Trees

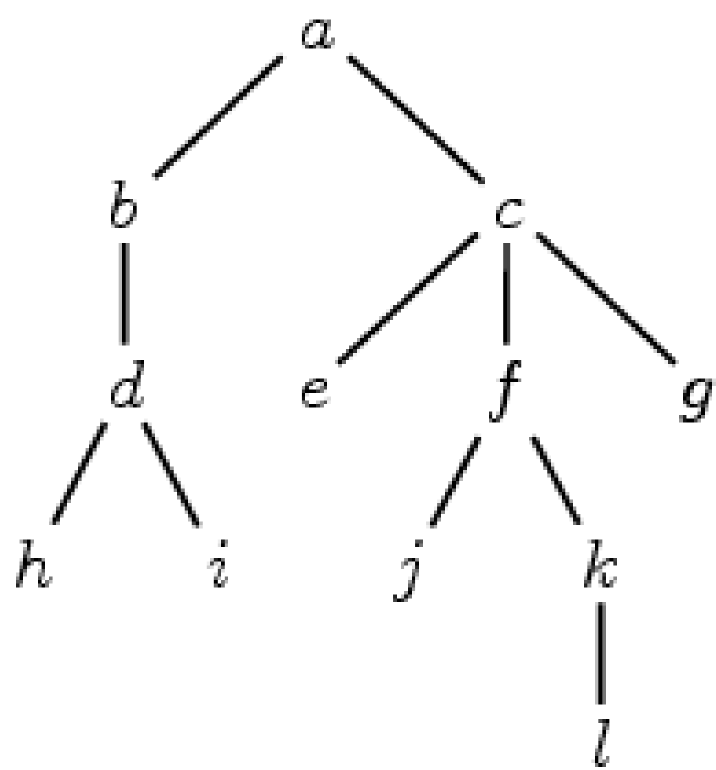
Terminology. Rooted Trees. Traversals. Labeled Trees and Expression Trees. Tree ADT. Tree Implementations. Binary Search Trees. Optimal Search Trees

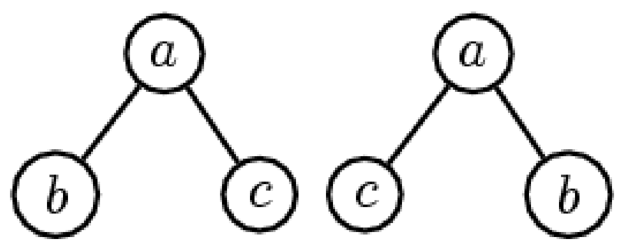
Trees

- Rooted tree: collection of elements called nodes, one of which is distinguished as root, along with a relation ("parenthood") that imposes a hierarchical structure on the nodes
- Formal definiton:
  - A single node by itself = tree. This node is also the root of the tree
  - Assume n = node and T_1, T_2, ..., T_k = trees with roots n_1, n_2, ..., n_k:
  - construct a new tree by making n be the parent of nodes n_1, n_2, ..., n_k
- Common data structure for non-linear collections.

Terminology for rooted trees

- ancestors, descendants, parent, children,
- leaves (vertices with no children),
- internal vertices (vertices with children)
  - “root” is an internal vertex
- path
- subtrees
- order of nodes, siblings,



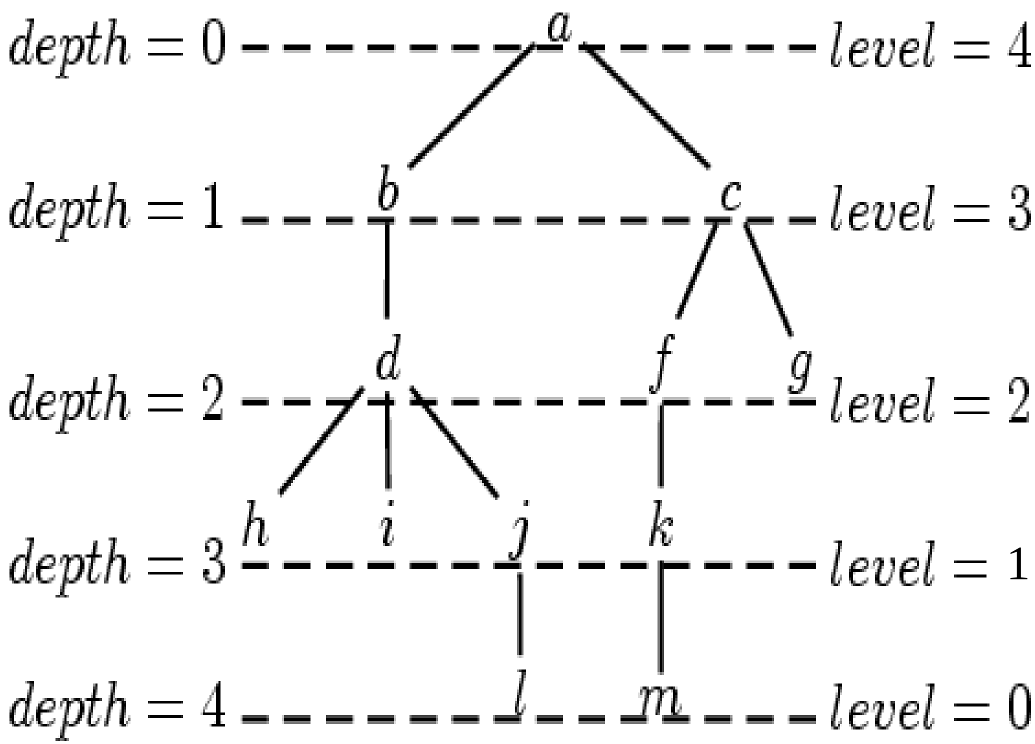


Terminology for rooted trees

- For a rooted tree T = (V, E) with root r $\in$V:
  - Path: $\langle n_1, n_2, ..., n_k \rangle$ such that n_i = parent n_{i+1} for $1 \leq i \leq k$. *length*(path): no. of nodes -1
  - The depth of a vertex $v \in V$ is depth(v) = the length of the path from *r* to *v*
  - The height of a vertex $v \in V$ is height(v) = the length of the longest path from v to a leaf
  - The height of the tree *T* is height(*T*) = height(*r*)
  - The level of a vertex v $\in$V is level(v) = height(*T*) − depth(*v*)
  - The subtree generated by a vertex $v \in V$ is a tree consisting of root *v* and all its descendants in *T*.

Terminology for rooted trees. Example

- For the tree on the right…
  - The root is *a*.
  - The leaves are *h*, *g*, *i*, *l*, *m*.
  - The proper ancestors of *k* are *a*, *c*, *f*.
  - The proper descendants of *d* are *h, i, j, l*.
  - The parent of *h* is *d*.
  - The children of *c* are *f*, *g*.
  - The siblings of *h* are *i*, *j*.
  - Height(*T*)=height(*a*)=4



Terminology for rooted trees

- A rooted tree is said to be:
  - m-ary if each internal vertex has at most m children.
    - $m = 2 \rightarrow$ binary; $m = 3 \rightarrow$ ternary
  - full m-ary if each internal vertex has exactly m children.
  - complete m-ary if it is full and all leaves are at level 0.
- Some limits:
  - Maximum height for a tree with n vertices is $n - 1$.
  - Maximum height for a full binary tree with n vertices is $(n - 1)/2$
  - The minimum height for a binary tree with n vertices is $\lfloor \log_2 n \rfloor$

Traversals

```
PREORDER(n)
list n
for each child c of n, if any, in order from the left
    do PREORDER(c)
```

```
INORDER(n)
if n is a leaf
  then list n
  else  INORDER(leftmost child of n)
        list n
        for each child c of n, except for the leftmost, in order from the left
            do INORDER(c)
```

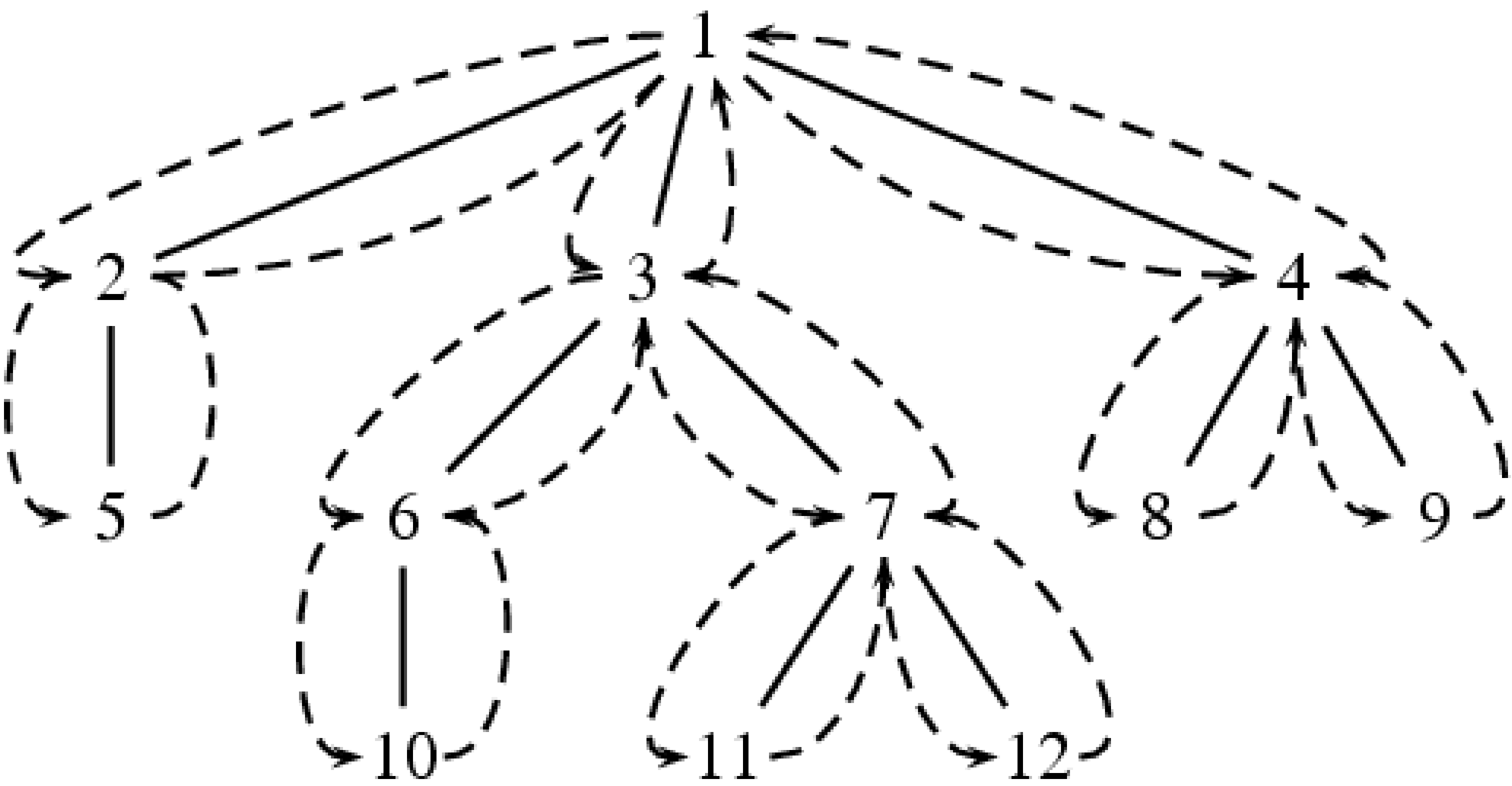
Ancestral information

- Traversals in preorder and postorder are useful for obtaining ancestral information. Suppose
  - post(n) is the position of node n in a postorder listing of the nodes of a tree, and
  - desc(n) is the number of proper descendants of node n. Then
  - nodes in the subtree with root n are numbered consecutively from post(n)-desc(n) to post(n)
- To test if a node x is a descendant of node y:

$$\text{post}(y) - \text{desc}(y) \leq \text{post}(x) \leq \text{post}(y)$$

- Similar relationship for preorder

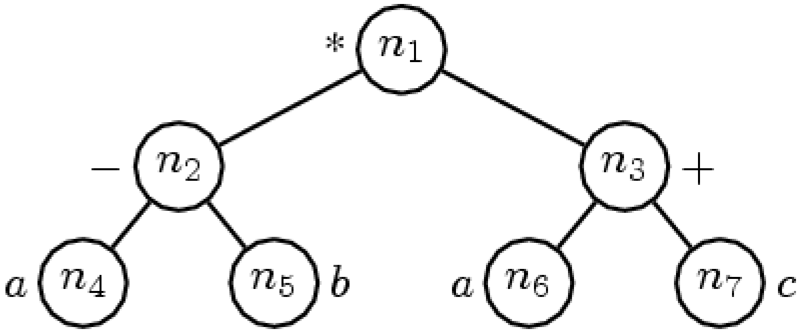
Traversal example



- preorder: 1, 2, 5, 3, 6, 10, 7, 11, 12, 4, 8, 9.
- postorder: 5, 2, 10, 6, 11, 12, 7, 3, 8, 9, 4, 1.
- inorder: 5, 2, 1, 10, 6, 3, 11, 7, 12, 8, 4, 9.

Labelled trees and expression trees

- Binary trees can be used to represent expressions such as
  - compound propositions,
  - combinations of sets, and
  - arithmetic expressions.

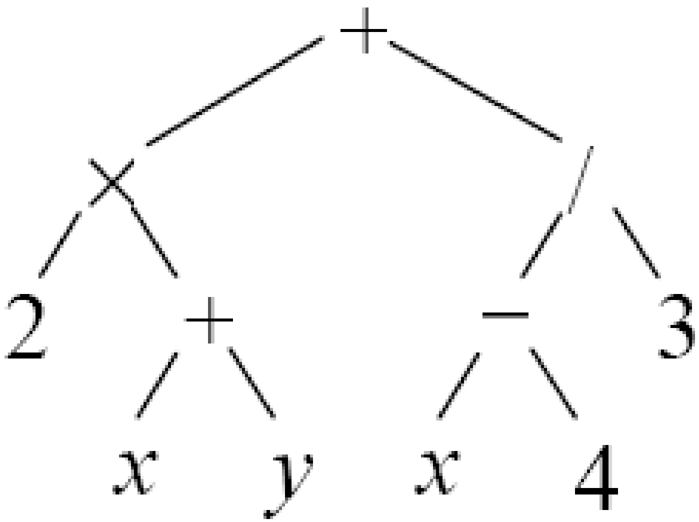


- Labelled tree:*Label* (*value*) associated with each node
- Expression tree: The internal vertices represent operators, and the leaves represent operands.
  - binary operator : 1st operand on the left leaf
    2nd operand on the right leaf
  - unary operator : single operand on the right leaf

Prefix, postfix, infix form

- From the binary trees, we can obtain expressions in three forms:
  - infix form:
    - use in-order traversal
    - parentheses are needed to avoid ambiguity
  - prefix form / Polish notation:
    - use pre-order traversal
    - no parentheses are needed
  - postfix form / reverse Polish notation:
    - use postorder traversal
    - no parentheses are needed
- Prefix and postfix expressions are used extensively in computer science.

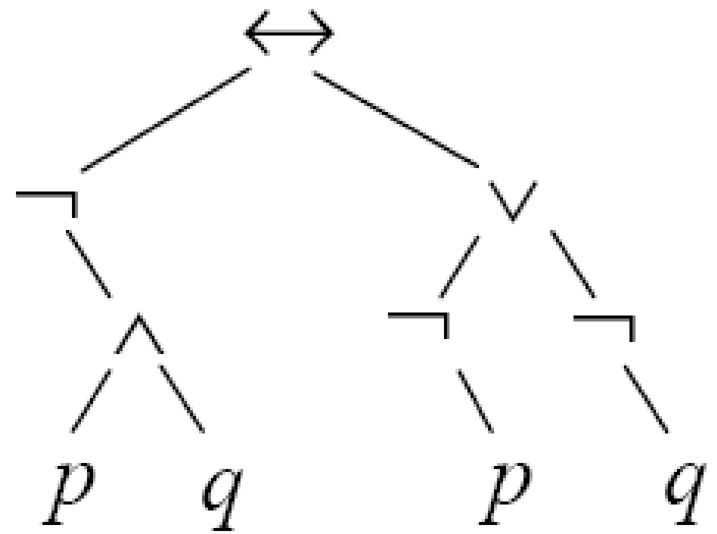
Expression tree examples



infix: $(2 \times (x + y)) + ((x - 4) / 3)$

prefix: $+ \times 2 + x\ y\ / - x\ 4\ 3$

postfix: $2\ x\ y + \times x\ 4 - 3\ / +$



infix: $(\neg(p \wedge q)) \leftrightarrow (\neg p \vee \neg q)$

prefix: $\leftrightarrow \neg \wedge p\ q \vee \neg p \neg q$

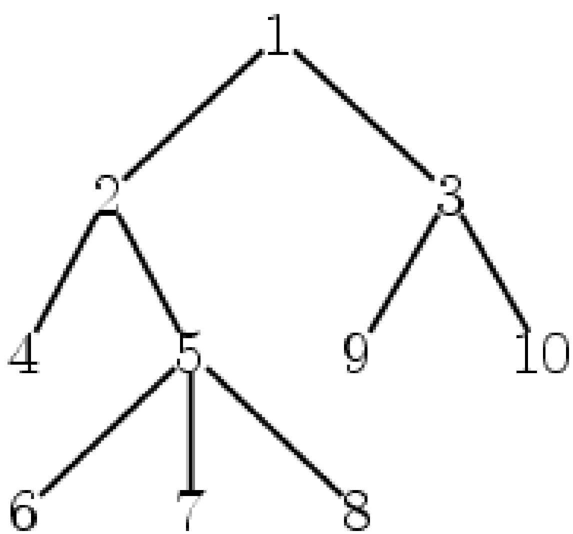
postfix: $p\ q \wedge \neg p \neg q \neg \vee \leftrightarrow$

ADT Tree

- ADT that supports the folowing operations:
  - parent(*n, T*): returns parent of node *n* in tree *T.* For root returns *null tree* (denoted Λ)*.*
    - Input: node, tree; Output: node or Λ
  - leftmostChild(*n, T*): returns the leftmost child of node *n* in tree *T* or Λ for a leaf
    - Input: node, tree; Output: node or Λ
  - rightSibling(*n, T*): returns the right sibling of node *n* in tree *T* or Λ for the rightmost sibling
    - Input: node, tree; Output: node or Λ
  - label(*n, T*): returns the label (associated value) of node *n* in tree *T*
    - Input: node, tree; Output: label
  - root(*T*): returns the root of *T*
    - Input: tree; Output: node or Λ
- Support operations may also be defined

Implementations of trees. A vector

- Example
  - Supports only “parent” operation



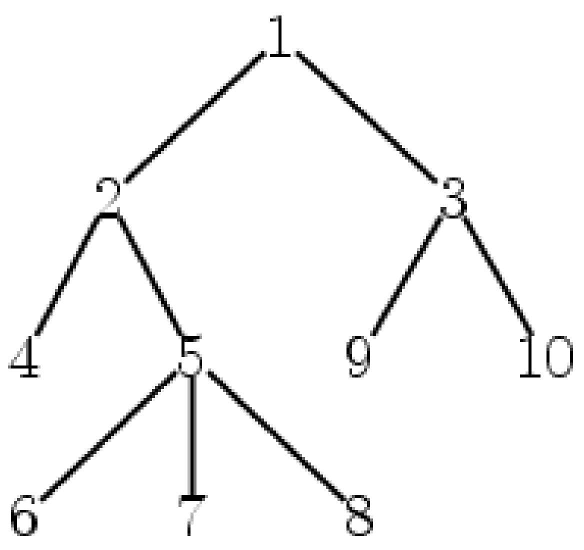
(a)

1	2	3	4	5	6	7	8	9	10
0	1	1	2	2	5	5	5	3	3

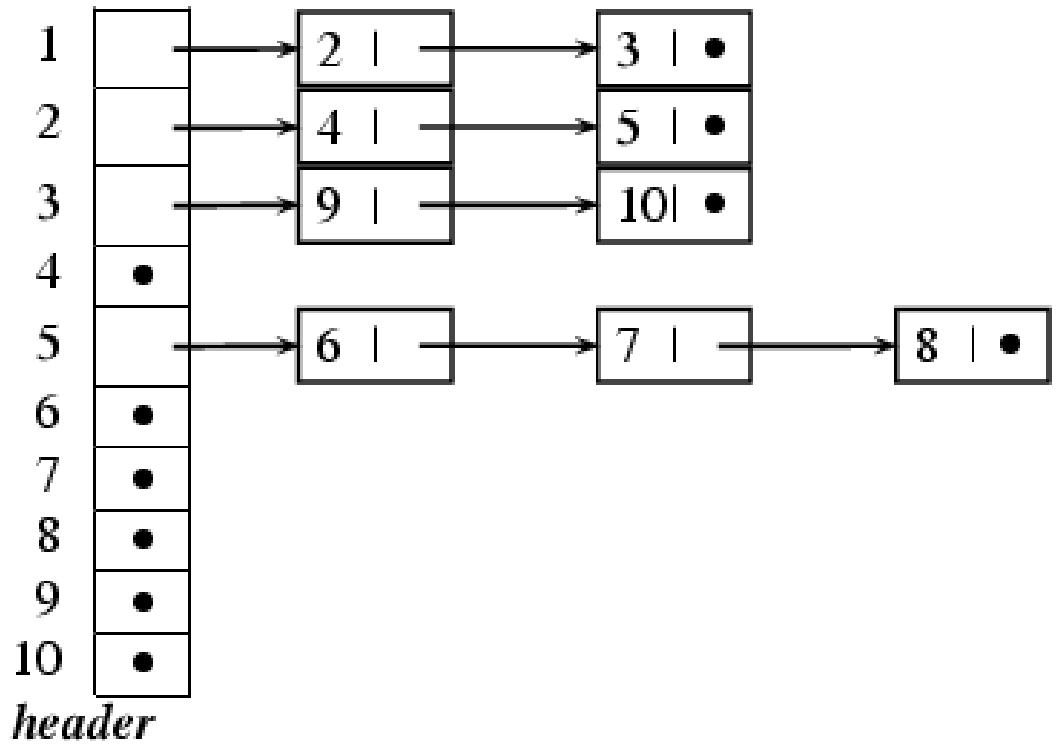
(b)

(*a*) Tree (*b*) Data structure

Implementations of trees. Lists of children

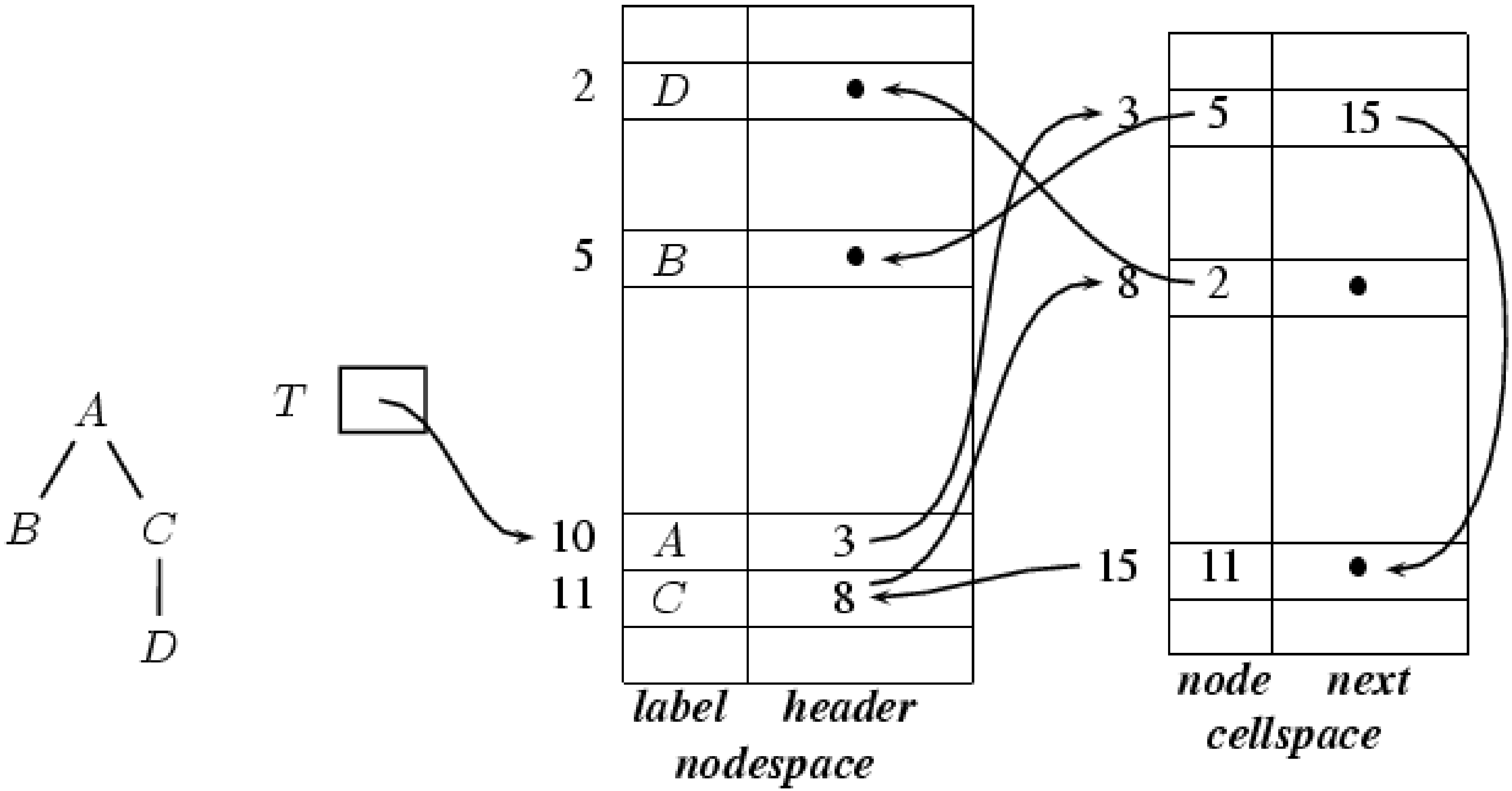


(a) Tree



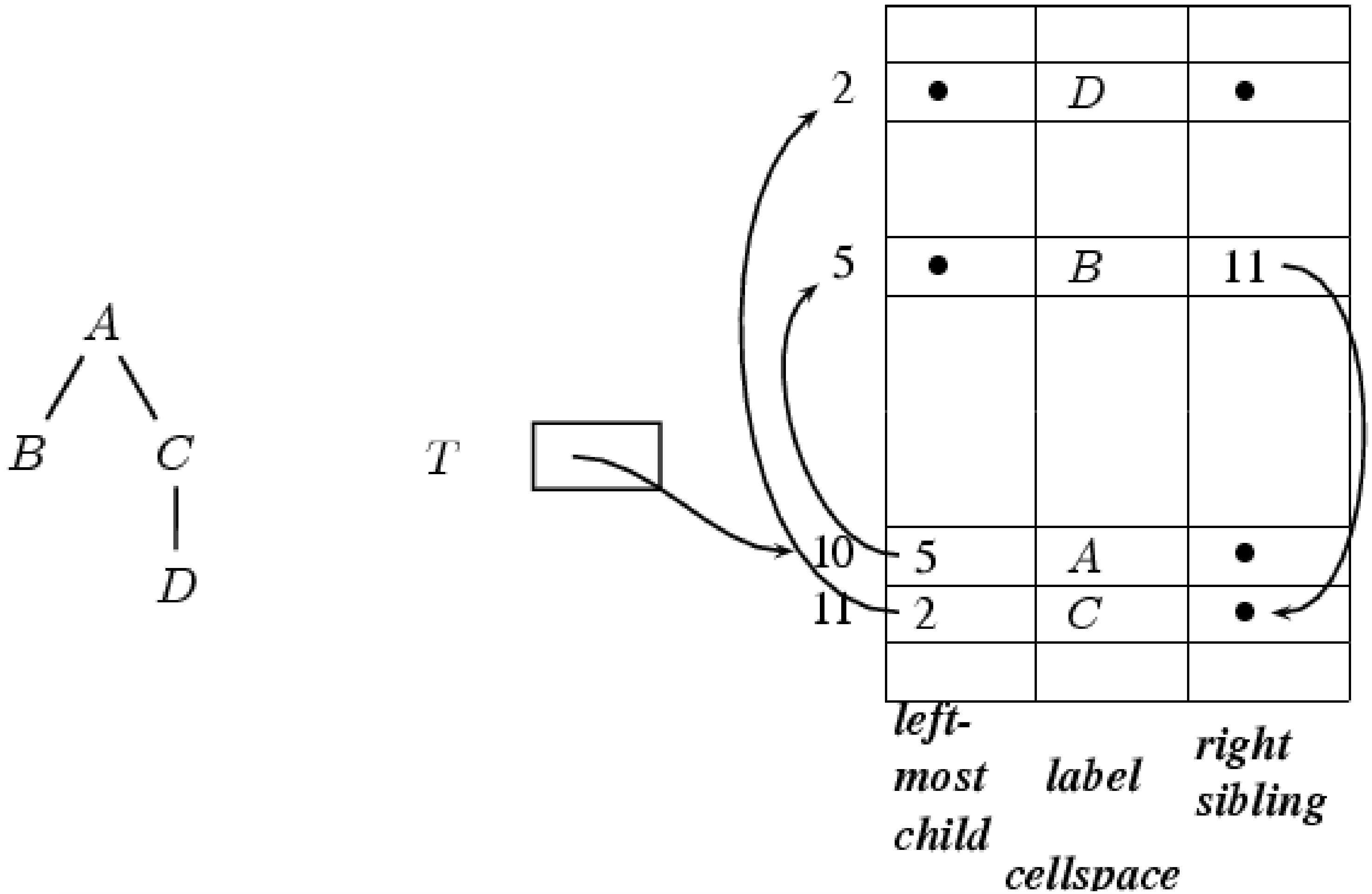
(b) Data structure

Implementations of trees. Lists of children



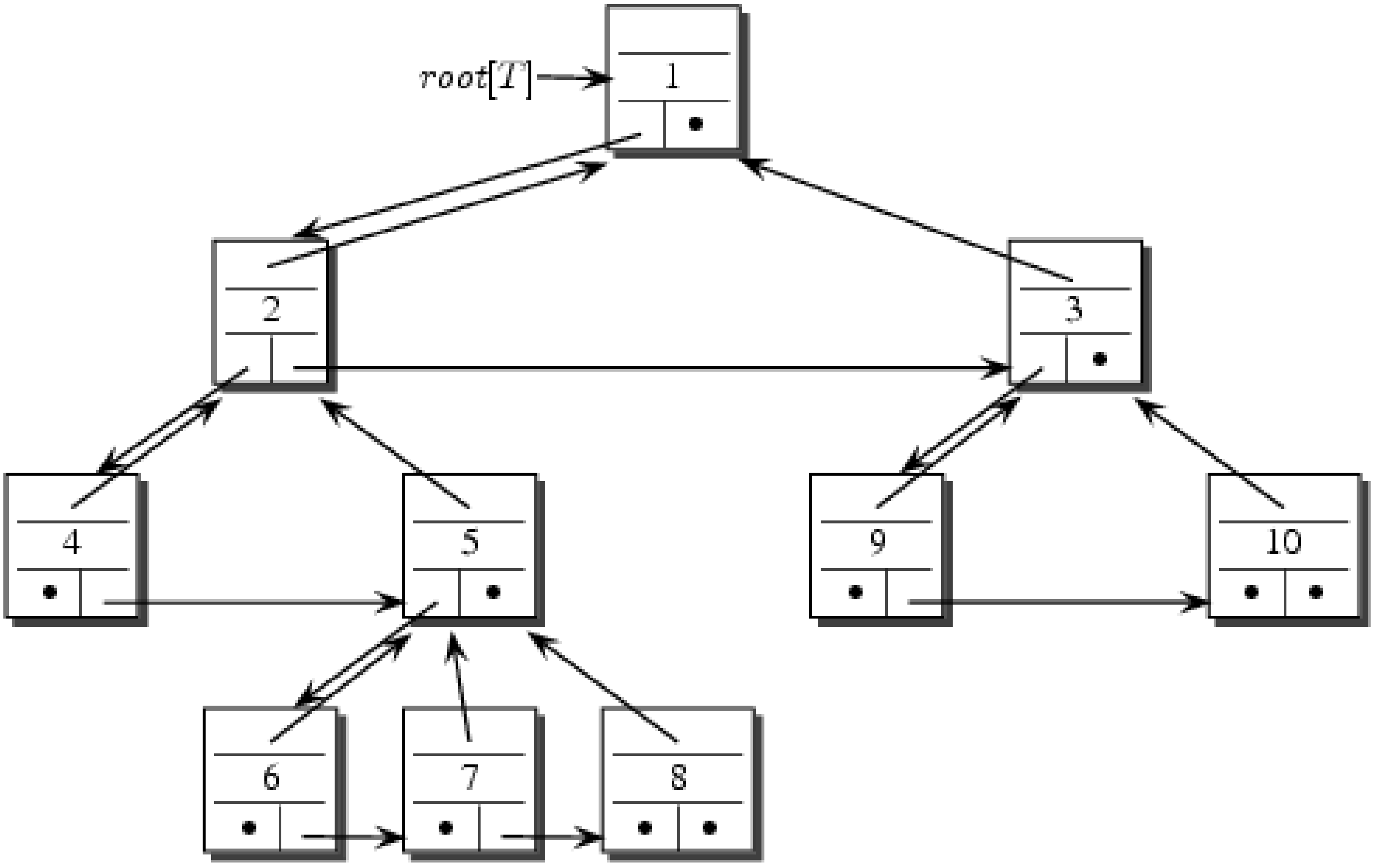
(a) Tree (b) Data structure

Tree leftmost child – right sibling example



(a) Tree (b) Data structure

Another leftmost child – right sibling example

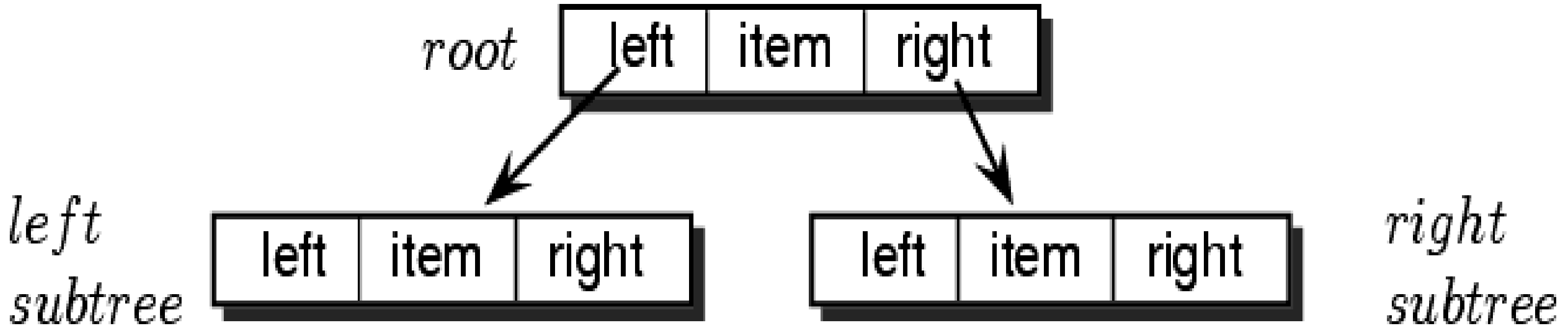


Binary search trees (BST)

- Binary search tree property: for every node, say *x*, in the tree,
  - the values of all keys in the left subtrees are smaller than the key value stored at ***x***,
  - and the values of all the keys in the right subtree are larger than the key value in ***x***
- Typical representation:linked representation of binary tree, with *key, left, right*, [and *p* (*parent*)] fields.

BST implementation

```
typedef struct t_node
{
        void *item;
        struct t_node *left;
        struct t_node *right
} NodeT;
```



BST implementation. Find (recursive)

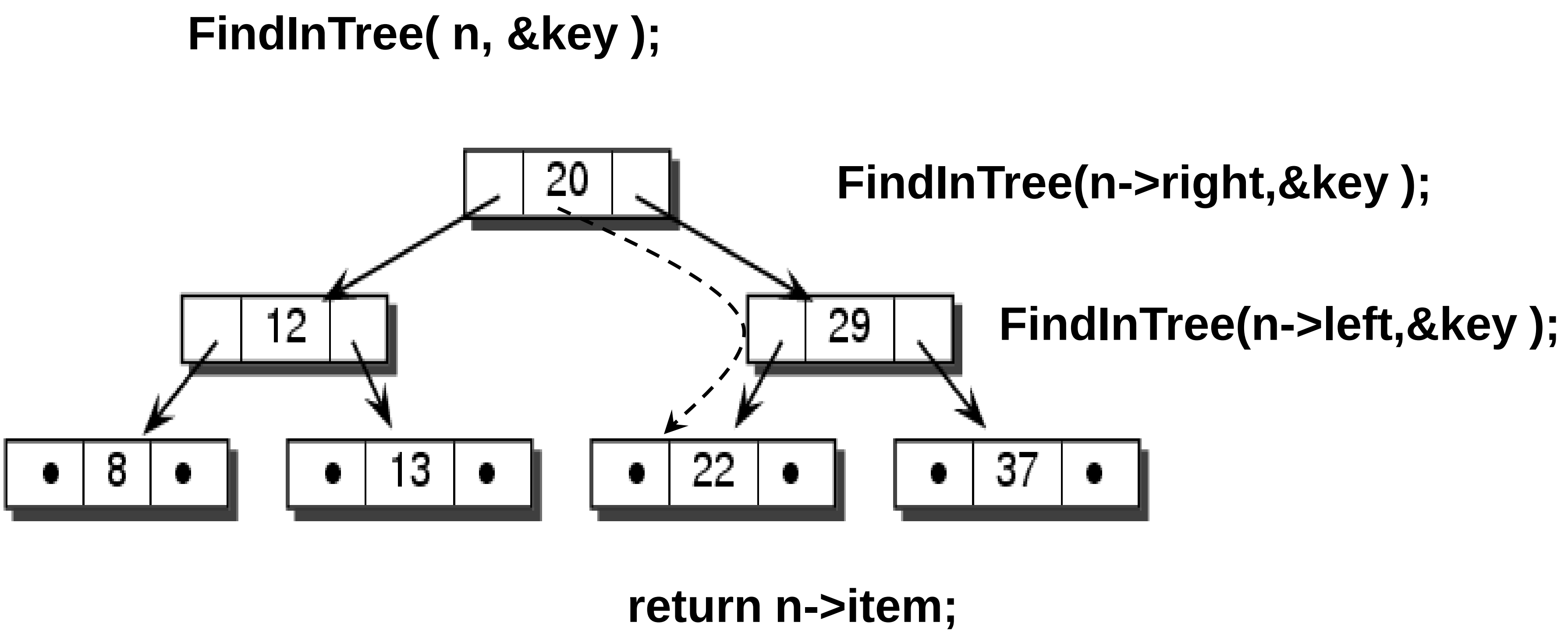
```
extern int KeyCmp( void *a, void *b );
/* Returns -1, 0, 1 for keys stored with a < b, a == b, a > b */
void *FindInTree( NodeT *t, void *key ) {
  if ( t == (Node)0 ) return NULL;
  switch( KeyCmp( key, ItemKey(t->item) ) ) {
    case -1 : return FindInTree( t->left, key );
    case 0:   return t->item;
    case +1 : return FindInTree( t->right, key );
    }
  }
```

Less, search left

Greater, search right

BST implementation. Find (recursive)

- key = 22;
  if (FindInTree(root ,
  &key))…

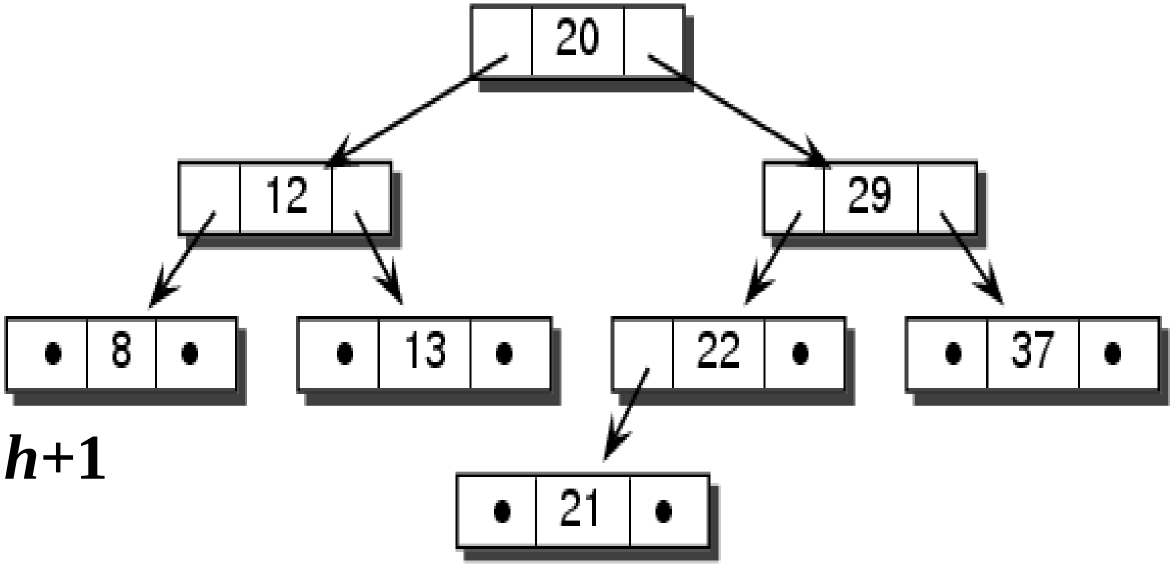


Find performance

- Height, h
  - Nodes traversed in a path from the root to a leaf
- Number of nodes, n
  - $n = 1 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$
  - $h = \lfloor \log_2 n \rfloor$
- Complete Tree
- Since we need at most $h+1$ comparisons, find in $O(h+1)$ or $O(\log n)$
- Same as binary search

BST. Adding a node

- Add 21 to the tree



  - We need at most $h+1$ comparisons
  - Create a new node (constant time)

  ∴ add takes $c_1(h+1)+c_2$ or $c \log n$

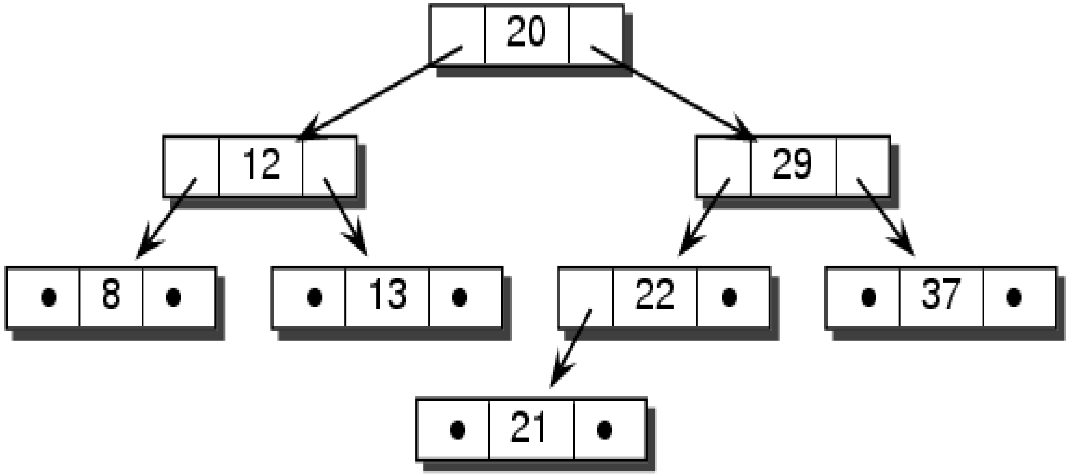
  - So addition to a tree takes time proportional to $\log n$

Addition implementation (recursive)

```
static void AddToTree( NodeT **t, NodeT *new )
{
  NodeT *base = *t;
  /* If it's a null tree, just add it here */
  if ( base == NULL )
       {  *t = new; return; }
  else
    if( KeyLess(ItemKey(new->item),
        ItemKey(base->item)) )
      AddToTree( &(base->left), new );
    else
      AddToTree( &(base->right), new );
  }
```

BST. Deleting a node

- Cases of deletion
  1. A leaf (simple)
  2. A node with a single child
  3. A node with two children
- Example:
  1. Delete 8, 13, 21 or 37
  2. Delete 22
  3. Delete 12, 20 or 29



BST. Deleting a node

```
BSTMinimum(x, k)
    ▷ Input: x: node; k: key to find
    ▷ Output: node or NIL(x, k)
    while left[x] ≠ NIL
        do x ← left[x]
    return x
```

```
BSTSuccessor(x, k)
    ▷ Input: x: node; k: key to find
    ▷ Output: node of minimum key
    if right[x] ≠ NIL
        then return BSTMinimum(right[x])
    y ← p[x] ▷ p[x] is parent of node x
    while y ≠ NIL ∧ x = right[y]
        do x ← y
            y ← p[y]
    return y
```

BST. Deleting a node

```
BSTDelete(T, z)
    ▷ Input: z: node; T: tree
    ▷ Output: nothing
    if z is a leaf                      ▷ (case 0)
        then remove z
    if z has one child                  ▷ (case 1)
        then make p[z] point to the child
    if z has two children               ▷ (case 2)
        then swap z with its successor
             perform case 0 or case 1 to delete it
```

BST Animation

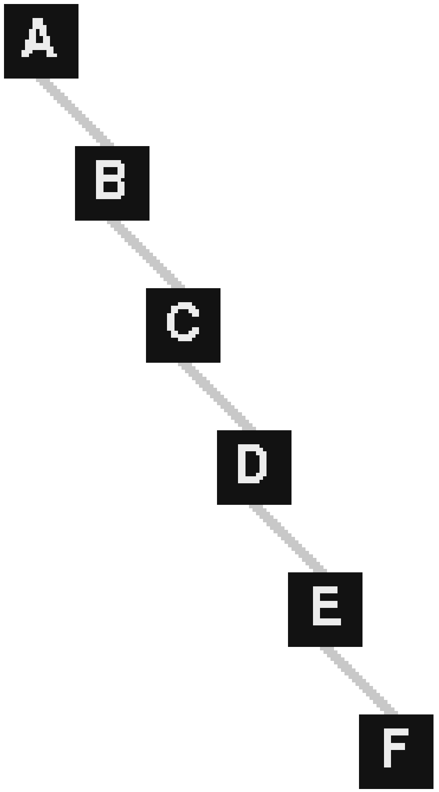
- http://people.ksp.sk/~kuko/bak/

BST performance

- Find $c \log n$
- Add $c \log n$
- Delete $c \log n$
- Apparently efficient in every respect!
- Take this list of characters and form a tree

  A B C D E F

- unbalanced



Performance comparison

	Arrays Simple, fast Inflexible	Linked List Simple Flexible	Trees Still Simple Flexible
Add	O(1) O(n) *inc sort*	O(1) *sort -> no adv*	
Delete	O(n)	O(1) - *any* O(n) - *specific*	
Find	O(n) O(logn) *binary search*	O(n) *(no bin search)*	O(log n)

Reading

- AHU, chapter 3
- CLR, chapters 11.3, 11.4
- CLRS, chapter 10.4, 12
- Preiss, chapter: Trees.
- Knuth, vol. 1, 2.3
- Notes