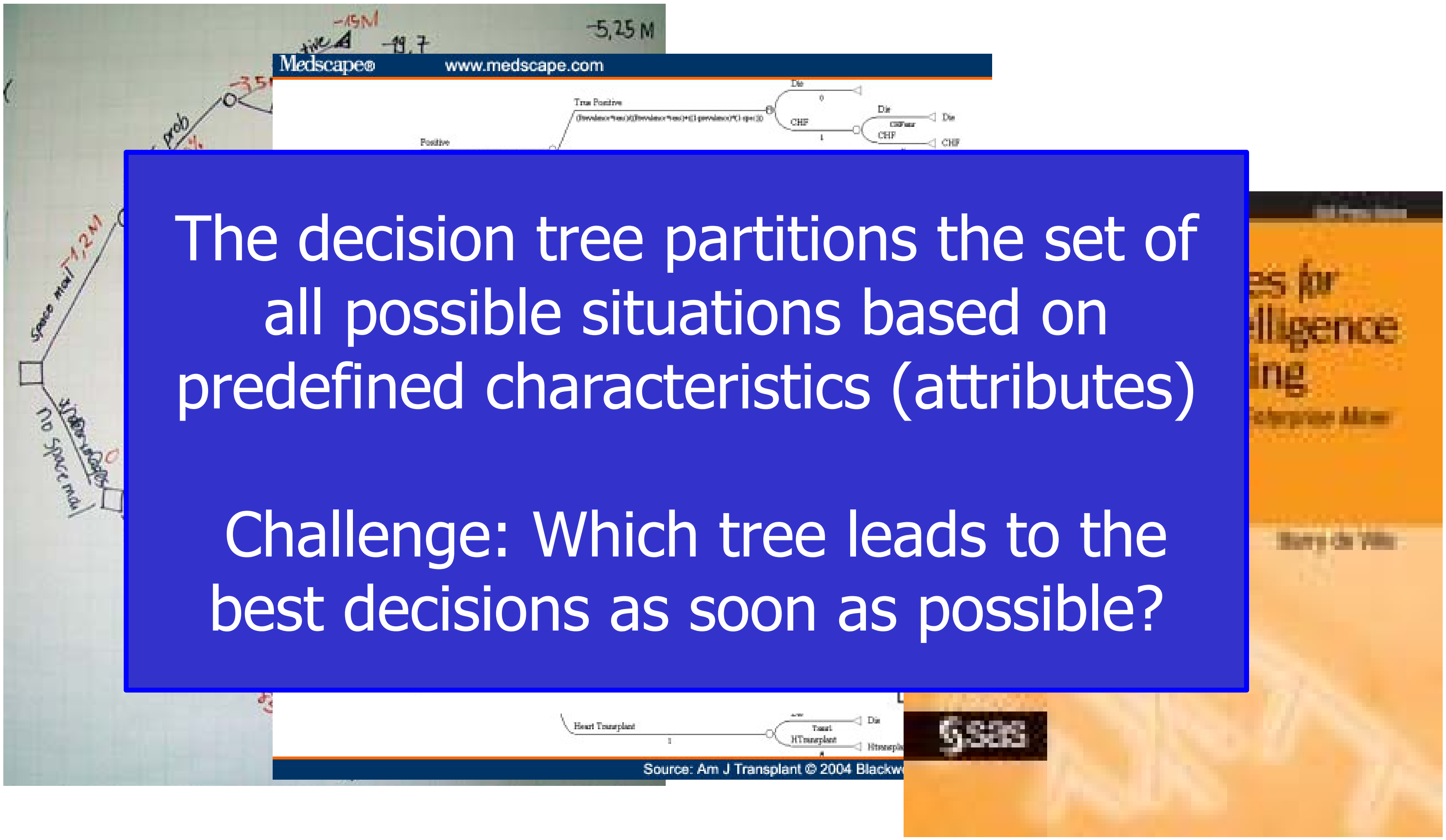
Outlook	Temperature	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	Normal	False	Yes
…	…	…	…	…

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)

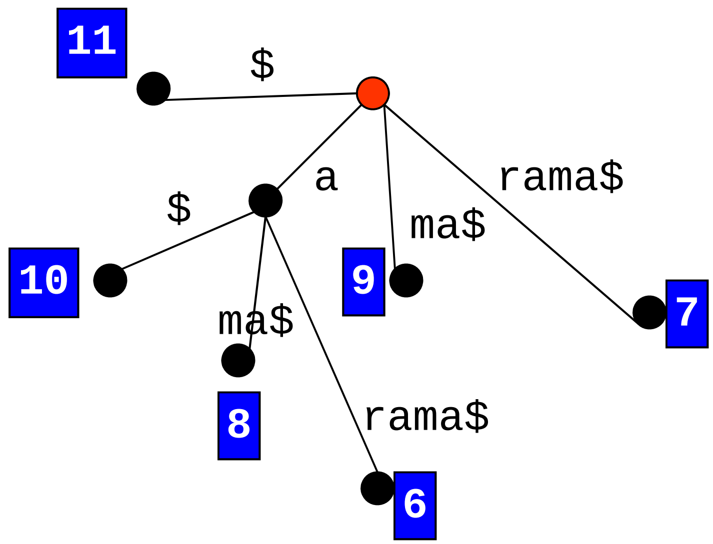
Challenge: Which tree leads to the best decisions as soon as possible?

Suffix-Trees

- Recall the problem of finding all occurrences of a (short) string P in a (long) string T
- Fastest way $O(|P|)$: Suffix Trees
- Look 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

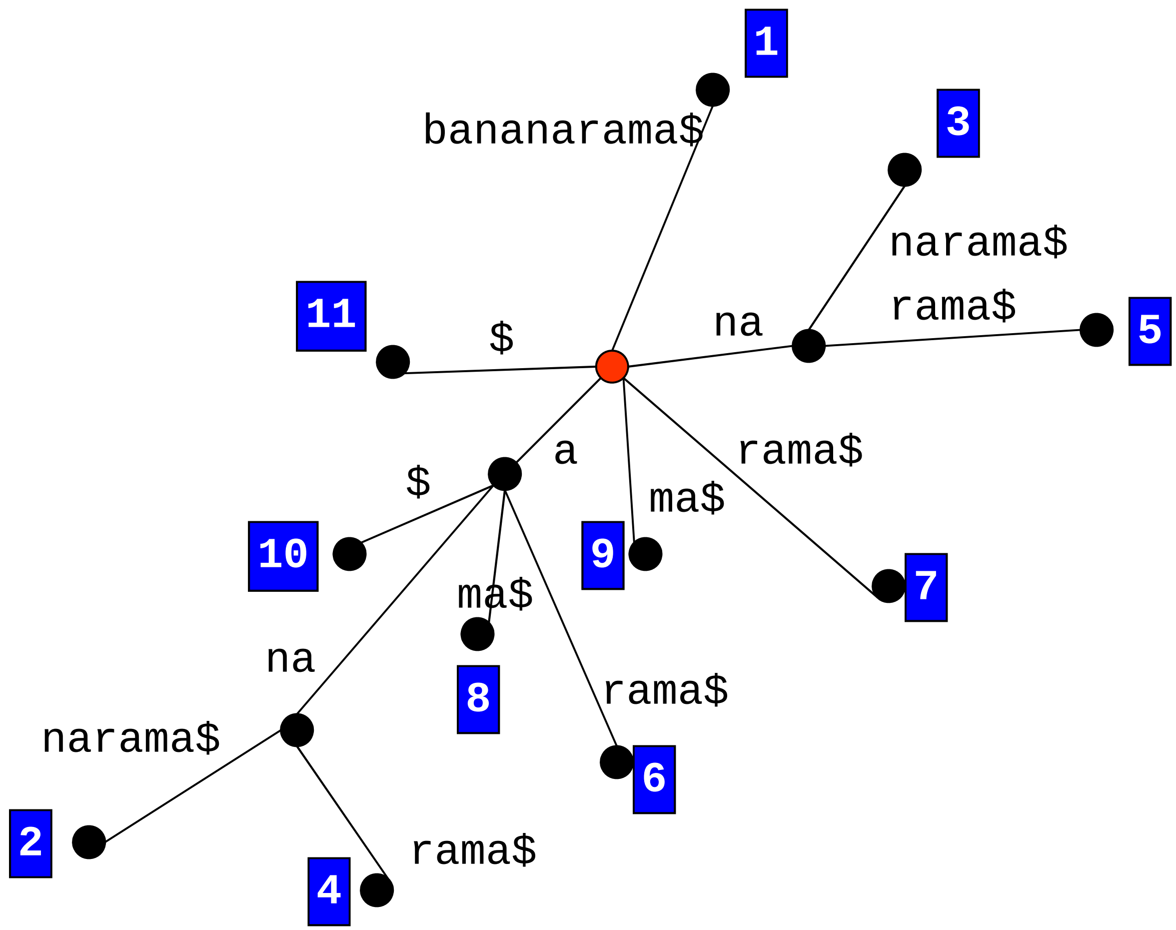
Example

```
1234567891011
BANANARAMA $
 ANANARAMA $
  NANARAMA $
   ANARAMA $
    NARAMA $
     ARAMA $
      RAMA $
       AMA $
        MA $
         A $
           $
```



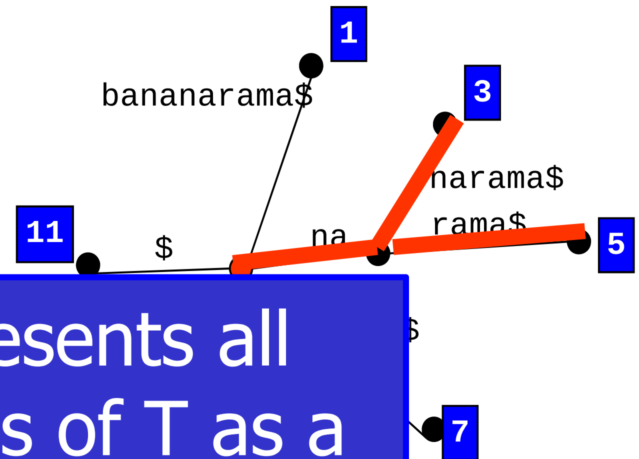
Example

```
1234567891011
BANANARAMA $
 ANANARAMA $
  NANARAMA $
   ANARAMA $
    NARAMA $
     ARAMA $
      RAMA $
       AMA $
        MA $
         A $
           $
```



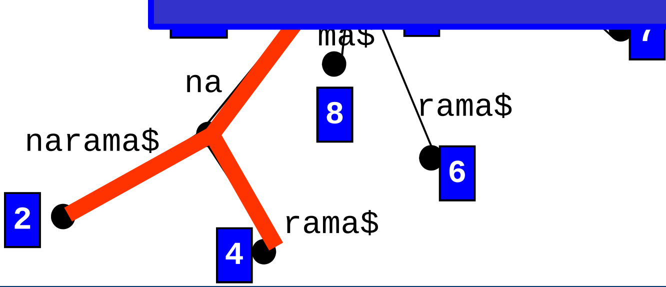
Searching in the Suffix Tree

P = „na“



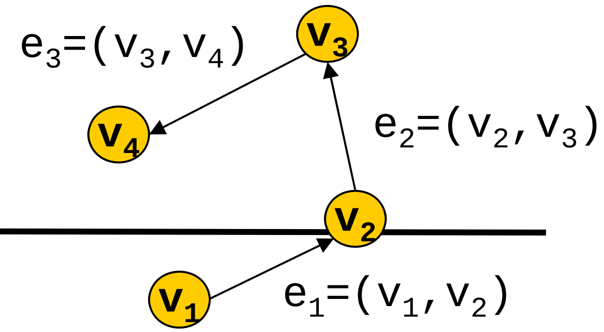
The suffix tree for T represents all common prefixes of suffixes of T as a unique path from root.

Challenge: Construction of a suffix tree in linear time.



P = „an“

Graphs



- Definition

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

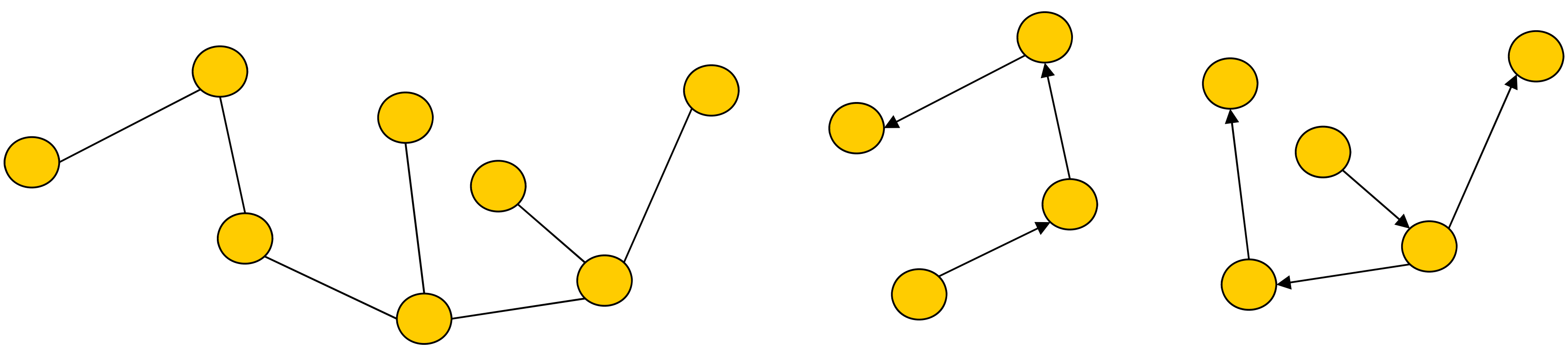
  - *A sequence of edges e_1, e_2, .., e_n is called a path iff* $\forall 1 \leq i < n-1$: *e_i=(v', v) and e_{i+1}=(v, v''); the length of this path is n*
  - *A path (v_1,v_2), (v_2,v_3), ..., (v_{n-1},v_n) is acyclic iff all v_i are different*
  - *G is connected if every pair v_i, v_j is connected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

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

  - *Every edge (v,v')∈E is called outgoing for v*
  - *Every edge (v',v)∈E is called incoming for v*

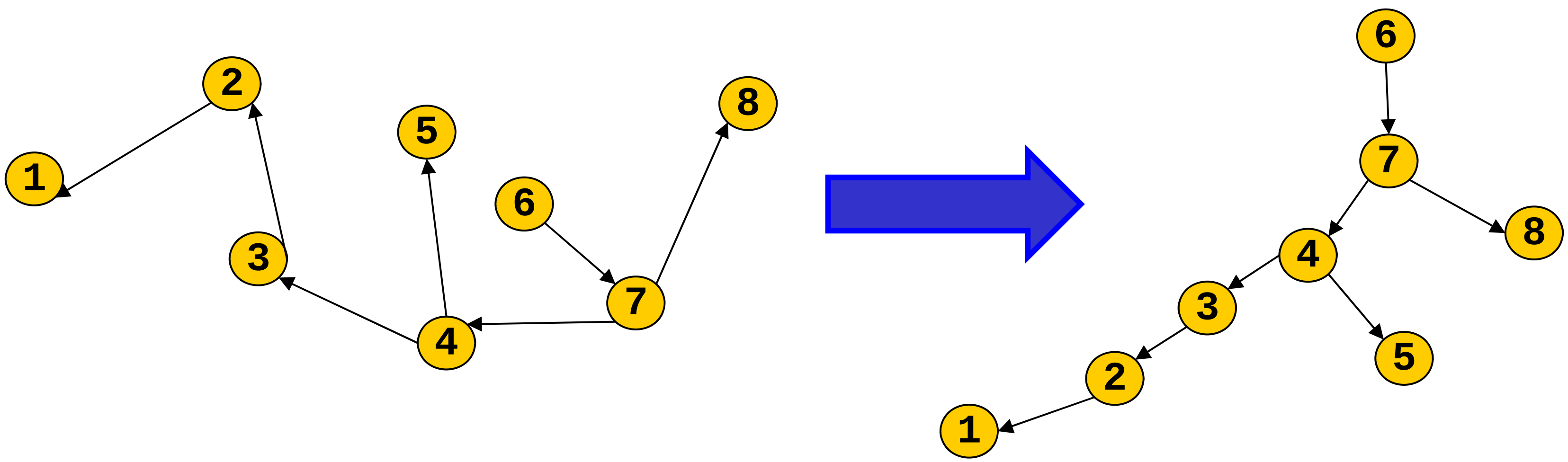
Trees as Connected Graphs

- Definition
  - *A undirected connected acyclic graph is called a undirected tree*
  - *A directed connected acyclic graph in which every node has at most one incoming edge is called a directed tree*
- Lemma
  - *In a undirected tree, there exists exactly one path between any pair of nodes*

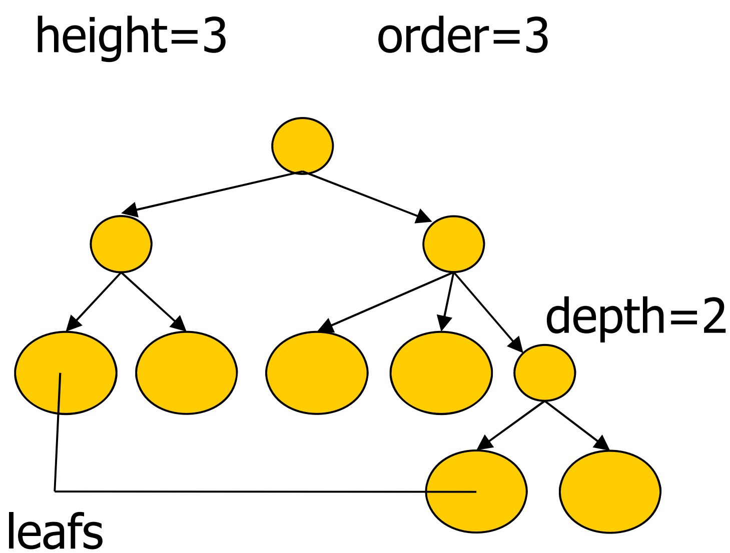


Rooted Trees

- Definition
  A directed tree with exactly one vertex v with no incoming edges is called a rooted tree; v is called the root of the tree
- From now on: "Tree" means a directed, rooted tree
- Lemma
  - *In a directed rooted tree, there exists exactly one path between root and any other node*

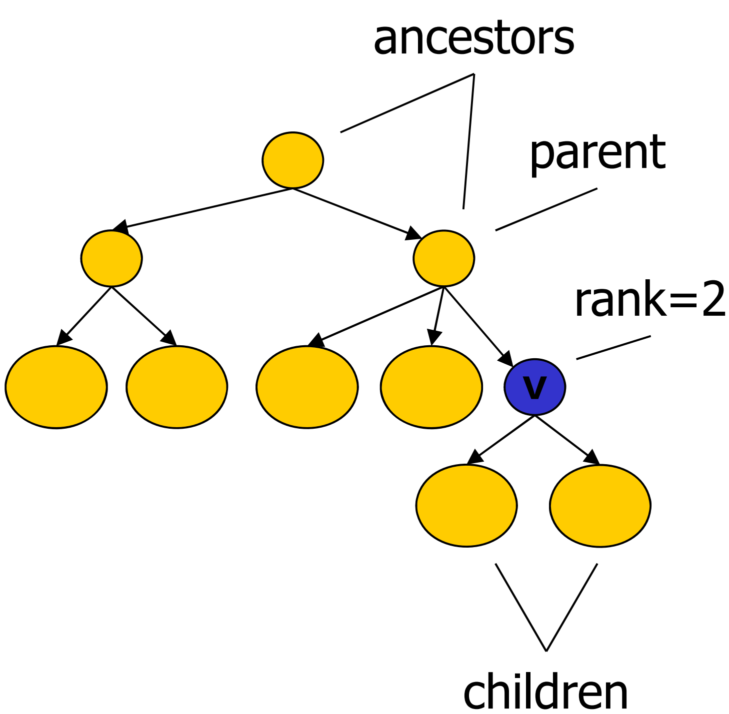


Terminology



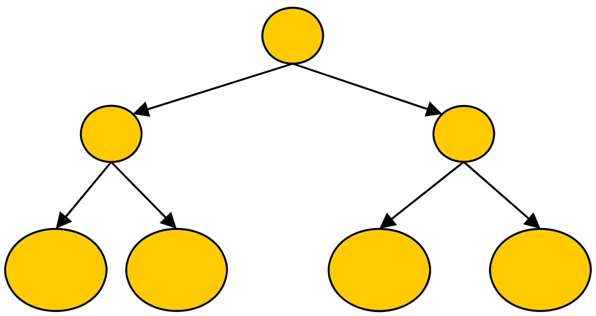
- Definition
  Let T be a tree. Then ...
  - *A node with no outgoing edge is a leaf; other nodes are inner nodes*
  - *The depth of a node p is the length of the (only) path from root to p*
  - *The height of T is the depth of its deepest leaf*
  - *The order of T is the maximal number of children of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

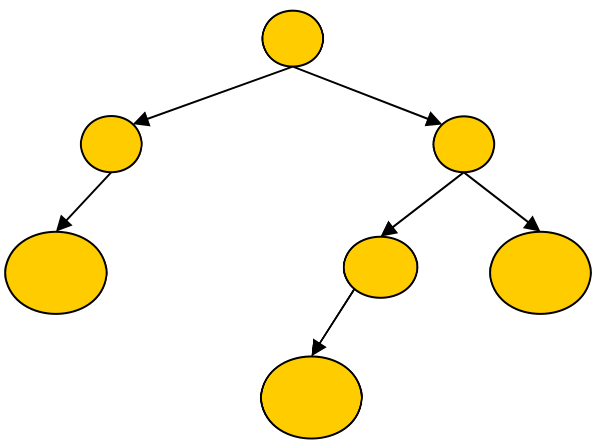
More Terminology



- Definition
  Let T be a tree and v a node of T. Then …
  - *All nodes incident to an outgoing edge of v are its children*
  - *v is called the parent of all its children*
  - *All nodes on the path from root to 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

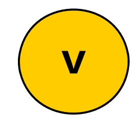


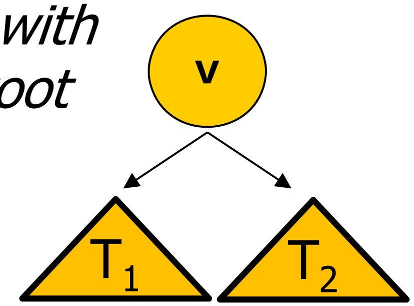


- Definition
  Let T be a directed tree of order k. T is complete if all its inner nodes have rank k and all leaves have the same depth
- In this lecture, we will mostly consider rooted ordered trees of order two (binary trees)

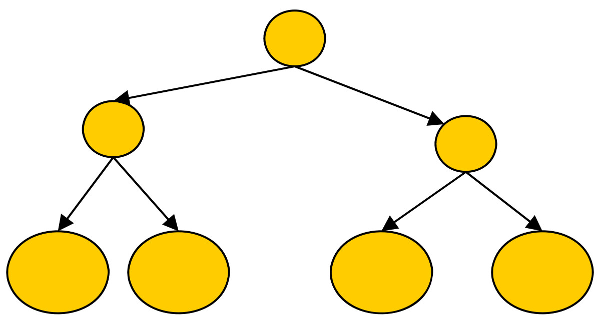
Recursive Definition of Trees

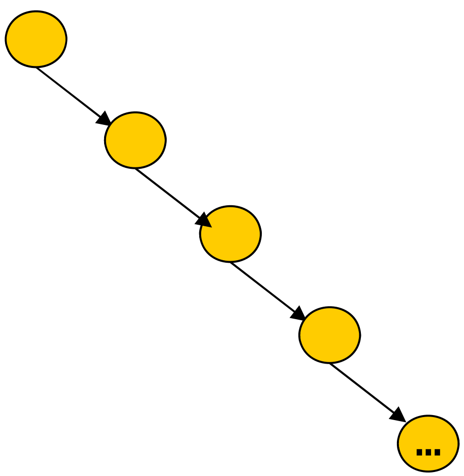
- We defined trees as graphs with certain constraints
- Will mostly traverse trees using recursive functions
- The relationship may become clearer when using a recursive definition of (binary) trees
- Definition
  A tree is a structure defined as follows:
  a) *A single node v and an empty set is a tree with height 0*
  b) *If T_1 and T_2 are (possible empty) 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 with* $height = \max(height(T_1), height(T_2)) + 1$*; v is its root*





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,* $height(T) \in [floor(\log(|V|)), |V|-1]$

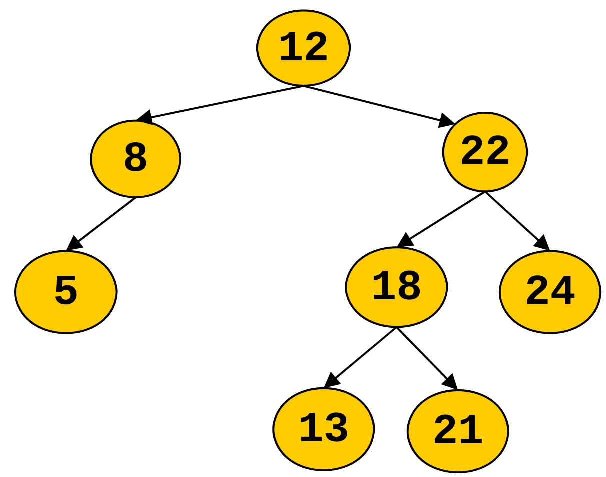
Content of this Lecture

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

Search Trees

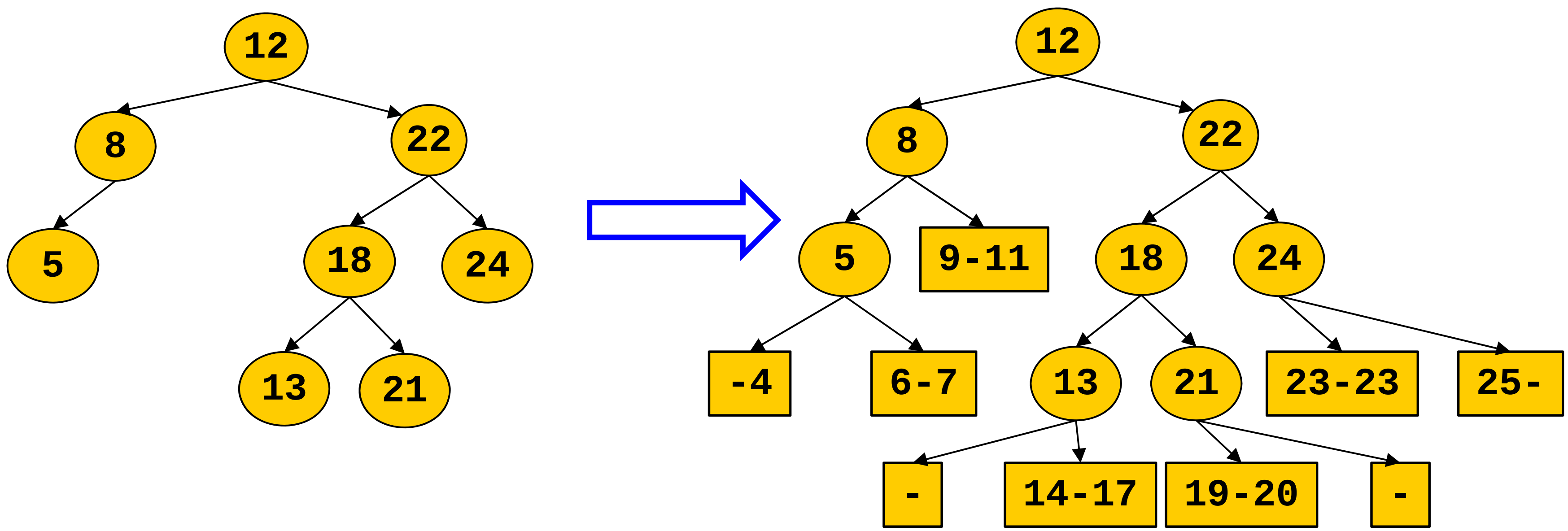
- Definition

  A search tree T=(V,E) is a rooted binary tree with n=|V| differently key-labeled nodes such that $\forall v \in V$:
  - *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
  - We will use integer labels
  - "node" ~ "label of a node"
  - We only consider trees without duplicate keys (easy to adapt)
  - Search trees are used to manage and search a list of keys
  - Operations: search, insert, delete



Complete Trees

- Conceptually, we 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 (but none of its values is a key)



What For?

- For a search tree T=(V,E), we will reach $O\big(height(T)\big)$ for testing whether k∈T.

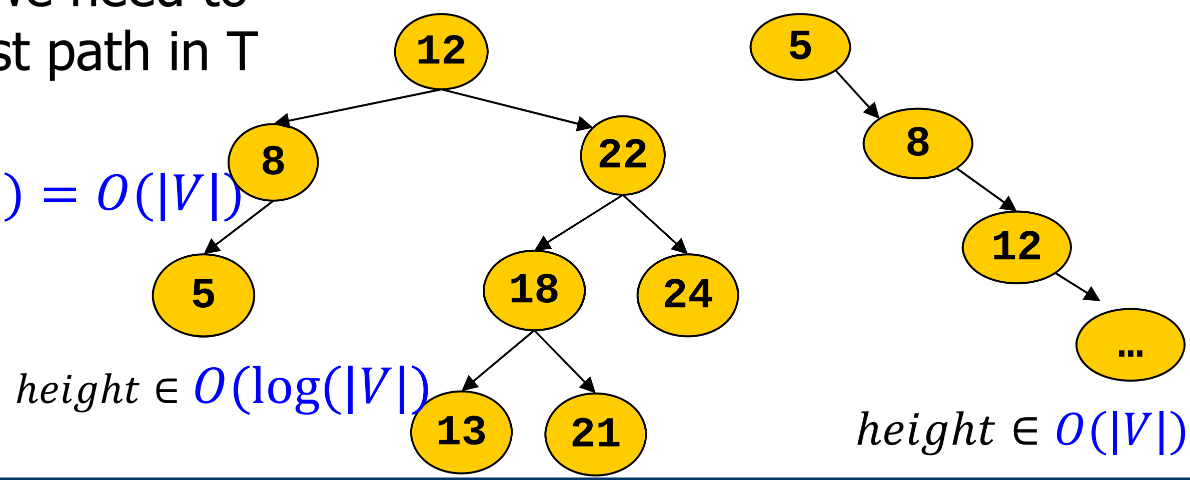
$$height(T) \in [floor(\log(|V|)), |V| - 1]$$

- Compared to binsearch, search trees are a dynamically growing / shrinking data structure
  - But need to store pointers

Searching

- Straight-forward
  - Comparing the search key to a node determines whether we have to look into the left or into the right subtree
  - 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: $O(height(T)) = O(|V|)$
  - 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;
}
```

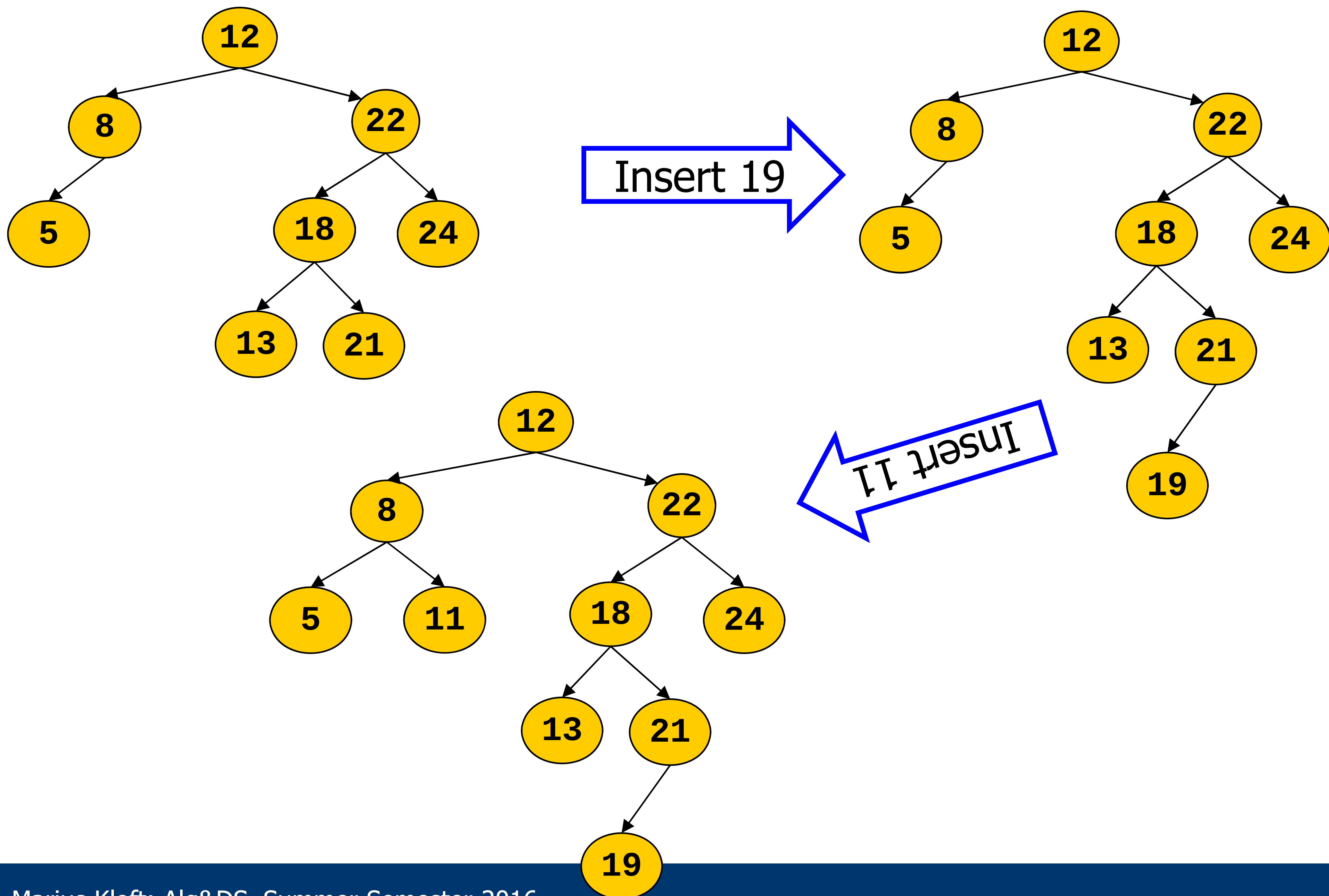


Insertion

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

- We 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
- This creates a new search tree which contains k
- Complexity: Same as search

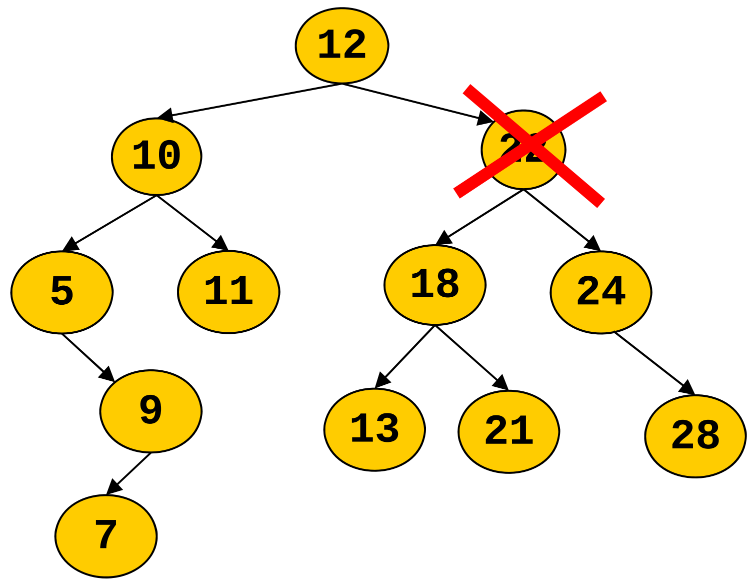
Example



Deletion

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

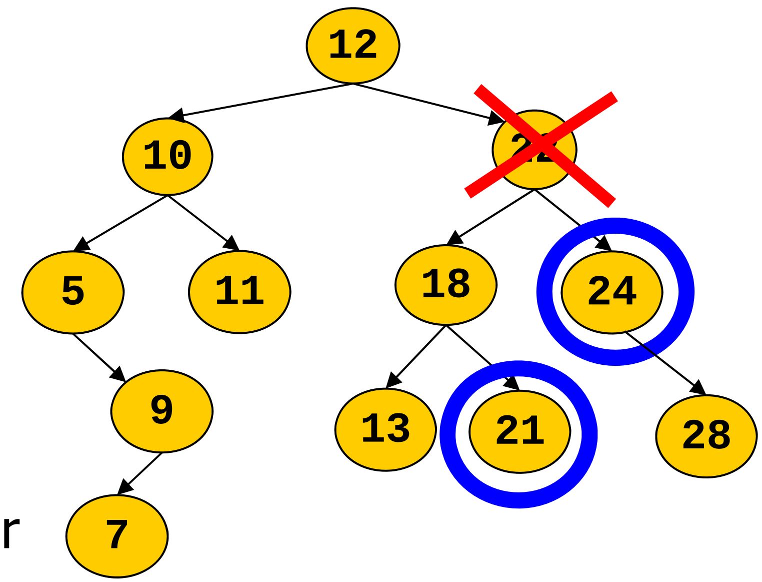
Observations



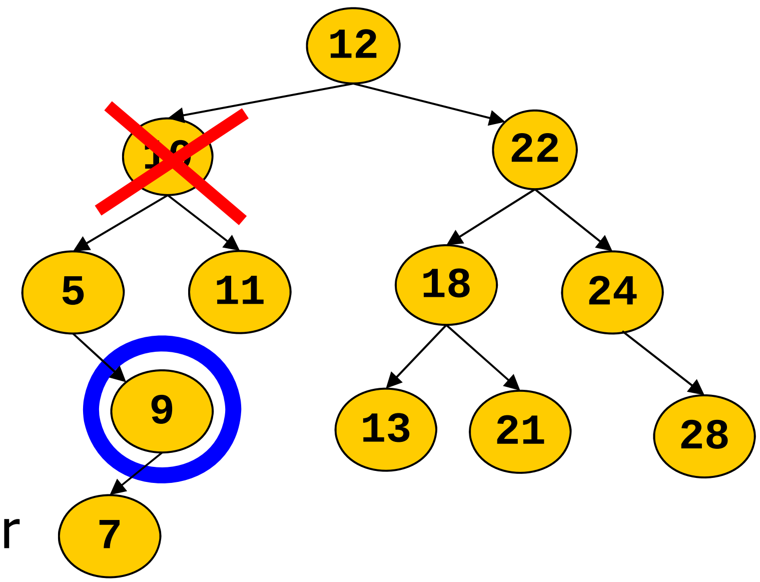
- We cannot remove q, but we can replace the label of q with another label - and remove this node
- We need a node q' which can be removed and whose label k' can replace k without hurting the search tree constraints SC
  - label(k')>max(label(left_child(k')), label(successors(left_child(k')))
  - label(k')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

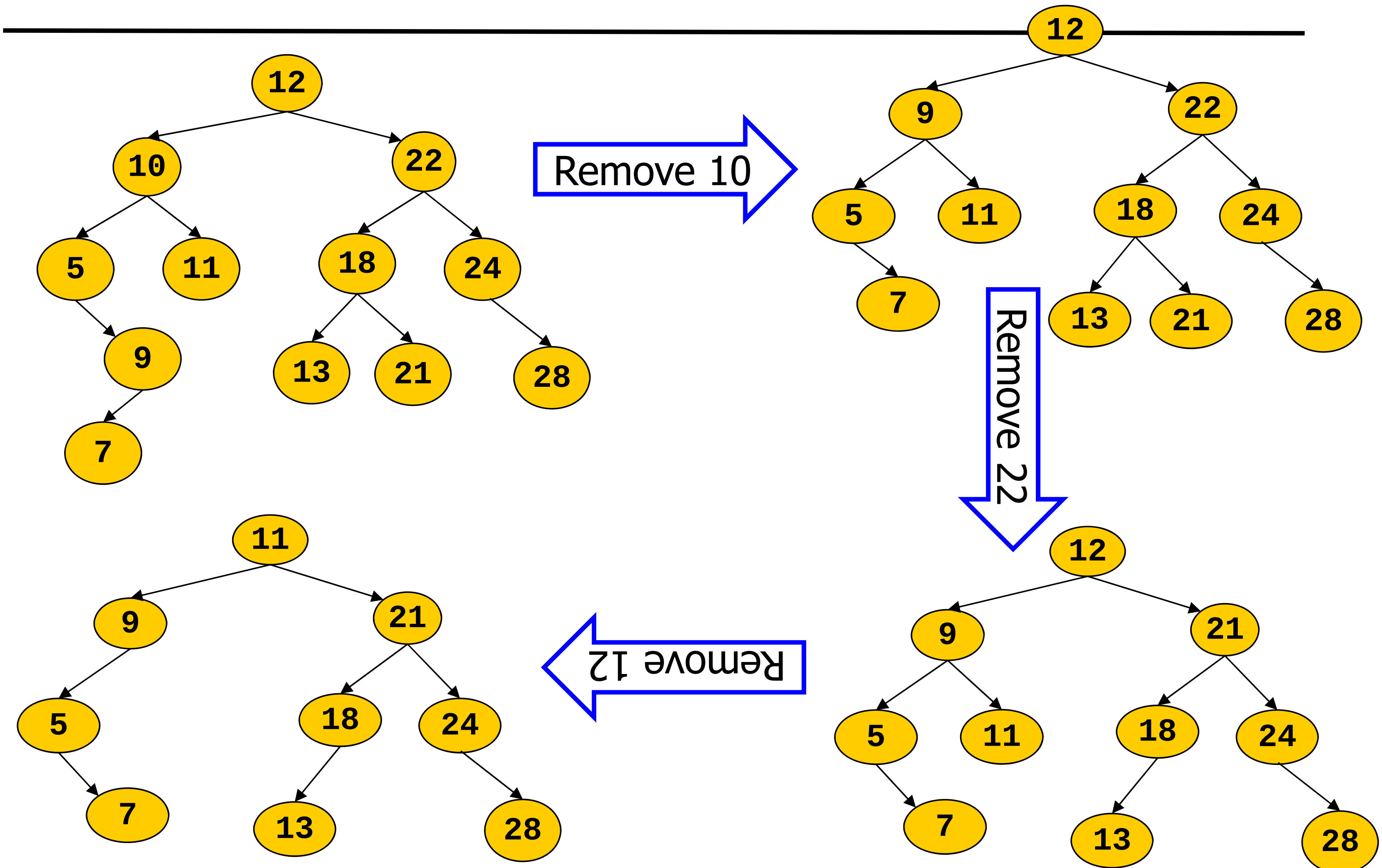


Observations



- 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 - can be removed

Example



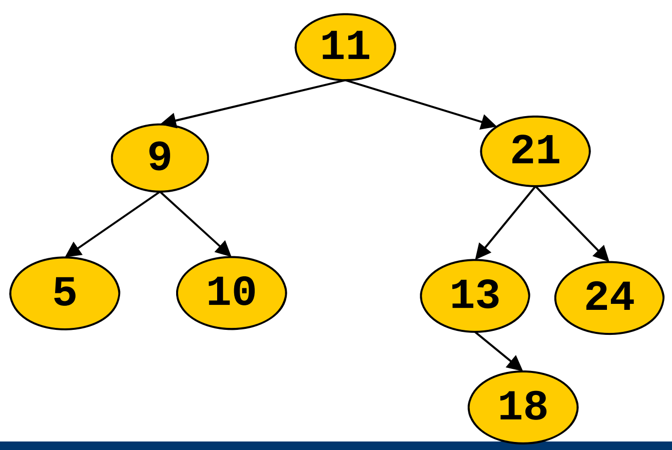
Content of this Lecture

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

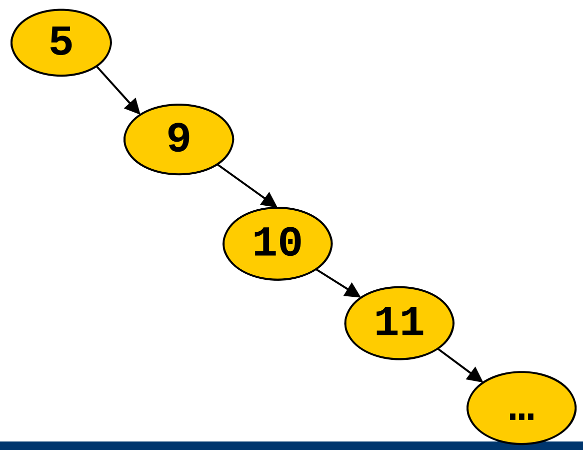
Natural Trees

- A search tree created by inserting and deleting keys in arbitrary order is called a natural tree
- A natural tree T=(V,E) has $height(T) \in [|V|-1, \log(|V|)]$
- The concrete height for a set of keys depends on the order in which keys were inserted
- Example

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



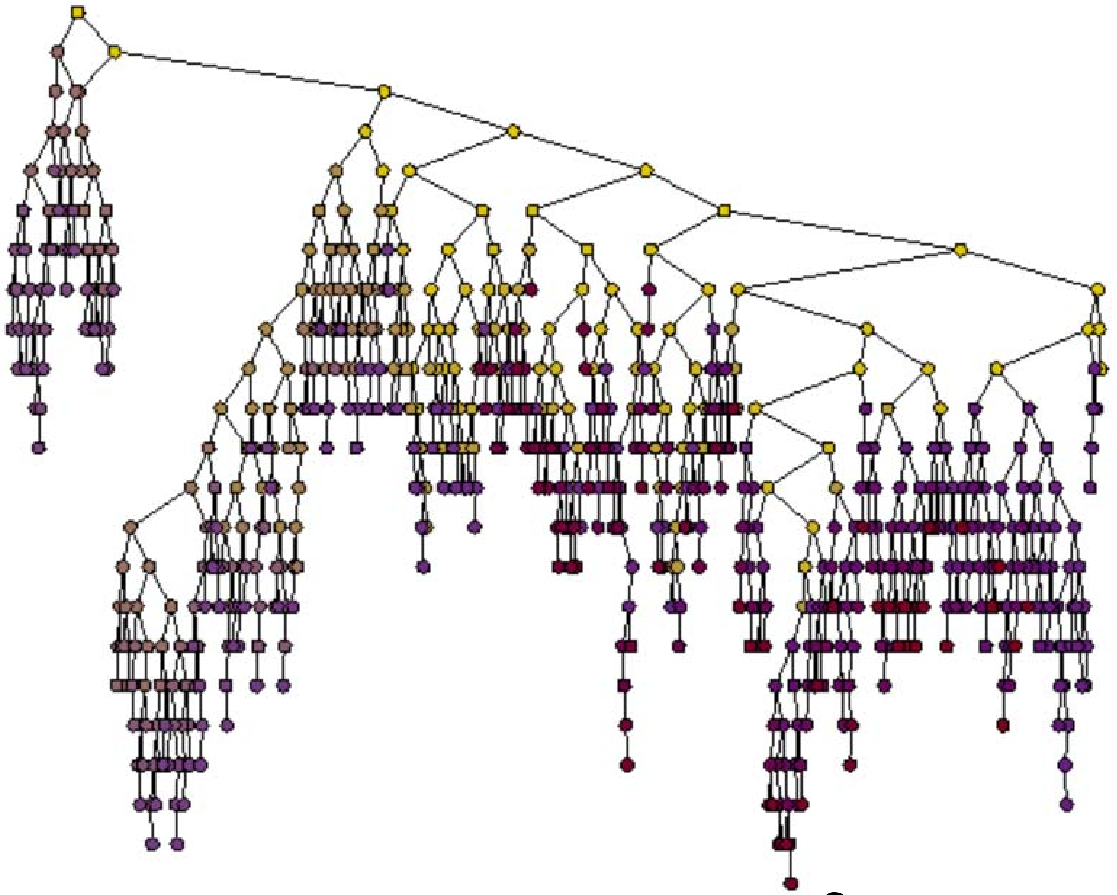
5,9,10,11,13,18,21,24



Average Case Analysis

- We have seen that a natural tree with n nodes has a maximal height of n-1
- Thus, searching will need O(n) comparisons in worst-case
- Nevertheless, 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 larger than the optimal search tree (with height O(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/

Sorted probe sequences (revisited)

- Consider a hash table A and a hash function for which h(k) = h(k’)
- when searching for k’, follow the probe sequence
  - first position: i = A[h(k’)]
  - next position: i = i – s(k,1), because s(k,j) – s(k, j-1) = s(k, 1)
  - if A[h(k’)] > k’ we can abort, all others in the probe sequence will be larger as well
  - gives same complexity for positive and negative searches
  - Example (this was messed up)
    - h(12) == h(5)
    - search for k = 5
    - A[h(5)] = 12 → abort search, all others will be larger than 5, $k \notin A$

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
- For deleting a given key k in a natural search tree, one sometimes needs a symmetric predecessor (SP) of a key. Define what a SP is, give an algorithm for finding it(starting from k), and analyze ist complexity
- Construct an AVL-tree from the following input, showing all intermediate steps (I: insert; D: delete): I5, I7, I3, I10, D7, I7, I13, I12, D5