COS 423 Lecture 3
Binary Search Trees

Dictionary: contains a set S of items, each with associated information.

Operations:

Access(x): Determine if x is in S. If so, return x's information.

Insert(x):(x not in S) Insert x and its information.

Delete(x):(x in S) Delete x and its information.

Binary Search

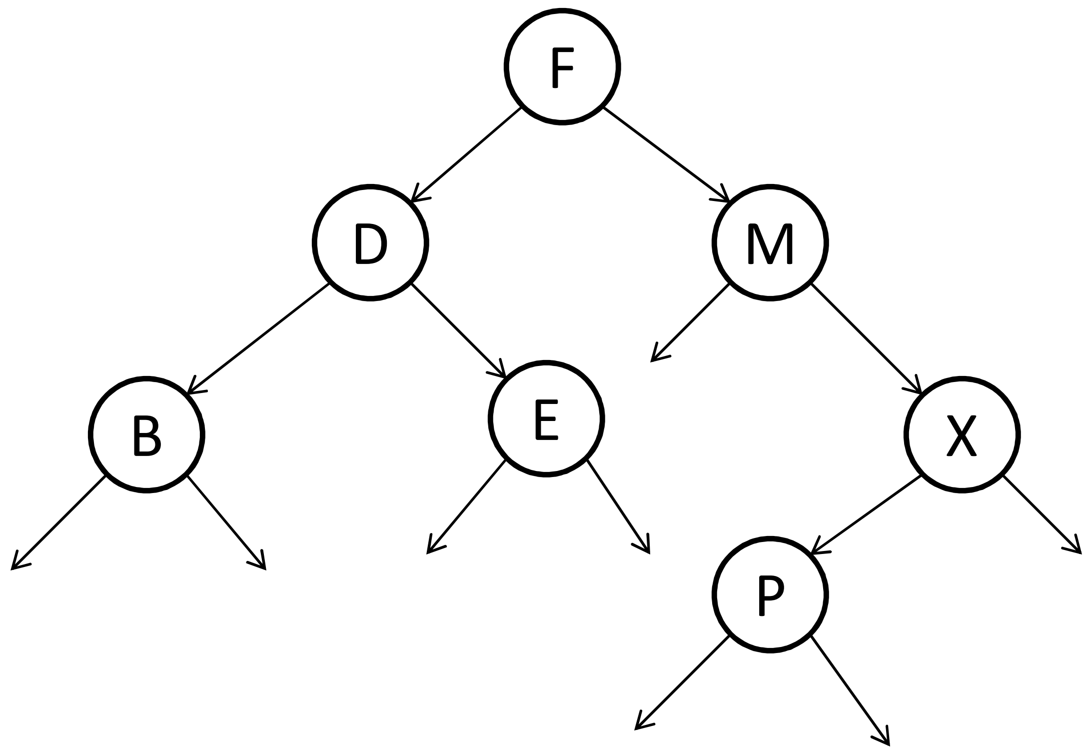
Universe of items (or of access keys or index values) is totally ordered, allowing binary comparison

Binary search: Maintain S in sorted order.

To find x in S:

- If S empty, stop (failure).
- If S non-empty, compare x to some item y in S.
  - If $x = y$, stop (success).
  - If $x < y$, search in $\{z \text{ in } S | z < y\}$.
  - If $x > y$, search in $\{z \text{ in } S | z > y\}$.

Implementation: Binary Search Tree



Binary tree: Each node x has a left child *left*(x) and a right child *right*(x), either or both of which can be *null*. Node x is the *parent* of both of its children: $p(left(x)) = p(right(x)) = x$.

A node is *binary*, *unary*, or a *leaf* if it has 0, 1, or 2 null children, respectively.

n = #(non-null) nodes

Binary Tree Parameters

Depth (path length from top):

$d(\mathit{root}) = 0$

$d(\mathit{left}(x)) = d(\mathit{right}(x)) = d(x) + 1$

Height (path length to bottom):

$h(\text{null}) = -1$

$h(x) = 1 + \max\{h(\mathit{left}(x)), h(\mathit{right}(x))\}$

Size (number of nodes in subtree):

$s(\text{null}) = 0$

$s(x) = 1 + s(\mathit{left}(x)) + s(\mathit{right}(x))$

Binary tree representation

At a minimum, each node contains pointers to its left and right children.

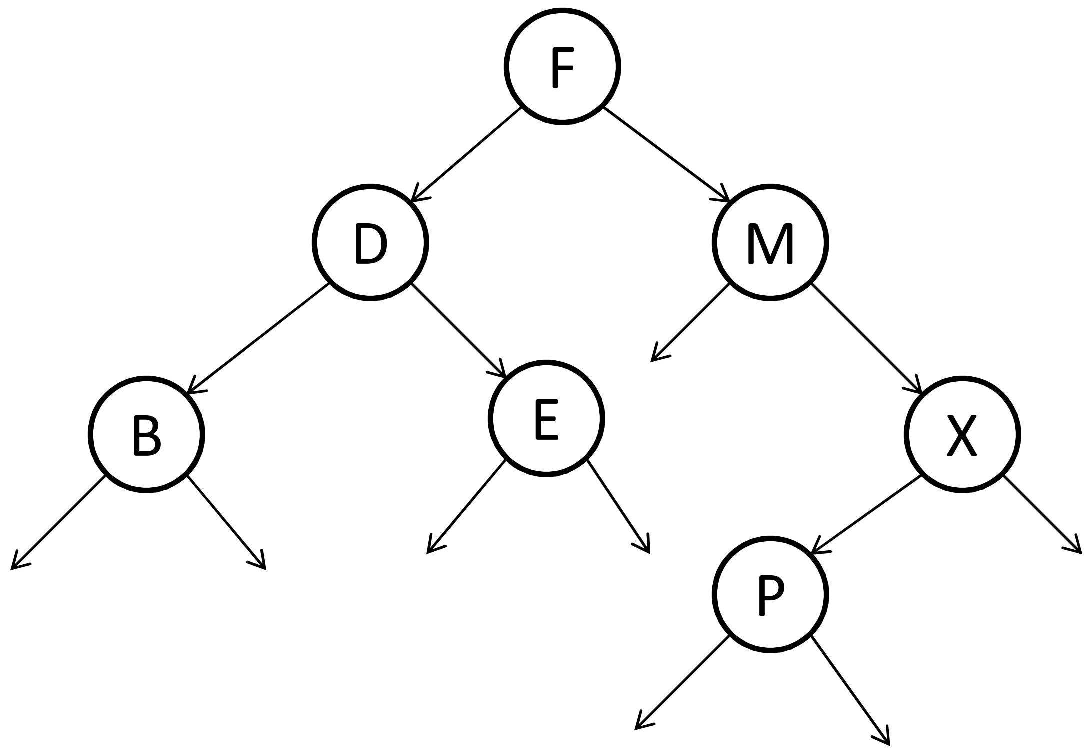
Depending on the application, each node x holds additional information, e.g. an item and its associated data; a pointer to $p(x)$; $size(x)$.

Binary Search Tree: Internal representation

Items (key plus data) in nodes, one per node, in *symmetric order* (*in-order*): items in left subtree are less, items in right subtree are greater.

To find an item takes O(d + 1) time, where d = depth of item's node, or of null node reached by search if item is not in tree.

Binary Search Tree
Internal representation



Binary Search Tree: External representation

Actual items (keys plus data) are in external (previously *null*) nodes.

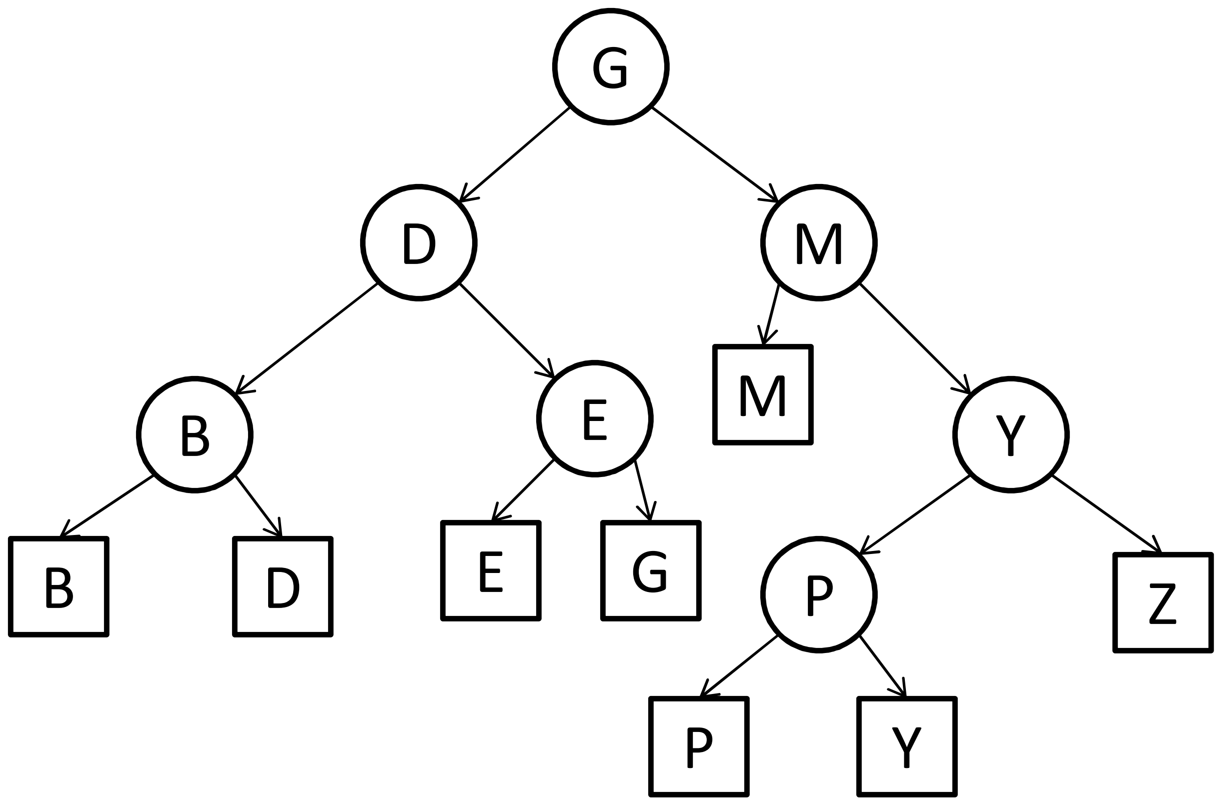
Internal (previously *non-null*) nodes hold dummy items (keys only) to support search: on equality, branch left.

All items are in symmetric order.

All searches stop at an external node.

Twice as many nodes, but some simplifications, notably in deletion

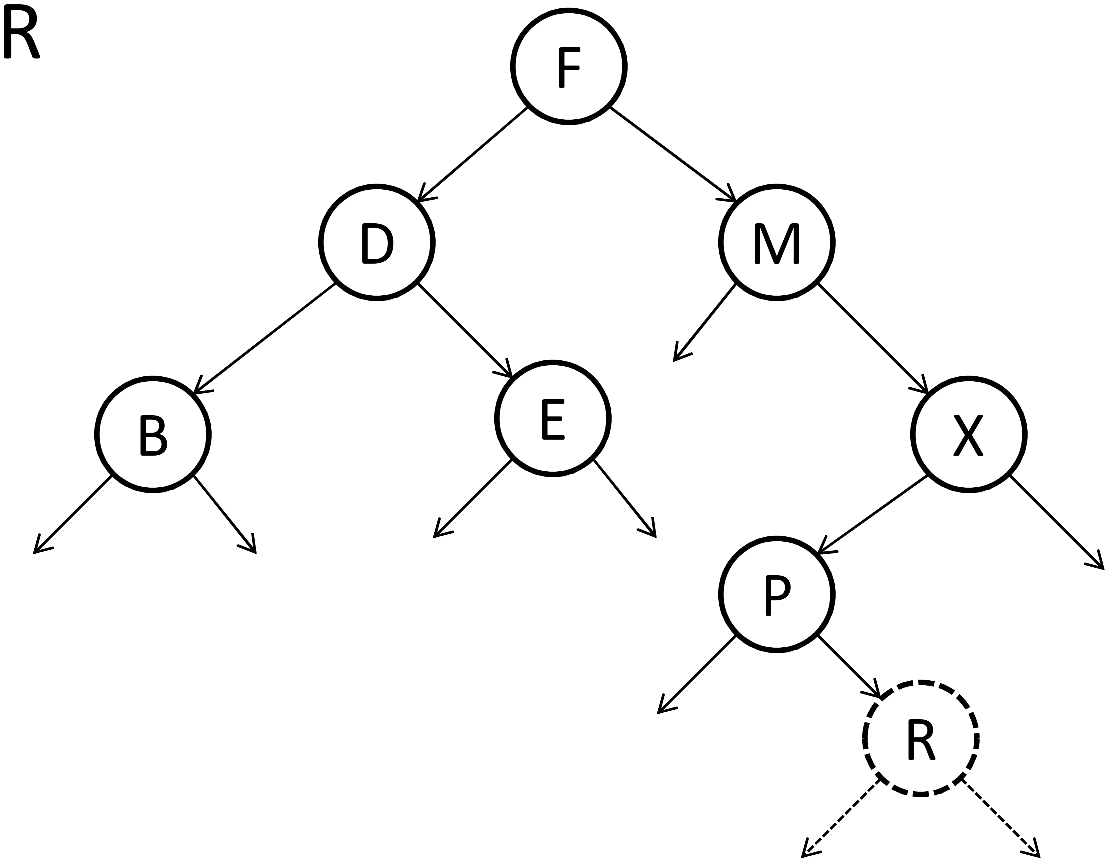
Binary Search Tree
External representation



Insertion (internal)

Search. Replace null by node with item.

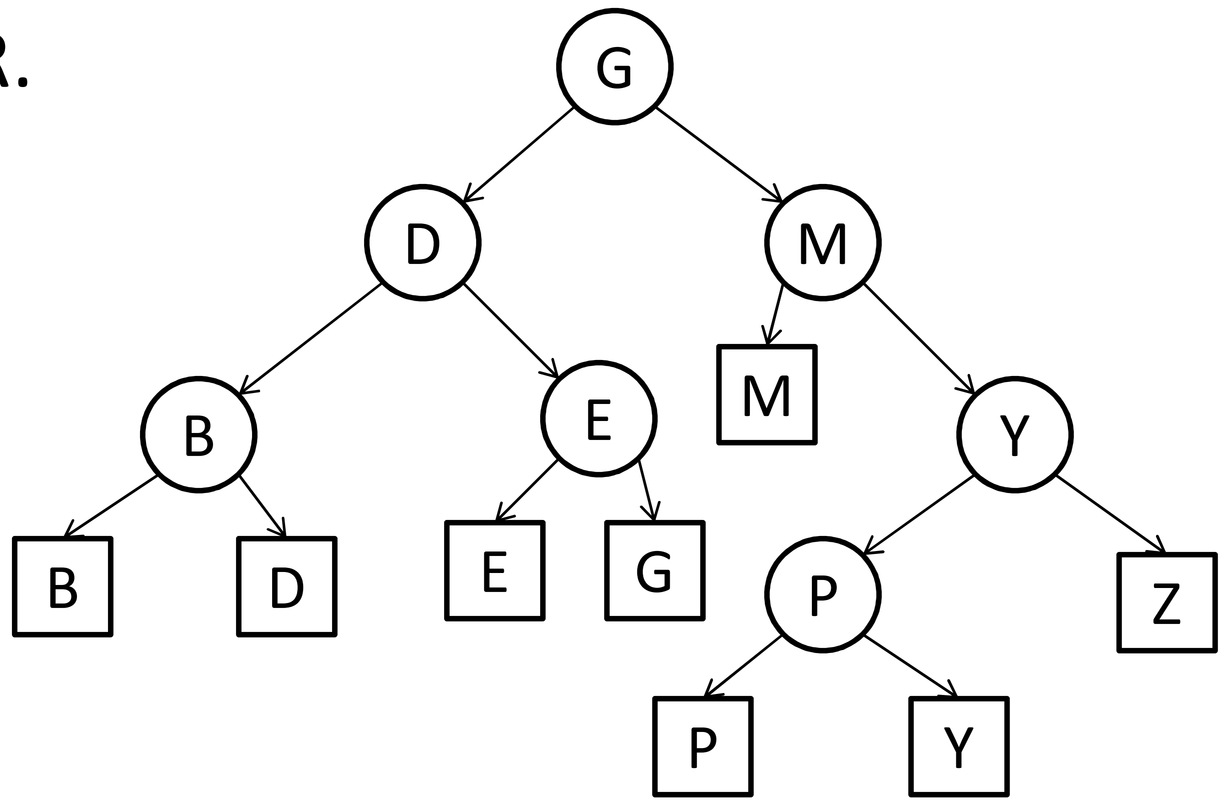
Insert R



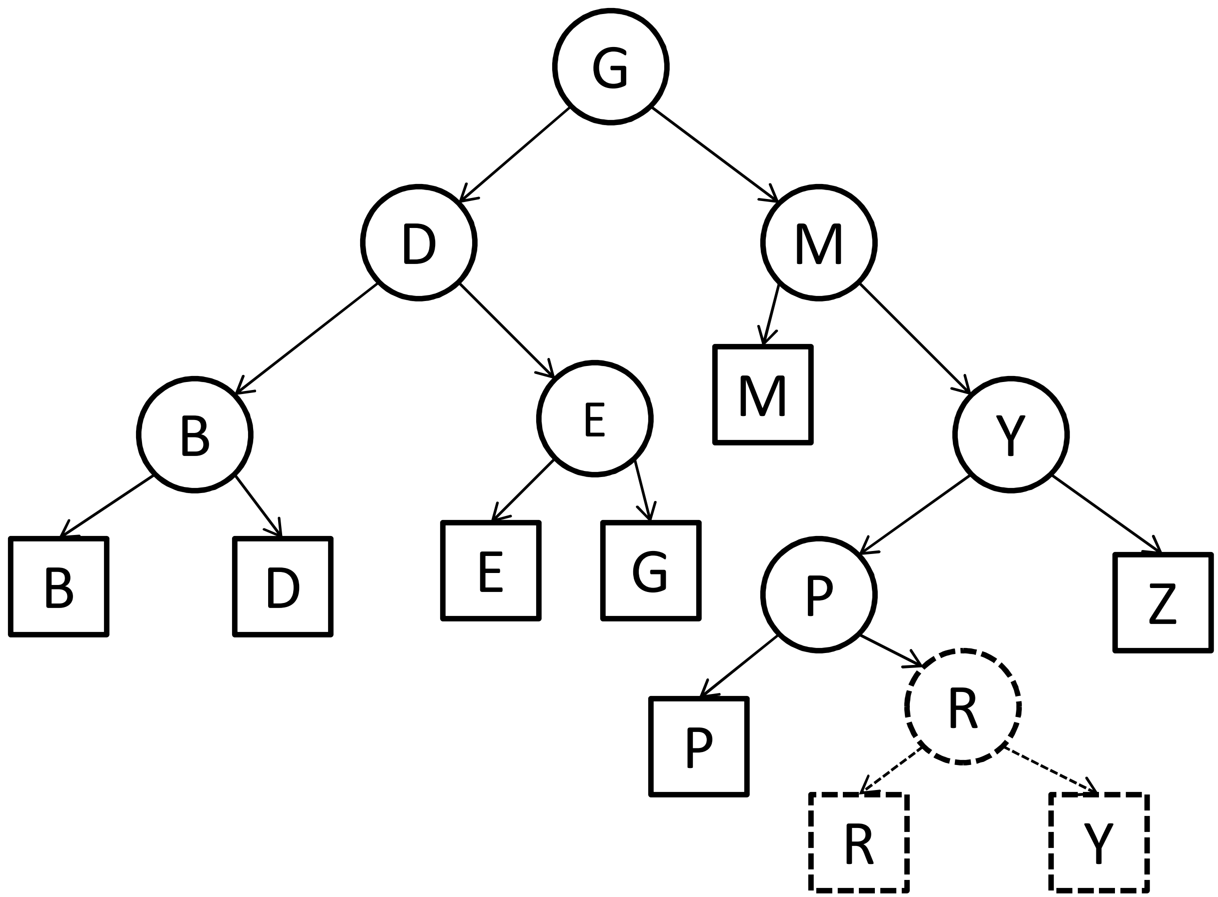
Insertion (external)

Search. Replace external node by internal node with two external children.

Insert R.



Insert R



Deletion

Lazy: Find item. Remove its data but leave its node (with key) so search is still possible.

Eager: Find item. Remove node. Repair tree.

Internal representation:

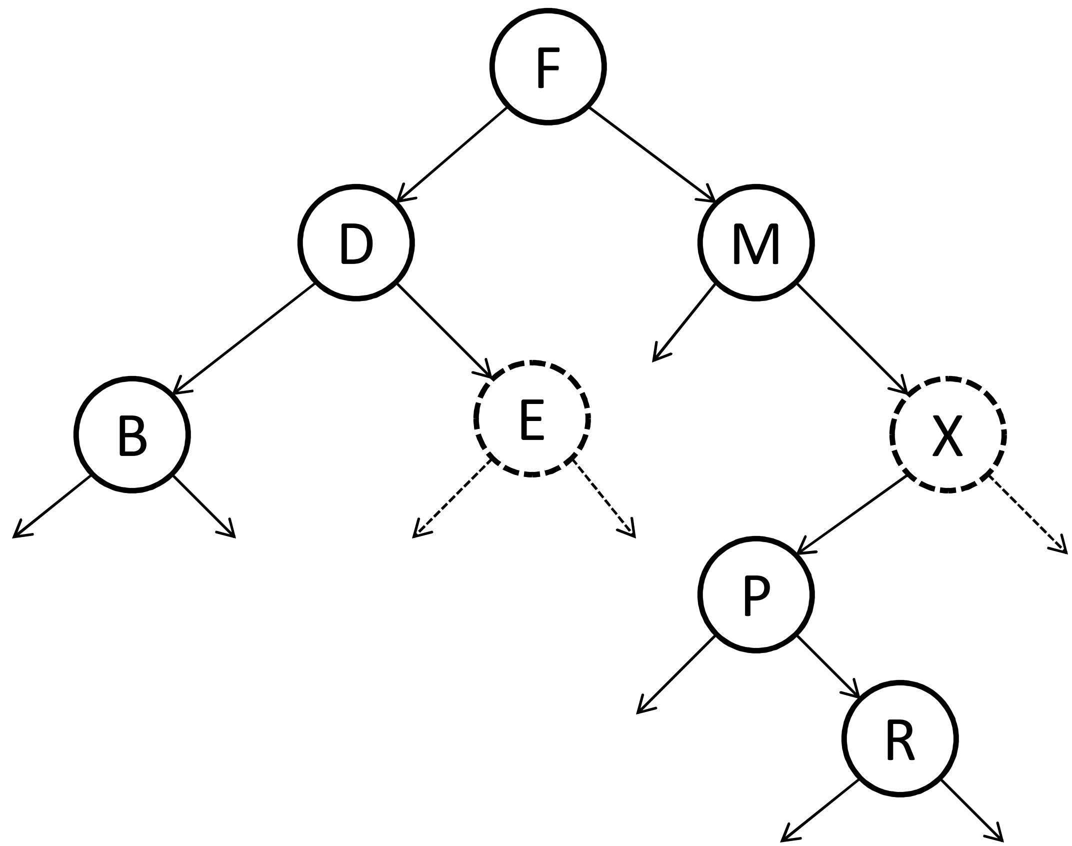
If leaf, delete node (replace by null).

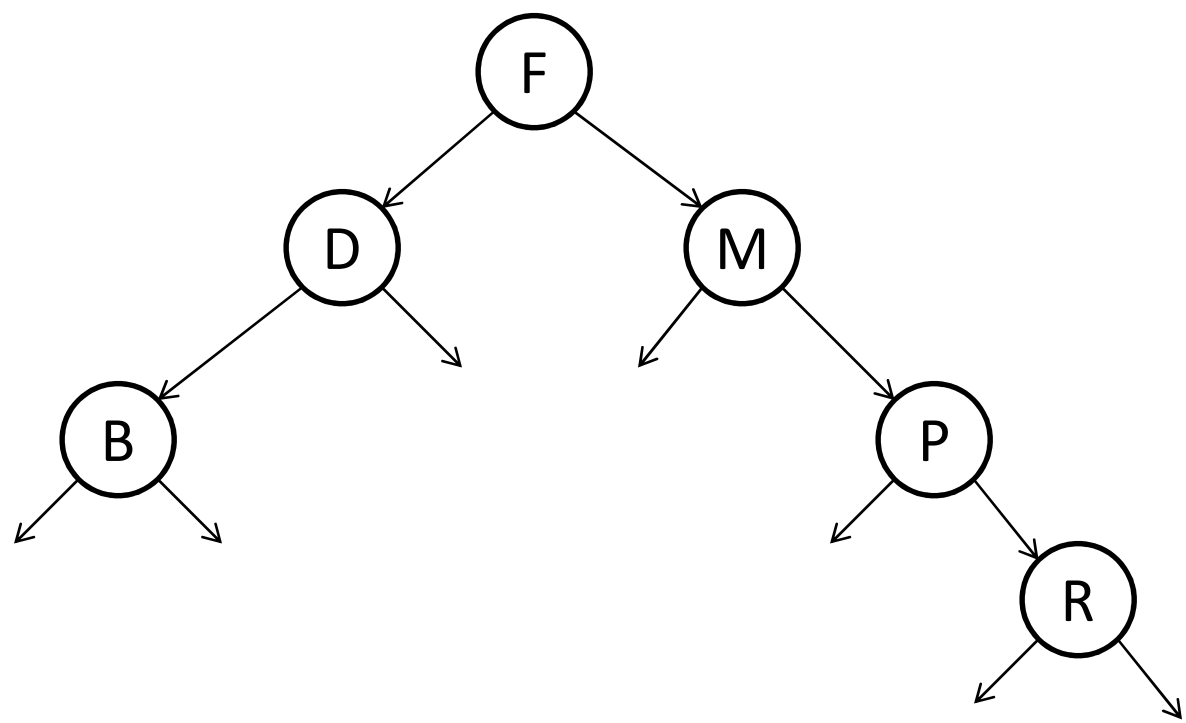
If unary, replace by other child.

If binary?

Delete E

Delete X



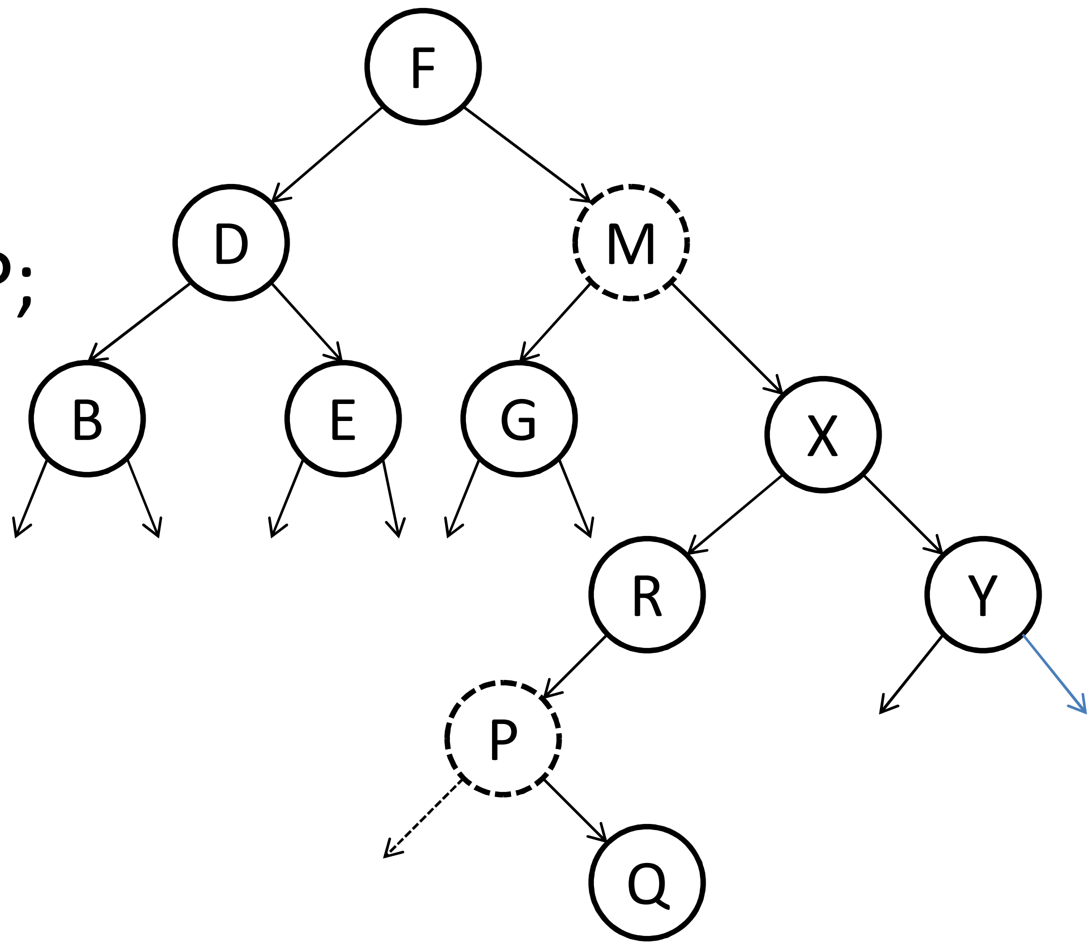

F
D
M
B
P
R

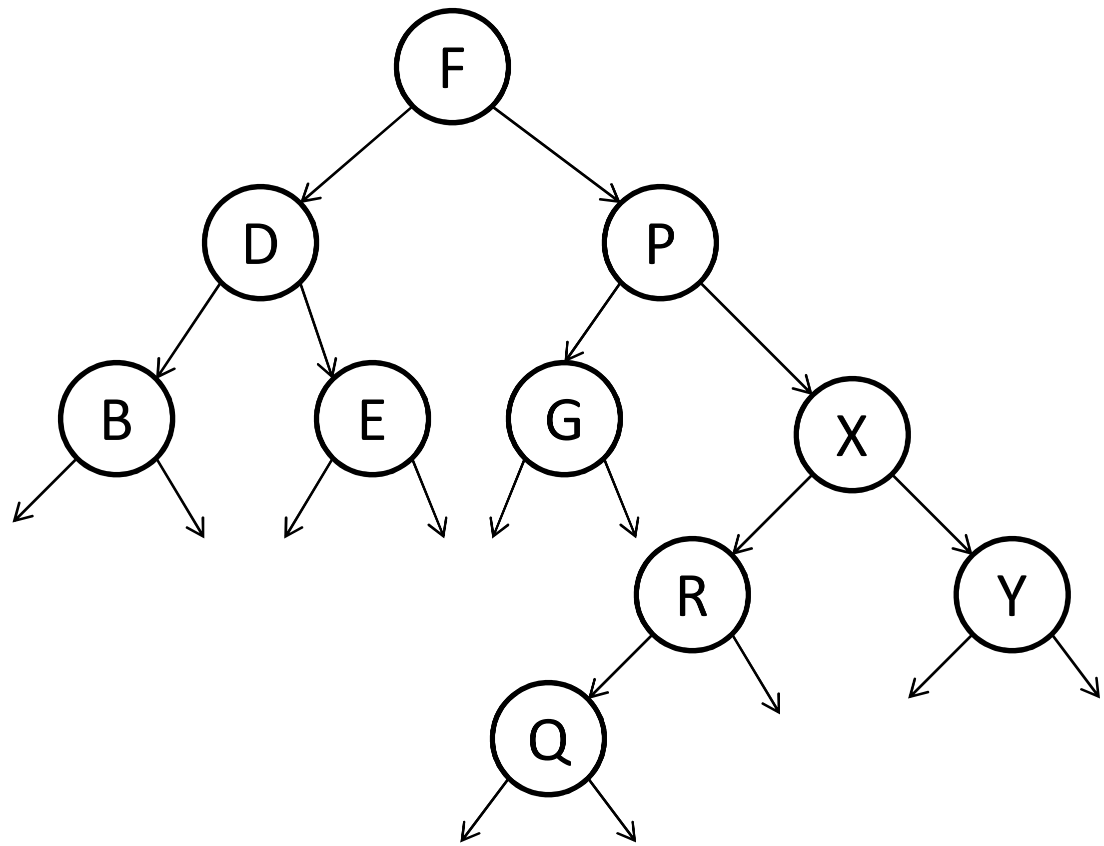
If binary, swap with successor (or predecessor). Now leaf or unary node; delete. To find successor, follow left path from right child.

Delete M:

Swap with P;

delete.



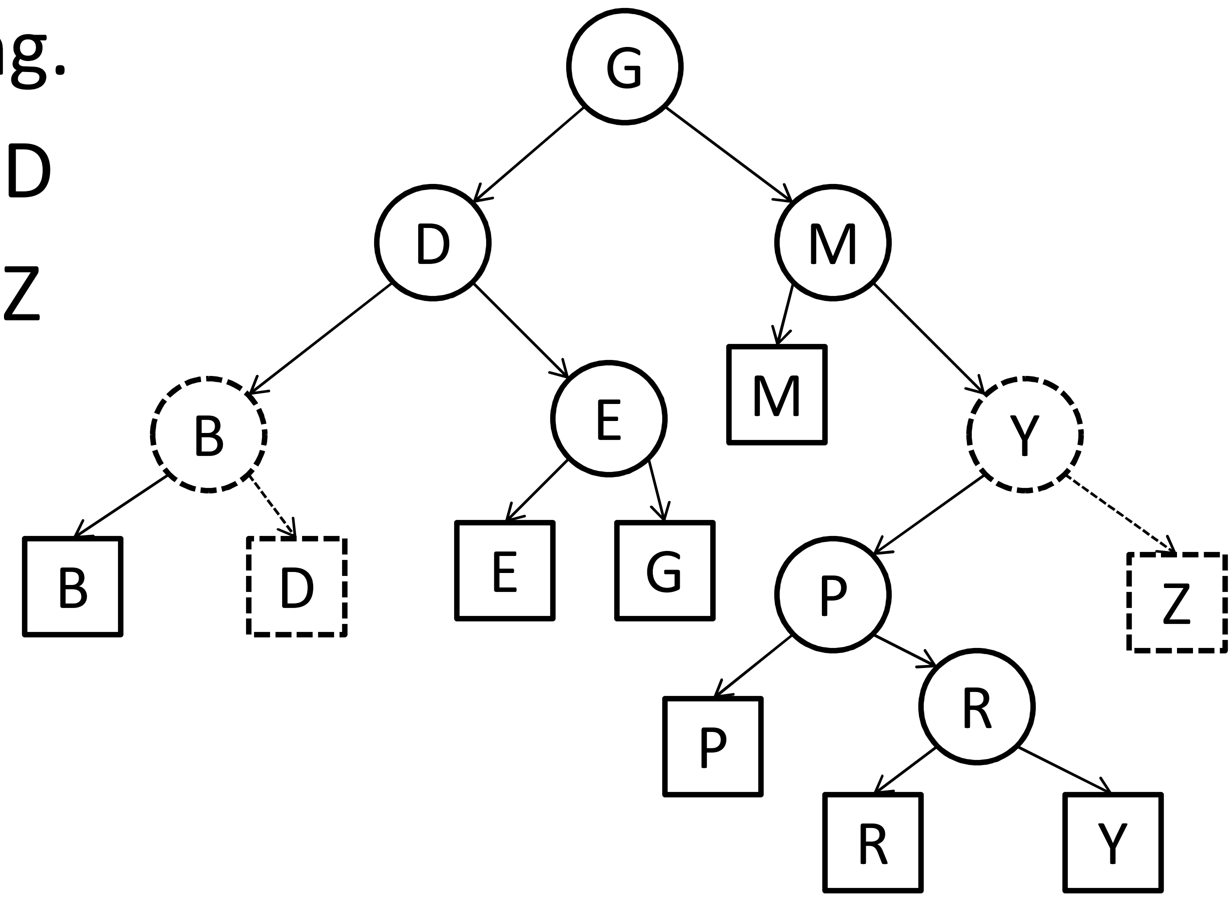

F
D
P
B
E
G
X
R
Y
Q

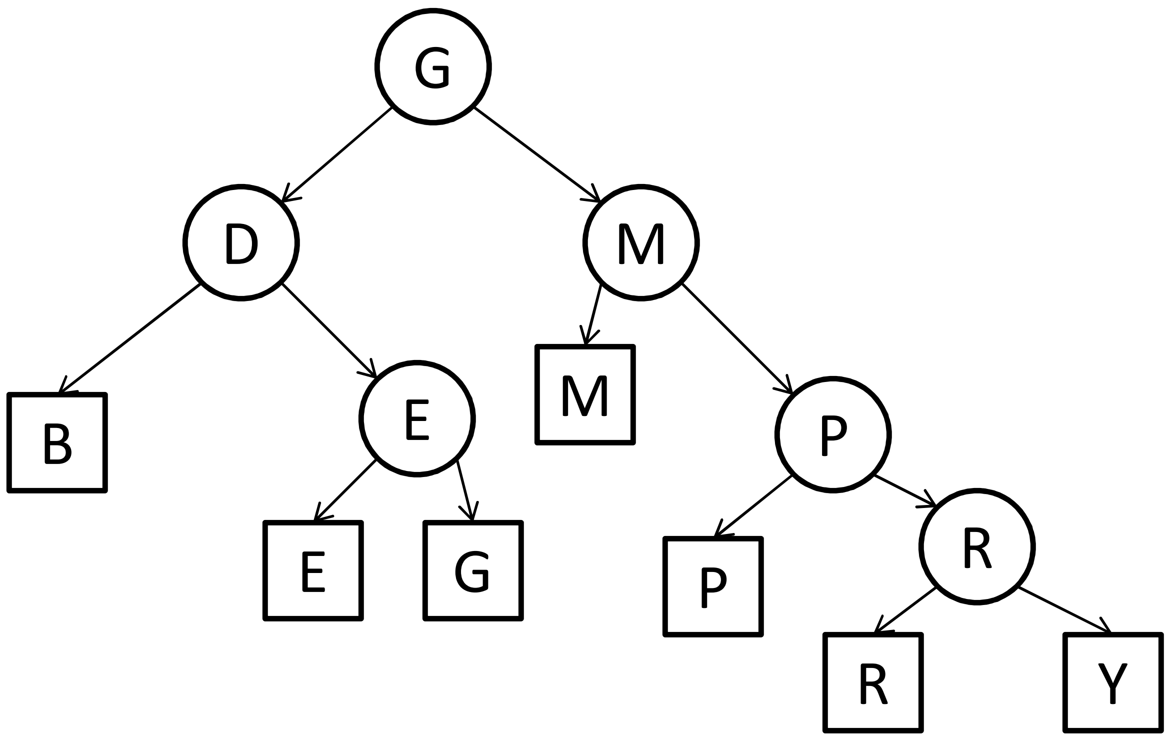
External representation: deletion simpler

Delete node and parent; replace parent by sibling.

Delete D

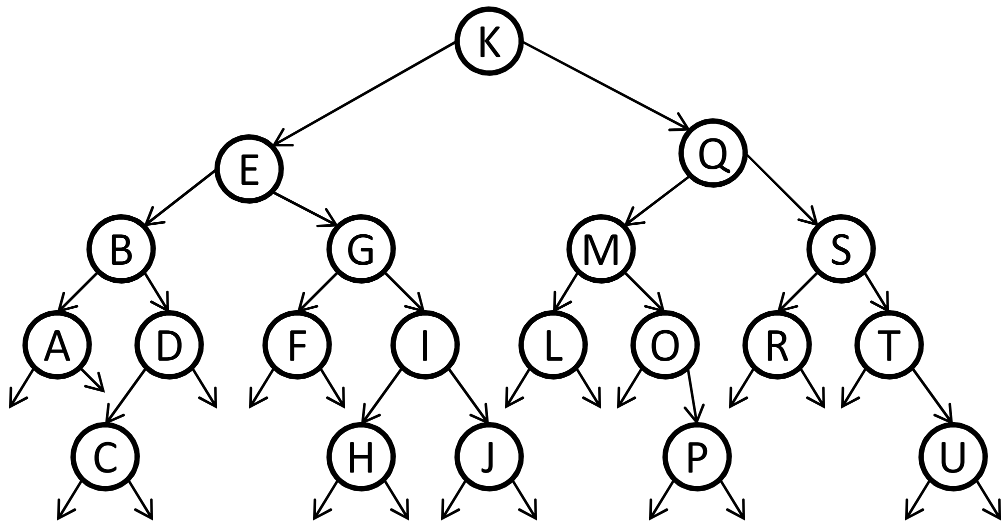
Delete Z




G
D
M
B
E
M
P
E
G
P
R
R
Y

Best case

All leaves have depths within 1: depth $\lfloor \lg n \rfloor$.
(*lg*: base-two logarithm)

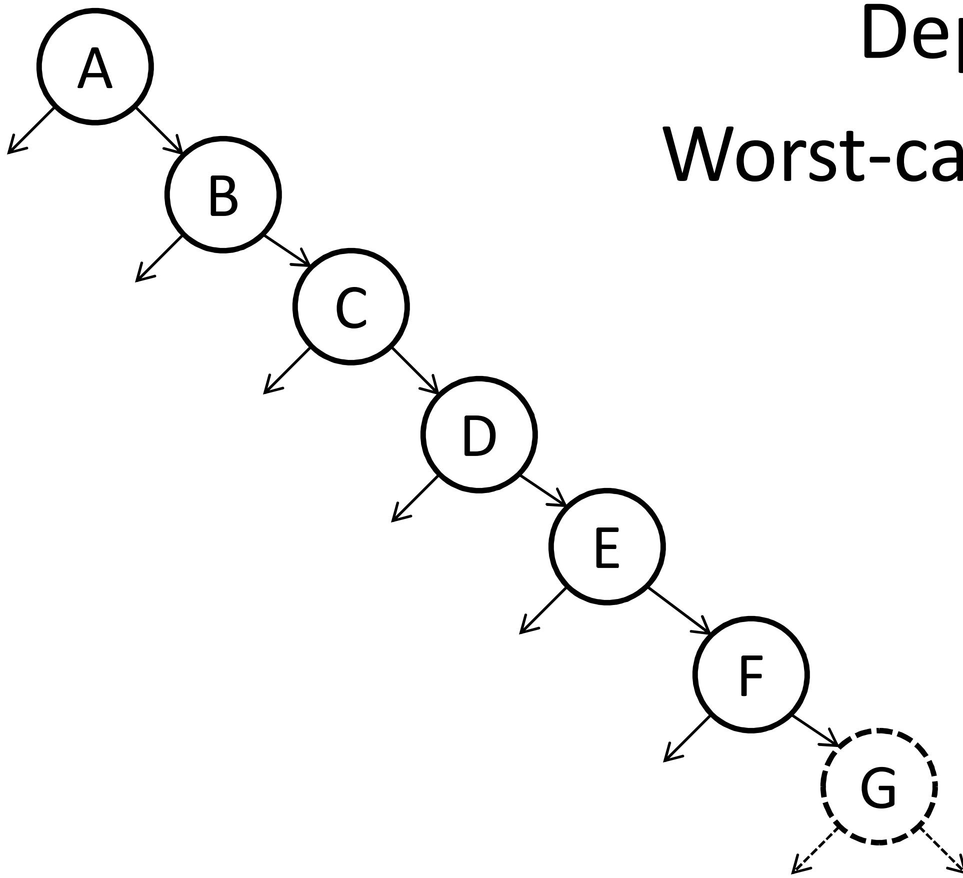


Can achieve if tree is static (insertion order chosen by implementation, no deletions)

Worst case

Natural but bad insertion order: sorted.

Insert A, B, C, D, E, F, G,…



Depth of tree is $n - 1$.

Worst-case access cost is n.

= list!