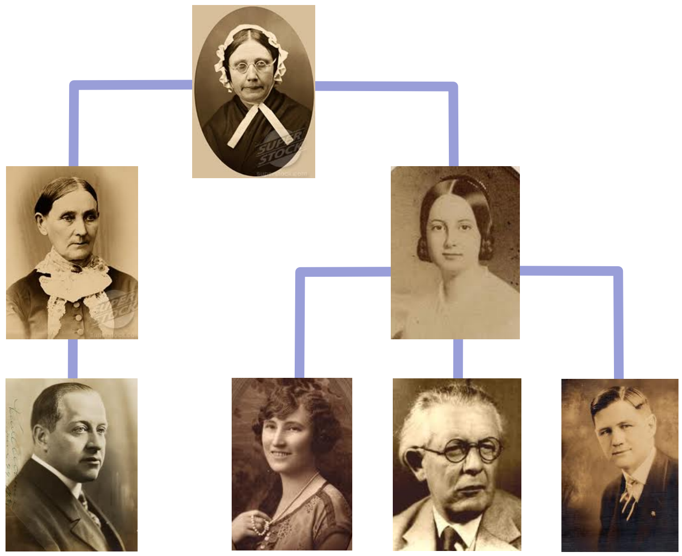
University of Guelph Prof. Pascal Matsakis

CIS2520

5. Trees



Reading suggestion: Chapter 9 of the textbook

DEFINITIONS AND TERMINOLOGY

Implementation

Traversal

Search

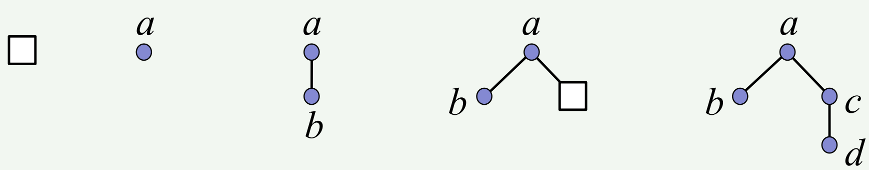
Insertion

Removal

TREES: Recursive Definition 5.3

1. The empty tuple () is a **tree**.
2. Any tuple $(N,T_1,T_2,\ldots,T_n)$ where $n \geq 0$ and $T_1,T_2,\ldots,T_n$ are trees is a **tree**.

$()$ (a) $(a,(b))$ $(a,(b),())$ $(a,(b),(c,(d)))$



TREES: Basic Terminology (1/3) 5.4

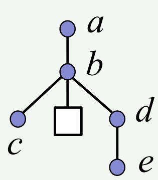
Consider a tree $T=(N,T_1,T_2,\ldots,T_n)$.

✧ N is the **root** of T.

✧ The root N_k of T_k (if any) is the k^{th} **child** of N.

✧ N is the **parent** of N_k.

✧ N_1, N_2, …, N_n are **siblings**.



a is the root
d is the 3^{rd} child of b
b is the parent of c, d
c, d are siblings

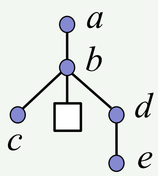
TREES: Basic Terminology (2/3) 5.5

Consider a tree $T=(N,T_1,T_2,\ldots,T_n)$.

✧ **1.** N is an n-**node** (or node of **order** n) of T at **level** 0.
2. An m-node of T_k at level ℓ is an m-**node** of T at **level** $\ell+1$.

✧ **1.** N_k is a **descendant** of N.
2. A descendant of N_k is a **descendant** of N.

✧ **1.** T_k is a **subtree** of T, and it is the k^{th} **subtree** of N.
2. A subtree of T_k is a **subtree** of T.



b is a 3-node at level 1
c, d are nodes at level 2
c, d, e are the descendants of b
e is the only descendant of d
$(d,(e))$ is the 3[rd] subtree of b

TREES: Basic Terminology (3/3) 5.6

Consider a tree $T=(N,T_1,T_2,\ldots,T_n)$.

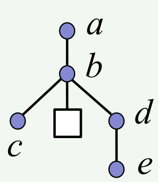
✧ A node without children is a **leaf**, or an **external node**.

✧ A node with children is a **parent**, or an **internal node**.

✧ If N″ is a descendant of N′ then N′ is an **ancestor** of N″.

✧ The maximum order of a node is the **order** of T.

✧ The maximum level of a node is the **height** of T.

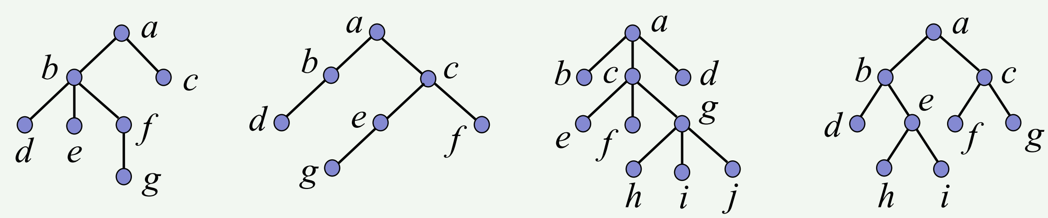


the leaves are c, e
the internal nodes are a, b, d
the ancestors of d are a, b
the order of the tree is 3
the height of the tree is 3

A tree is an ***m-ary tree*** if every node is an m-node.
A 2-ary tree is a ***binary tree***, a 3-ary tree is a ***ternary tree***, etc.

An m-ary tree is ***full*** if every node has exactly 0 or m children.
An m-ary tree is ***balanced*** if, for each node, the node's subtrees have heights that differ by at most 1. (*Note:* The height of () is −1.)

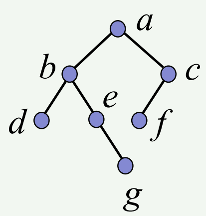
ternary tree not full not balanced	binary tree not full balanced	ternary tree full not balanced	binary tree full balanced



(the empty subtrees are not represented)

Let T be a binary tree.
For any node N of T:

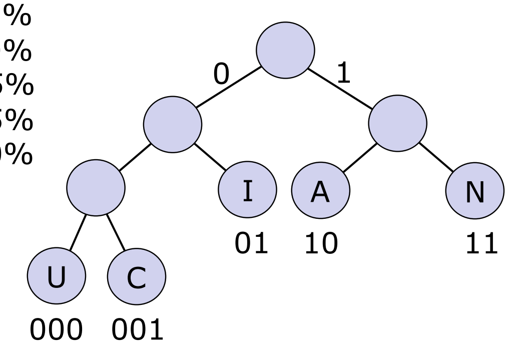
✧ The 1st child of N (if any) is the **left child** of N.
✧ The 2nd child of N (if any) is the **right child** of N.
✧ The 1st subtree of N is the **left subtree** of N.
✧ The 2nd subtree of N is the **right subtree** of N.



e is the right child of b
f is the left child of c
$(g,(),())$ is the right subtree of e
$(b,(d,(),()),(e,(),(g,(),())))$ is the left subtree of a

TREES > m-ary > Binary > Huffman (example) 5.9

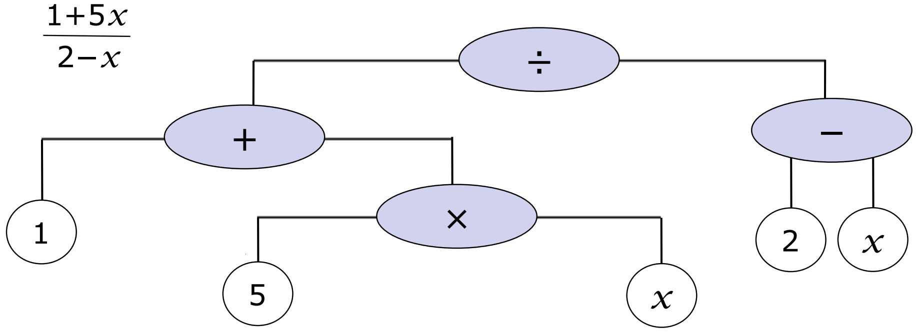
A: 30%
C: 10%
I: 25%
N: 25%
U: 10%



Huffman tree:
gives optimal prefix-free binary code.

TREES > m-ary > Binary > Expression (example) 5.10

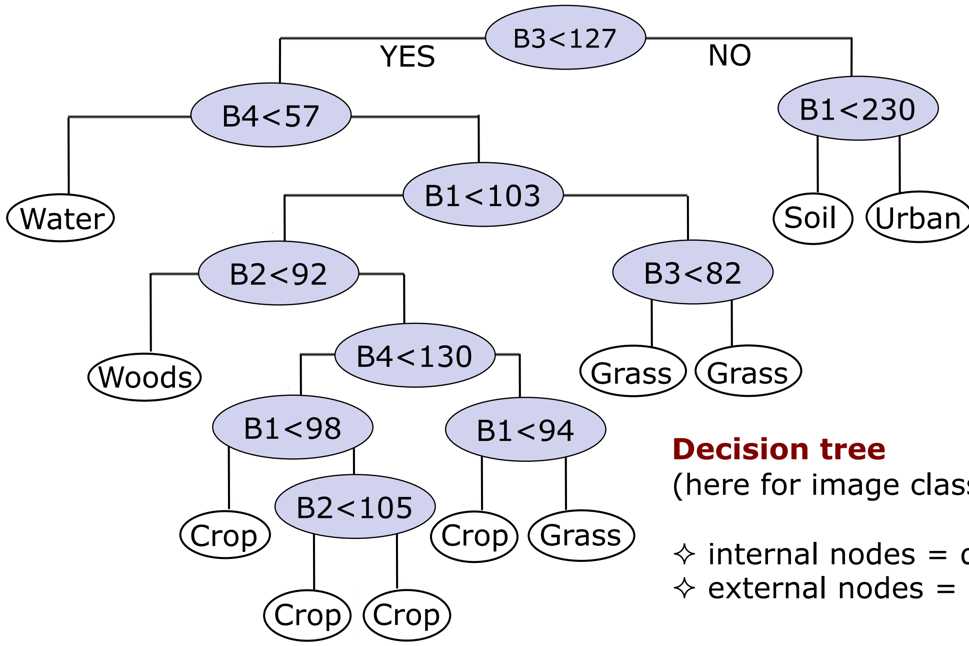
$$\frac{1+5x}{2-x}$$



Expression tree:

✧ internal nodes = operators
✧ external nodes = operands

TREES > m-ary > Binary > Decision (example) 5.11



Decision tree
(here for image classification):

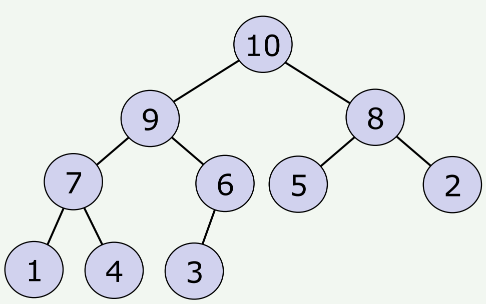
✧ internal nodes = questions
✧ external nodes = decisions

TREES > m-ary > Binary > Heap 5.12

Let T be a binary tree of height h. Assume:

✧ Any node with less than 2 children is at level h or h−1.
✧ The leaves at level h are placed as far left as possible.
✧ Each node is in the form (item,**key**).
✧ All keys from all nodes are comparable under some order relation $\preccurlyeq$
✧ The key of each child is less than or equal to the key of its parent.

Then (T,$\preccurlyeq$) is a **heap**.



(the items are not indicated)

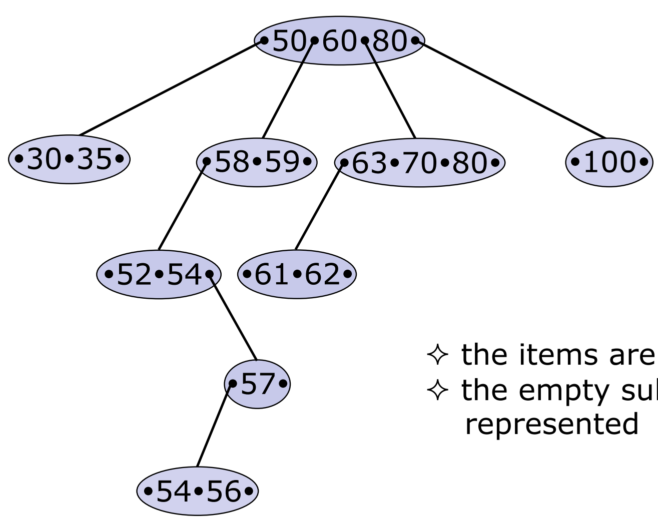
TREES > m-Way Search 5.13

Let T be a tree. Assume:

- ✧ Each node is in the form $((\text{item}_1,\mathbf{key_1}),(\text{item}_2,\mathbf{key_2}),\ldots,(\text{item}_n,\mathbf{key_n}))$ where $n \geq 1$ depends on the node.
- ✧ All keys from all nodes are comparable under some order relation $\preccurlyeq$
- ✧ The keys in each node are in ascending order.
- ✧ A node with n keys has n+1 subtrees.
- ✧ The keys in the first i subtrees are less than or equal to the i^{th} key.
- ✧ The keys in the other subtrees are greater than or equal to the i^{th} key.

Then $(T,\preccurlyeq)$ is a **multiway search tree**.
A multiway search tree of order m is an **m-way search tree**.

TREES > m-Way Search (example 1) 5.14



- ✧ the items are not indicated
- ✧ the empty subtrees are not represented

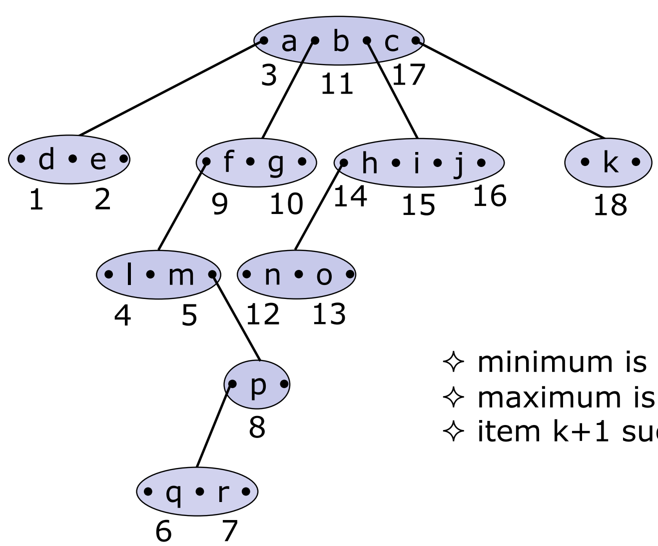
TREES > m-Way Search (some terms) 5.15

Let (T,≼) be a multiway search tree.
Assume all keys are distinct:

- ✧ The **minimum** item is the item with minimum key.
- ✧ The **maximum** item is the item with maximum key.
- ✧ The **successor** of an item is the item with the smallest larger key.
- ✧ The **predecessor** of an item is the item with the largest smaller key.

These definitions can easily be extended
to the case when not all keys are distinct.

TREES > m-Way Search (example 2) 5.16

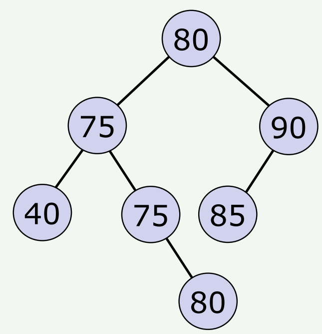


- ✧ minimum is item 1
- ✧ maximum is item 18
- ✧ item k+1 successor of k

TREES > m-Way Search > Binary Search 5.17

Let $(T, \preccurlyeq)$ be a multiway search tree.

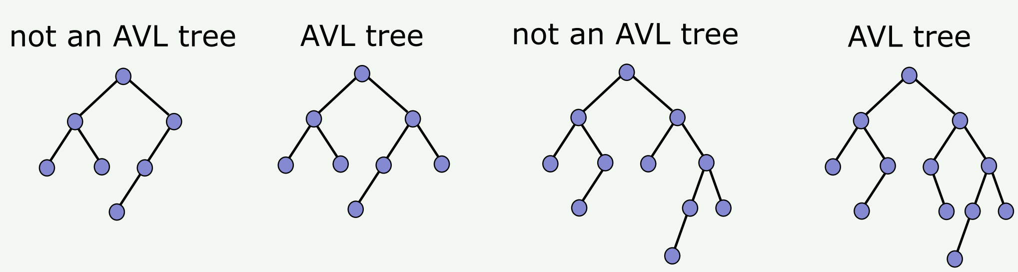
If T is a binary tree
then $(T, \preccurlyeq)$ is a **binary search tree**.



TREES > m-Way Search > Binary Search > AVL 5.18

Let $(T, \preccurlyeq)$ be a binary search tree.

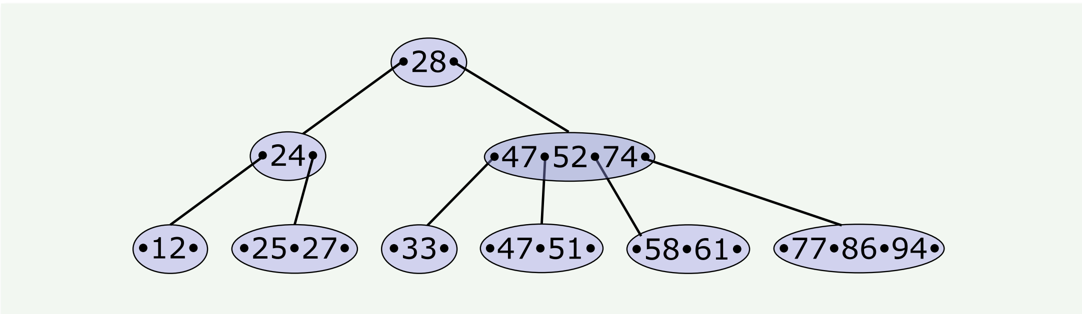
If T is balanced
then $(T, \preccurlyeq)$ is an **AVL tree**.



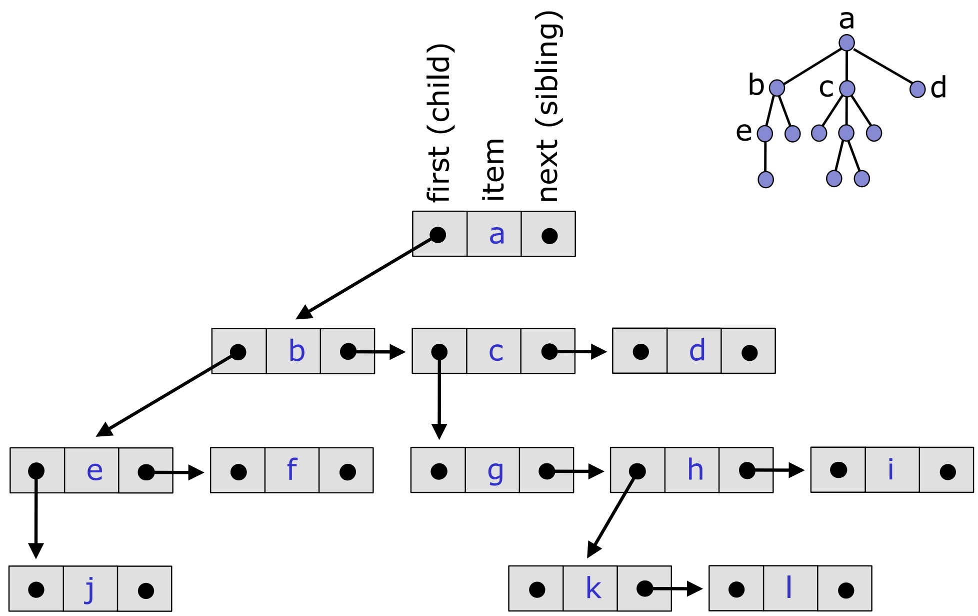
Let $(T, \preccurlyeq)$ be a multiway search tree. Assume:

✧ Each node is with 1, 2 or 3 keys.
✧ A parent with n keys has n+1 children.
✧ All the leaves are at the same level.

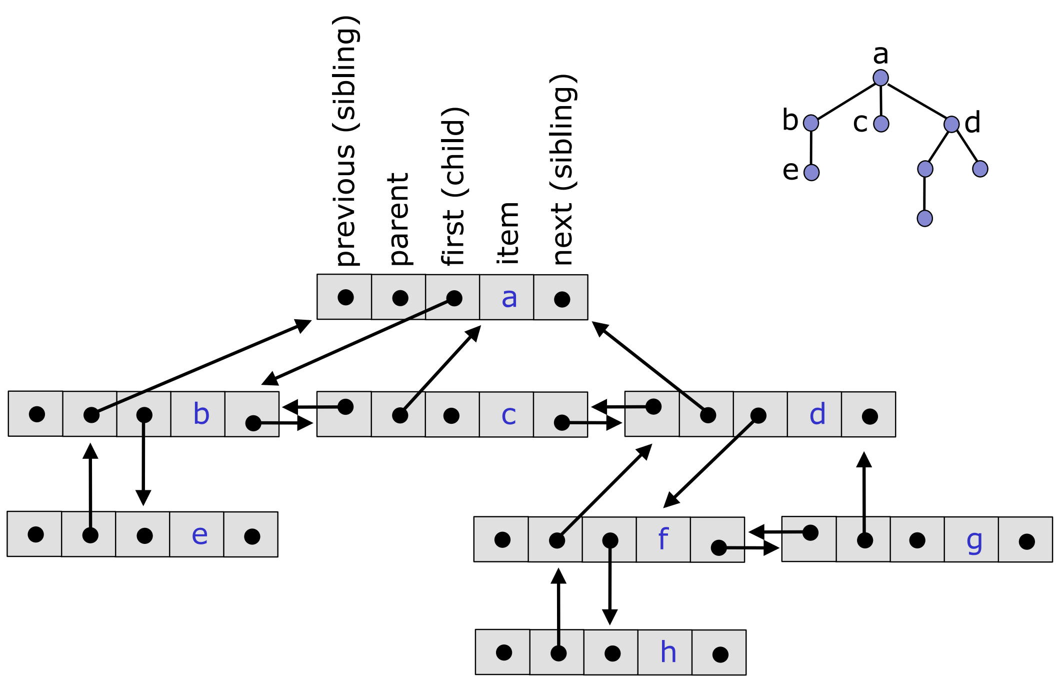
Then $(T, \preccurlyeq)$ is a **2-4 tree**.



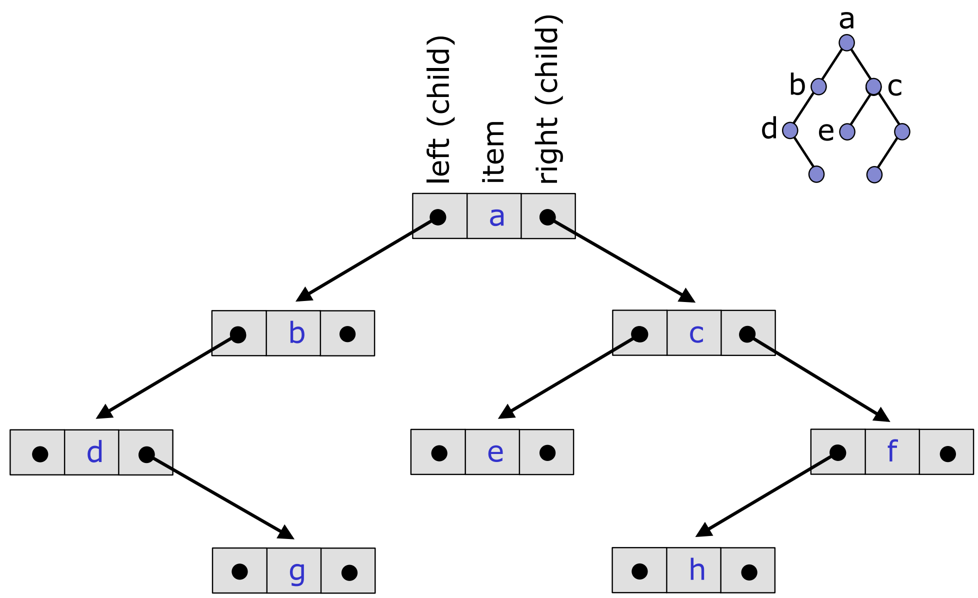
TREES (1/2) 5.21



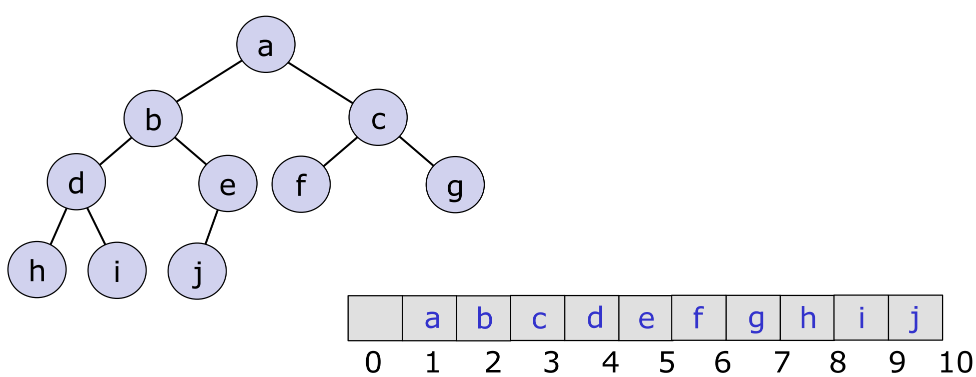
TREES (2/2) 5.22



BINARY TREES 5.23



HEAPS 5.24



If node at rank i then left child at rank 2i
and right child at rank 2i+1

Definitions and Terminology
Implementation
TRAVERSAL
Search
Insertion
Removal

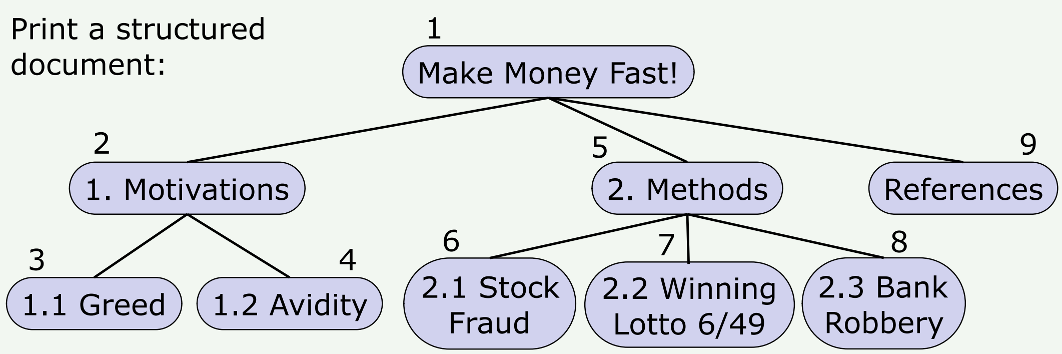
Trees > PREORDER 5.26

Goal: visit every node once and only once.

```
function Preorder (N)
   visit N
   for each child C of N from first to last
      Preorder(C)
```

O(n) with n number of nodes

Print a structured document:



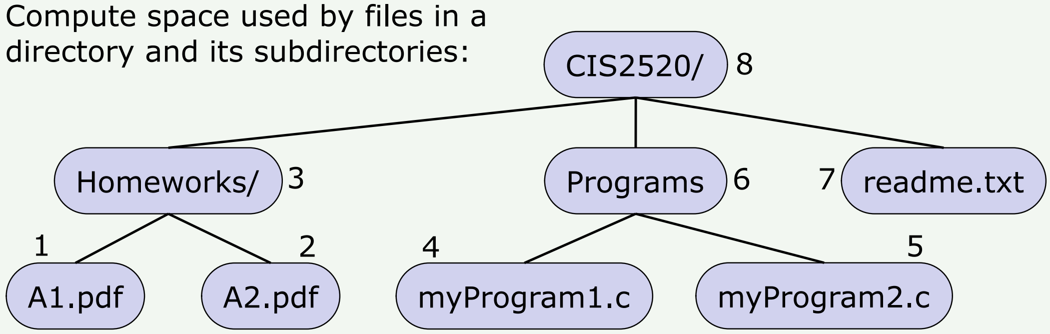
Trees > POSTORDER 5.27

Goal: visit every node once and only once.

```
function Postorder (N)
   for each child C of N from first to last
      Postorder(C)
   visit N
```

O(n) with n number of nodes

Compute space used by files in a directory and its subdirectories:



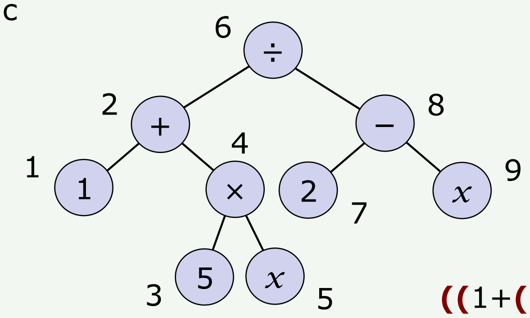
Trees > Binary > INORDER 5.28

Goal: visit every node once and only once.

```
function Inorder (N)
   if N has a left child L then Inorder(L)
   visit N
   if N has a right child R then Inorder(R)
```

O(n) with n number of nodes

Print an arithmetic expression:



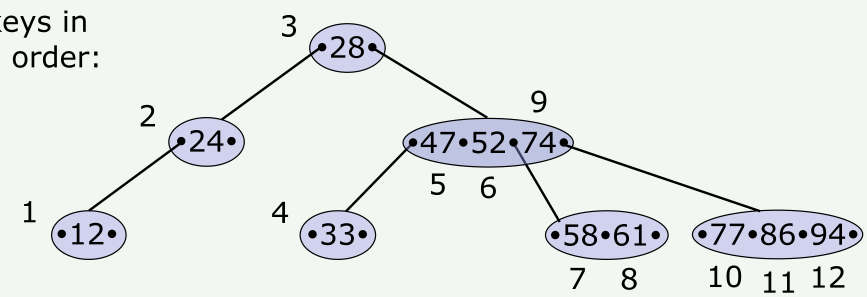
$((1+(5\times x))\div(2-x))$

Goal: visit every (item,key) pair once and only once.

```
function Inorder (N)
   for i=1 to the number k of keys of N
      if N has an ith child Ci then Inorder(Ci)
      visit (itemi,keyi)
   if N has a (k+1)th child Ck+1 then Inorder(Ck+1)
```

O(n) with n number of keys

Visit the keys in ascending order:

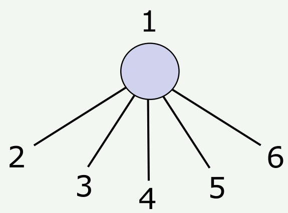


TREES 5.31

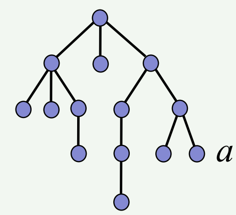
Goal: determine whether a given item is stored in the tree.

```
function Search (N, I)
   if N.item=I then return true
   for each child C of N from first to last
       if Search(C,I)=true then return true
   return false
```

O(n) with n number of nodes



Worst case:
item is stored in node a

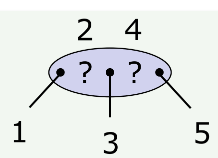


Trees > MULTIWAY SEARCH 5.32

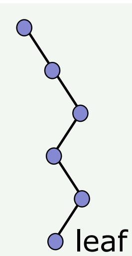
Goal: find in the tree an item with a given key.

```
function Search (N, K)
   let key_i be the i^th key of N
   if K<key_1 and N has a 1^st child C_1
   then return Search(C_1,K)
   for i=2 to the number k of keys of N
       if K=key_i-1 then return item_i-1
       if key_i-1<K<key_i and N has an i^th child C_i
       then return Search(C_i,K)
   if K=key_k then return item_k
   if K>key_k and N has a (k+1)^th child C_k+1
   then return Search(C_k+1,K)
   return nil
```

O(hm) with h=height m=order



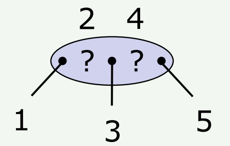
Worst case: h=n−1 and m=2, with n number of keys, and item in leaf



Goal: find in the tree an item with a given key.

```
function Search (N, K)
   let key_i be the i^th key of N
   if K<key_1 and N has a 1^st child C_1
   then return Search(C_1,K)
   for i=2 to the number k of keys of N
       if K=key_{i-1} then return item_{i-1}
       if key_{i-1}<K<key_i and N has an i^th child C_i
       then return Search(C_i,K)
   if K=key_k then return item_k
   if K>key_k and N has a (k+1)^th child C_{k+1}
   then return Search(C_{k+1},K)
   return nil
```

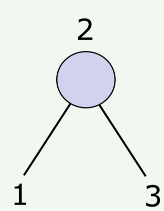
O(log n) with n number of keys



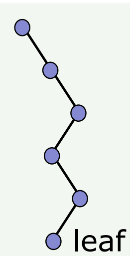
Goal: find in the tree an item with a given key.

```
function Search (N, K)
   if K<N.key and N.left≠nil
      return Search(N.left,K)
   if K=N.key
      return N.item
   if K>N.key and N.right≠nil
      return Search(N.right,K)
   return nil
```

O(h) with h=height



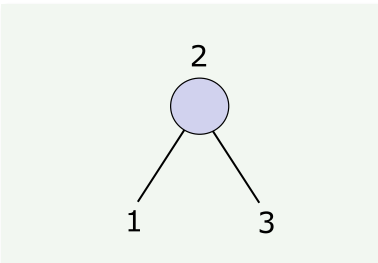
Worst case: h=n−1,
with n number of nodes,
and item stored in leaf



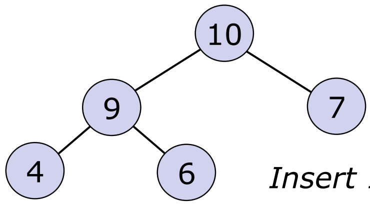
Goal: find in the tree an item with a given key.

```
function Search (N, K)
   if K<N.key and N.left≠nil
      return Search(N.left,K)
   if K=N.key
      return N.item
   if K>N.key and N.right≠nil
      return Search(N.right,K)
   return nil
```

O(log n)
with n number of nodes

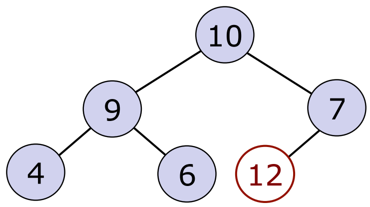


HEAPS

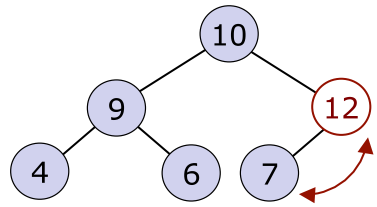


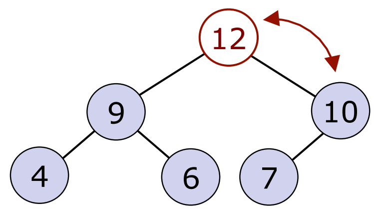
Insert 12?

1. Insert to the “end” of the heap



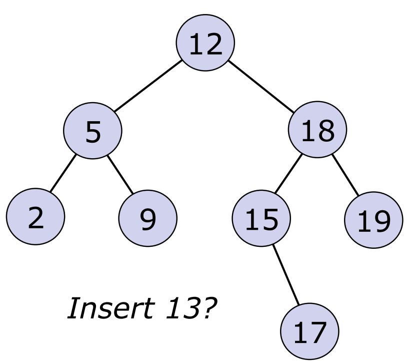
2. Restore the heap-order property, i.e., (up)heapify



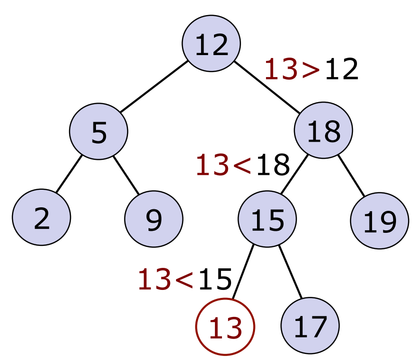


O(log n) with n number of nodes

BINARY SEARCH TREES



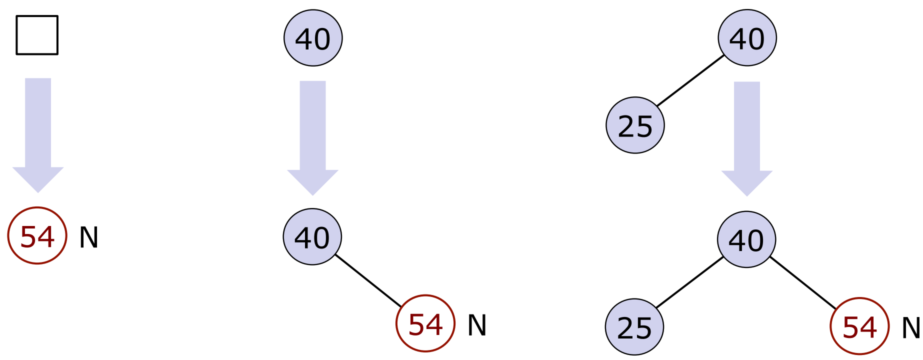
Insert 13?



O(h) with h height

AVL TREES (1/4) 5.39

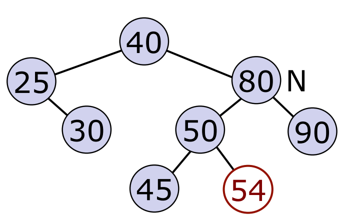
```
insert (item,key) as in a binary search tree
N=newly created node
if N.parent=nil or N.parent.parent=nil then return
......
```



return **return** **return**

AVL TREES (2/4) 5.40

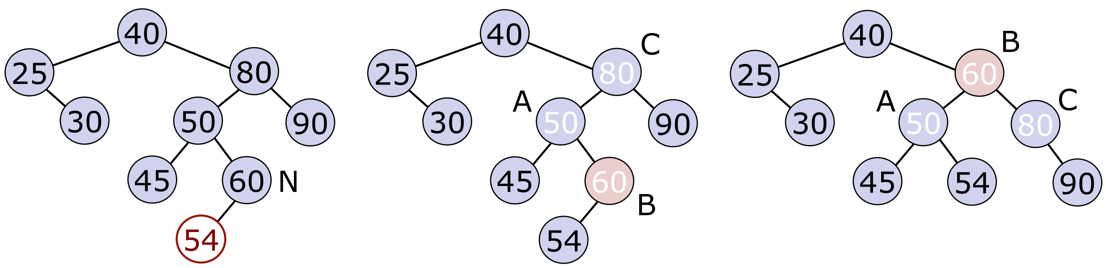
```
insert (item,key) as in a binary search tree
N=newly created node
if N.parent=nil or N.parent.parent=nil then return

while N.parent.parent is balanced
      N=N.parent
      if N.parent.parent=nil then return

......
```



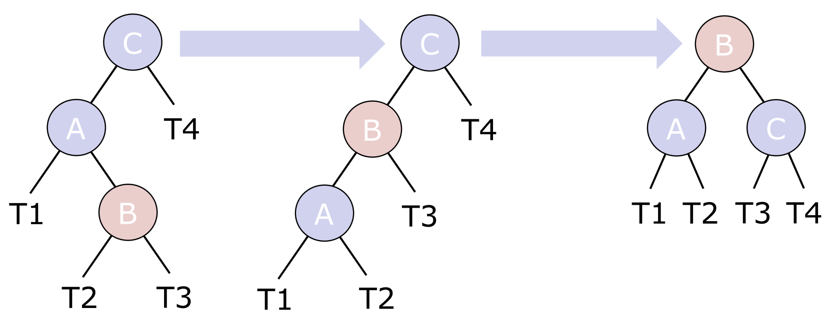
return

AVL TREES (3/4) 5.41

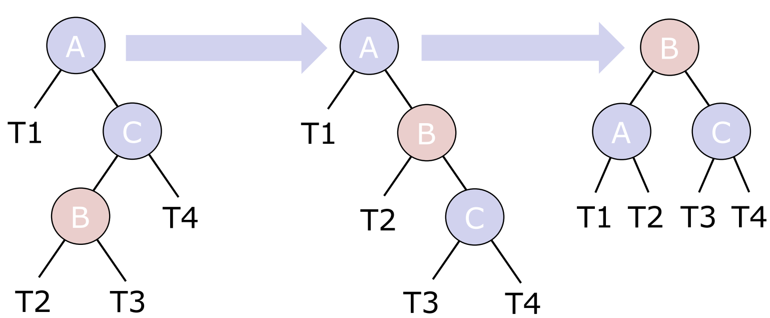
```
insert (item,key) as in a binary search tree
N=newly created node
if N.parent=nil or N.parent.parent=nil then return

while N.parent.parent is balanced
      N=N.parent
      if N.parent.parent=nil then return

A=1st of N, N.parent and N.parent.parent in inorder traversal
B=2nd of N, N.parent and N.parent.parent in inorder traversal
C=3rd of N, N.parent and N.parent.parent in inorder traversal
restructure the tree to make B the parent of A and C *
```



AVL TREES (4/4) 5.42



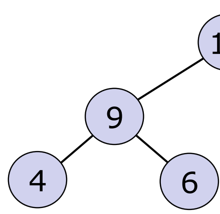
* restructure...



O(log n) with n number of nodes

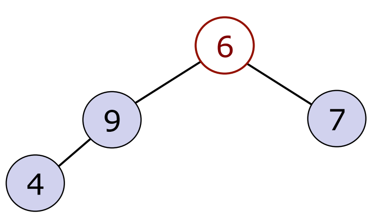
Definitions and Terminology
Implementation
Traversal
Search
Insertion
REMOVAL

HEAPS 5.44

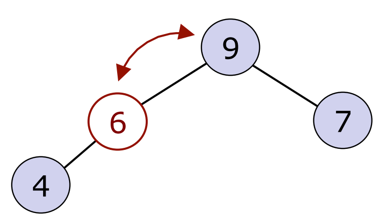


Remove?

1. Replace the root with the "last" node and delete the last node



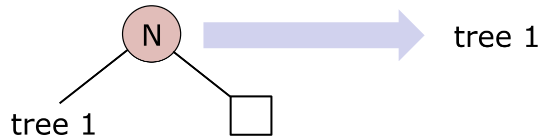
2. Restore the heap-order property, i.e., (down)heapify



O(log n) with n number of nodes

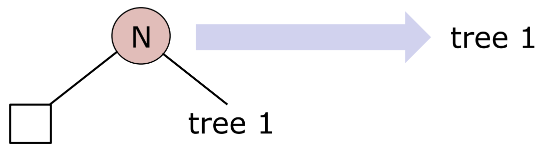
BINARY SEARCH TREES (1/2) 5.45

case 1:



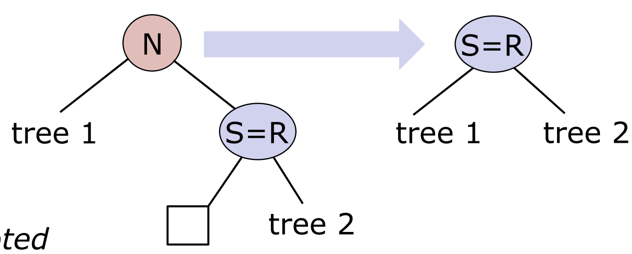
consider the subtree rooted
at the node N to be removed

case 2:



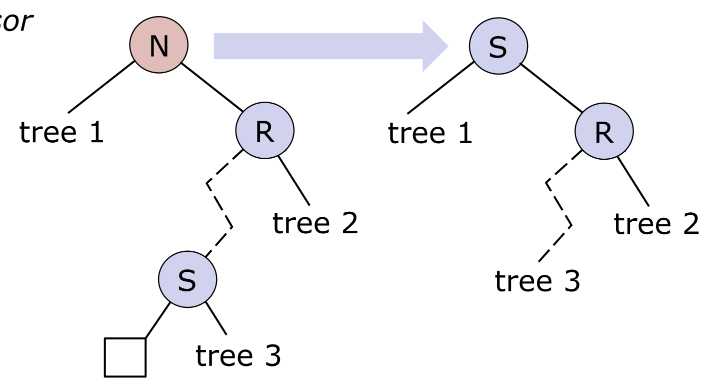
BINARY SEARCH TREES (2/2) 5.46

case 3:



consider the subtree rooted
at the node N to be removed
and let S be N's successor

case 4:



O(h) with h height