Data Structures & Algorithms for Problem Solving (CS1.304)

Lecture # 05

Avinash Sharma

Center for Visual Information Technology (CVIT),

IIIT Hyderabad

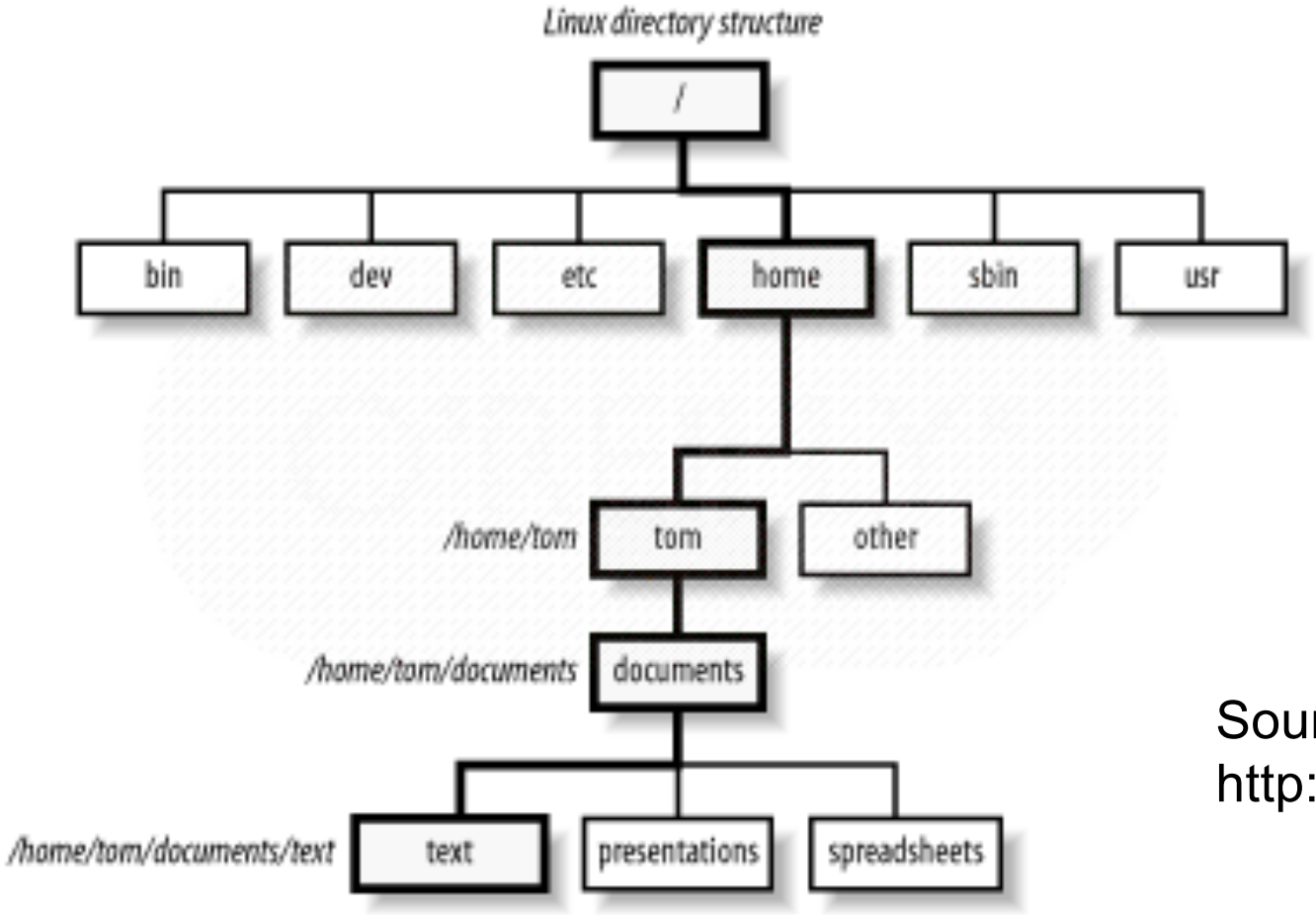
Introduction

- The story so far
  - Saw some fundamental operations as well as advanced operations on arrays, stacks, and queues
  - Saw a dynamic data structure, the linked list, and its applications.
- This week we will
  - Study data structures for hierarchical data
  - Operations on such data.
  - Leading to efficient insert/delete/find.

Motivation

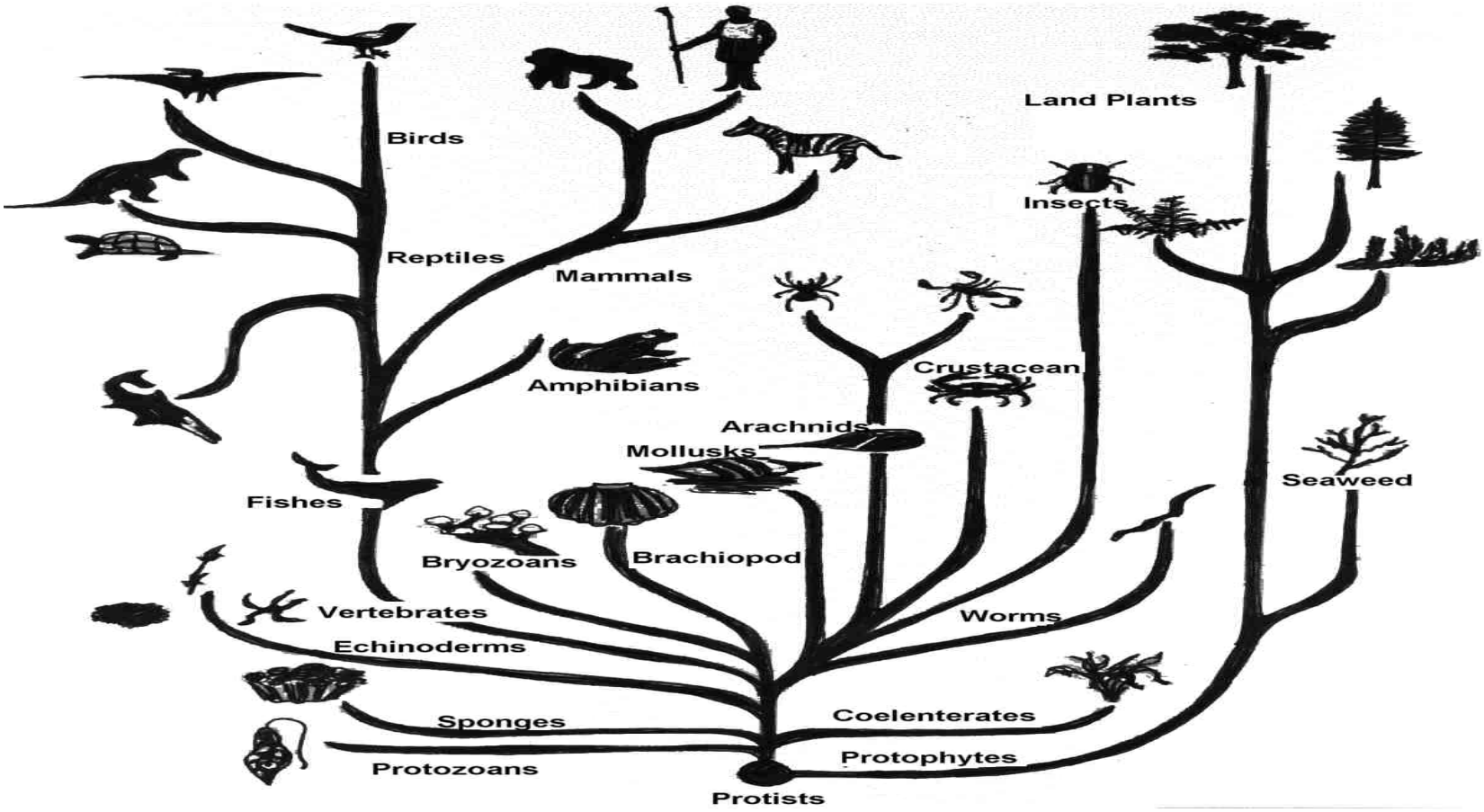
- Consider your home directory.
- /home/user is a directory, which can contain sub-directories such as work/, misc/, songs/, and the like.
- Each of these sub-directories can contain further sub-directories such as ds/, maths/, and the like.
- An extended hierarchy is possible, until we reach a file.

Motivation



Source:
http://khalihari.blogspot.in/

Motivation


Birds
Reptiles
Mammals
Amphibians
Fishes
Land Plants
Insects
Crustacean
Arachnids
Mollusks
Seaweed
Bryozoans
Brachiopod
Vertebrates
Worms
Echinoderms
Sponges
Coelenterates
Protophytes
Protozoans
Protists

Motivation

- In all of the above examples, there is a natural hierarchy of data.
  - In the first example, a (sub)directory can have one or more sub-directories.
- Similarly, there are several setting where there is a natural hierarchy among data items.
  - Family trees with parents, ancestors, siblings, cousins,...
  - Hierarchy in an organization with CEO/CTO/Managers/...

Motivation

- What kind of questions arise on such hierarchical data?
  - Find the number of levels in the hierarchy between two data items?
  - Print all the data items according to their level in the hierarchy.
  - Where from two members of the hierarchy trace their first common member in the hierarchy. Put differently, in the evolution process, when did man and amphibians start to branch out?

Motivation

- As a data structure question
  - How to formalize the above notions? Plus,
  - How can more members be added to the hierarchy?
  - How can existing data items be deleted from the hierarchy?
- Let's study a data structure that can handle hierarchical data.
- Study several applications of the data structure including those to:
  - expression verification and evaluation
  - searching

The Tree Data Structure

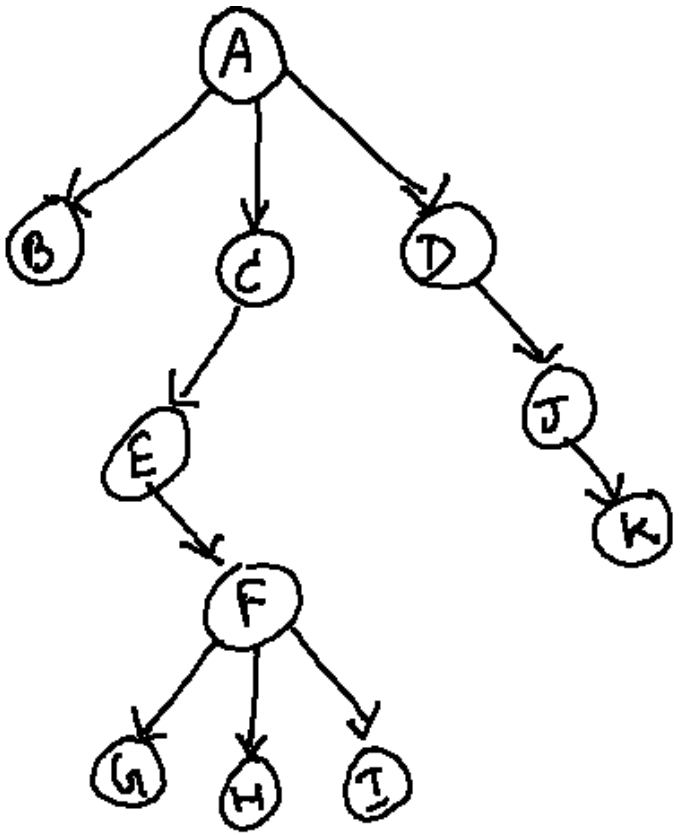
- Our new data structure will be called a tree.
- Defined as follows.
  - A tree is a collection of nodes.
  - An empty collection of nodes is a tree.
  - Otherwise a tree consists of a distinguished node r, called the root, and 0 or more non-empty (sub)trees $T_1, T_2, \cdots, T_k$ each of whose roots $r_1, r_2, ..., r_k$ are connected by a directed edge from r.
  - r is also called as the parent of the the nodes $r_1, r_2, ..., r_k$.

The Tree Data Structure

- A tree on n nodes always has n-1 edges.
- Why?
  - One parent for every one, except the root.
- Before going in to how a tree can be represented, let us know more about the tree.

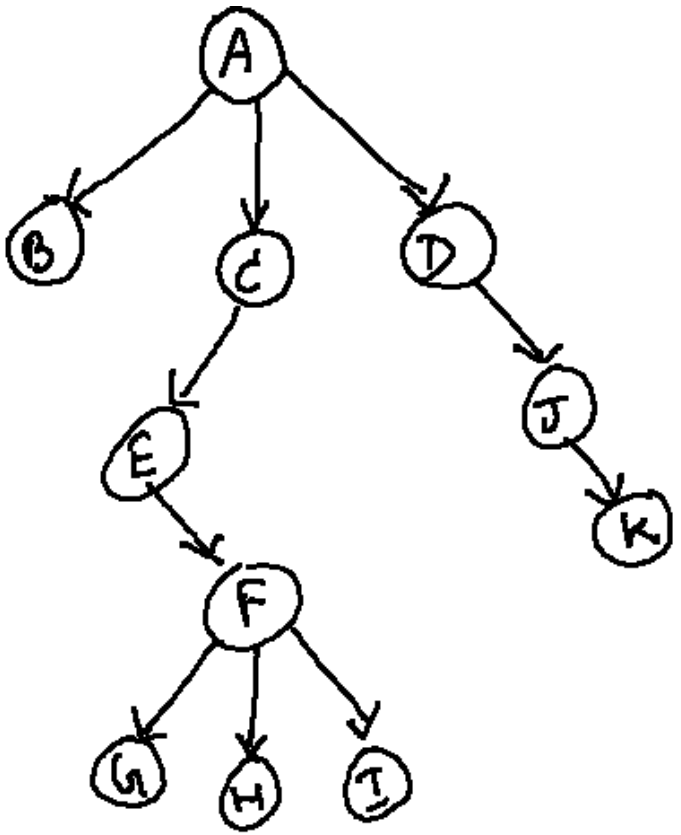
The Tree Data Structure

- Consider the tree shown to the right.
- The node A is the root of the tree.
- It has three subtrees whose roots are B, C, and D.
- Node C has one subtree with node E as the root.



The Tree Data Structure

- Nodes with the same parent are called as siblings.
- In the figure, G, H, and I are siblings.
- Nodes with no children are called leaf nodes or pendant nodes.
  - In the figure, B, G, H, I and K are leaf nodes.



A Few More Terms : Height, Level, and Path

- A path from a node u to a node v is a sequence of nodes $u=u_0, u_1, u_2, ..., u_k = v$ such that u_i is the parent of u_{i+1}, $i > 0$.
  - The path is said to have a length of k, the number of edges in the path.
  - A path from a node to itself has a length of 0.
- Example: A path from node C to F in our earlier tree is C->E->F.
- Observation: In any tree there is exactly one path from the root to any other node.

Depth

- Given a tree T, let the root node be said to be at a depth of 0.
- The depth of any other node u in T is defined as the length of the path from the root to u.
- Example: Depth of node G = 4.
- Alternatively, let the depth of the root be set to 0 and the depth of a node is one more than the depth of its parent.

Height

- Another notion defined for trees is the height.
- The height of a leaf node is set to 0. The height of a node is one plus the maximum height of its children.
- The height of a tree is defined as the height of the root.
- Example: Height of node C = 3.

Ancestors and Descendants

- Recall the parent-child relationship between nodes.
- Alike parent-children relationship, we can also define ancestor-descendant relationship as follows.
- In the path from node u to v, u is an ancestor of v and v is a descendant of u.
- If $u \neq v$, then u (v) is called a proper ancestor (descendant) respectively.

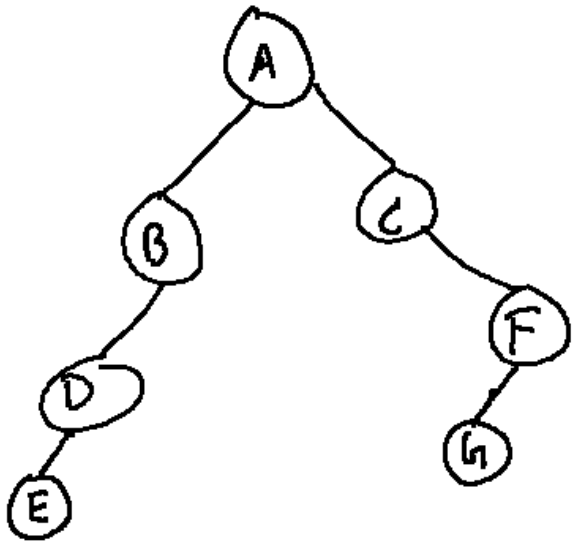
Implementing Trees

- Briefly, we also mention how to implement the tree data structure.
- The following node declaration as a structure works.

```
struct node
{
    int data;
    node *children;
}
```

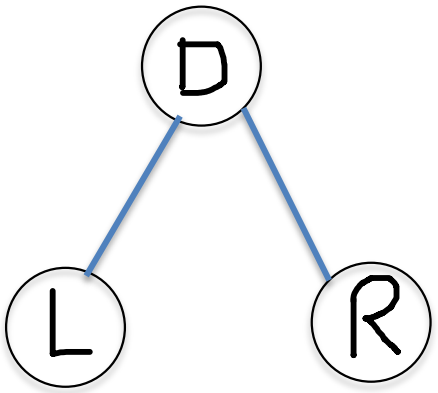
Binary Trees

- A special class of the general trees.
- Restrict each node to have at most two children.
  - These two children are called the left and the right child of the node.
  - Easy to implement and program.
  - Still, several applications.



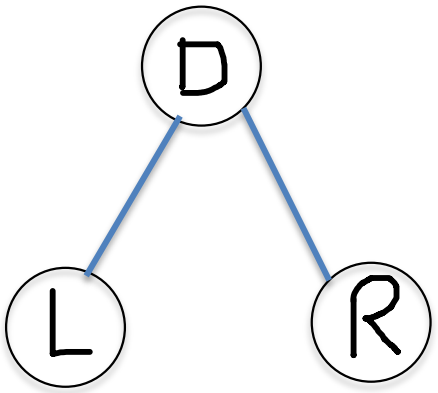
Our First Operation

- To print the nodes in a (binary) tree
- This is also called as a traversal.
- Need a systematic approach
  - ensure that every node is indeed printed
  - and printed only once.
- Several methods possible. Attempt a categorization.
- Consider a tree with a root D and L, R being its left and right sub-trees respectively.



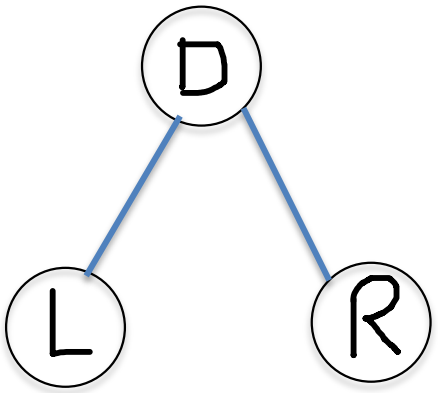
Tree Traversal

- Should we intersperse elements of L and R during the traversal?
  - OK – One kind of traversal.
  - NO. – One kind of traversal.
  - Let us study the latter first.
- When items in L and R should not be interspersed, there are six ways to traverse the tree.
  - List the six ways.



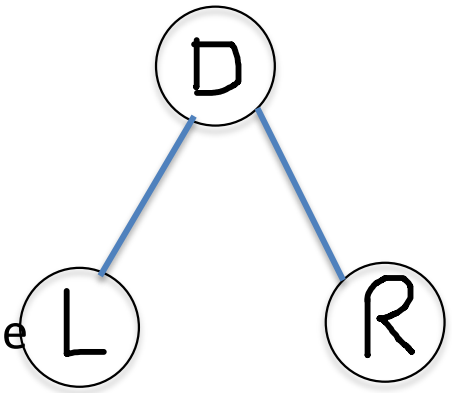
Tree Traversal

- Of these, let us make a convention that R can not precede L in any traversal.
- We are left with three:
  - L R D
  - L D R
  - D L R
- We will study each of the three. Each has its own name.



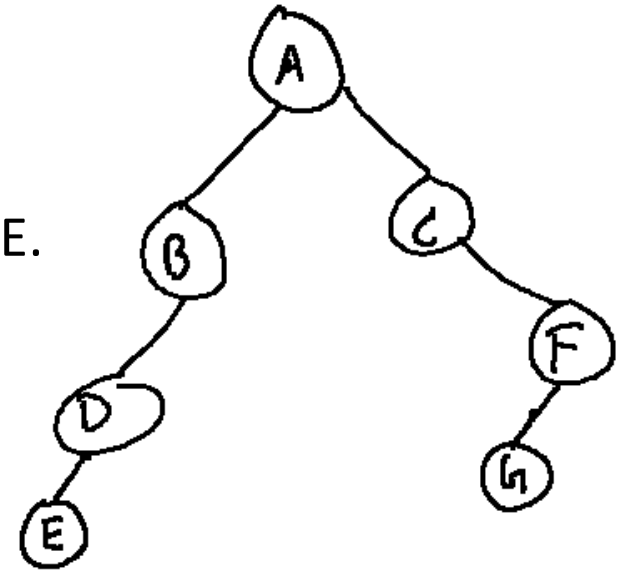
The Inorder Traversal (LDR)

- The traversal that first completes L, then prints D, and then traverses R.
- To traverse L, use the same order.
  - First the left subtree of L, then the root of L, and then the right subtree of L.



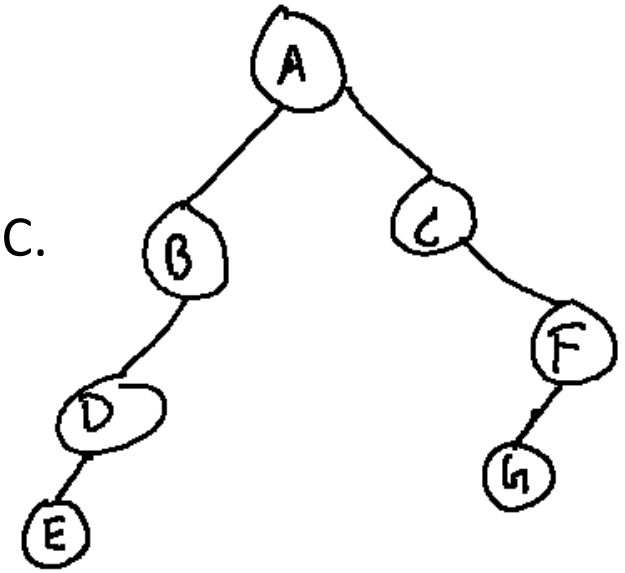
The Inorder Traversal

- Start from the root node A.
- We first should process the left subtree of A.
- Continuing further, we first should process the node E.
- Then come D and B.
- The L part of the traversal is thus E D B.



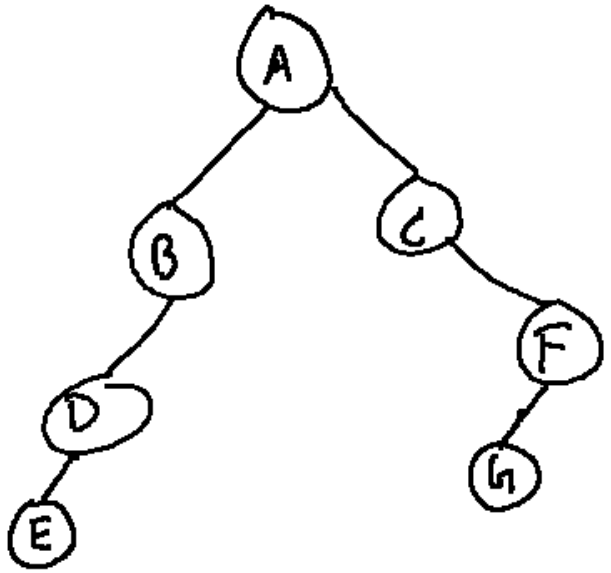
The Inorder Traversal

- Then comes the root node A.
- We next process the right subtree of A.
- Continuing further, we first should process the node C.
- Then come G and F.
- The R part of the traversal is thus C G F.



The Inorder Traversal

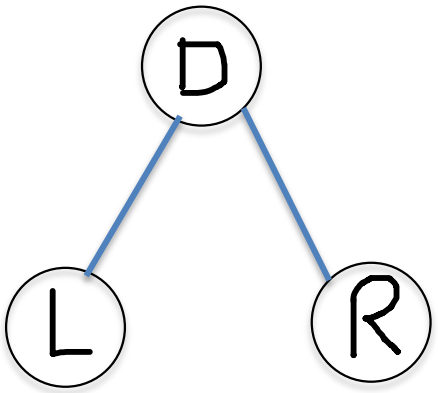
```
Procedure Inorder(T)
begin
        if T == NULL return;
        Inorder(T->left);
        print(T->data);
        Inorder(T->right);
end
```



Inorder: E D B A C G F

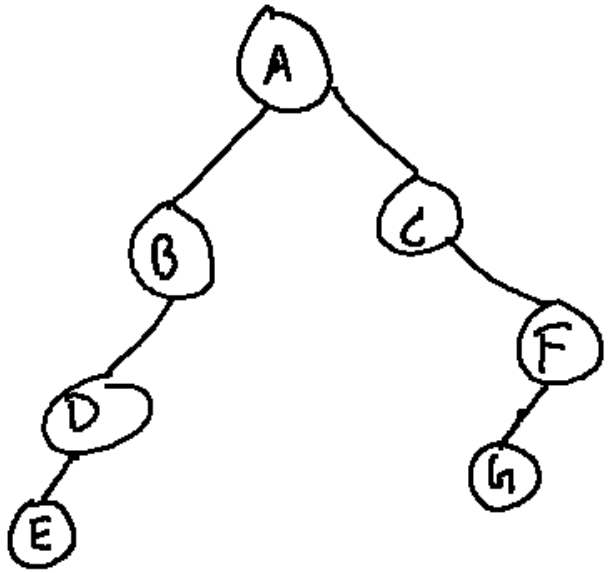
The Preorder Traversal (DLR)

- The traversal that first completes D, then prints L, and then traverses R.
- To traverse L (or R), use the same order.
  - First the root of L, then left subtree of L, and then the right subtree of L.



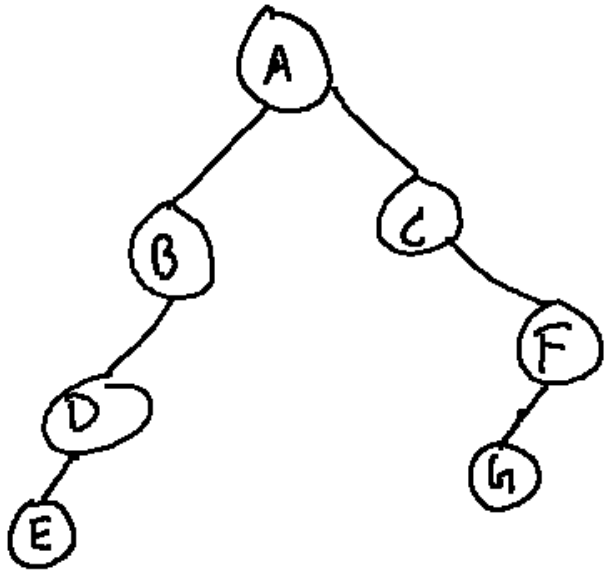
The Preorder Traversal

- Start from the root node A.
- We first should process the root node A.
- Continuing further, we should process the left subtree of A.
- This suggests that we should print B, D, and E in that order.
- The L part of the traversal is thus B D E.



The Preorder Traversal

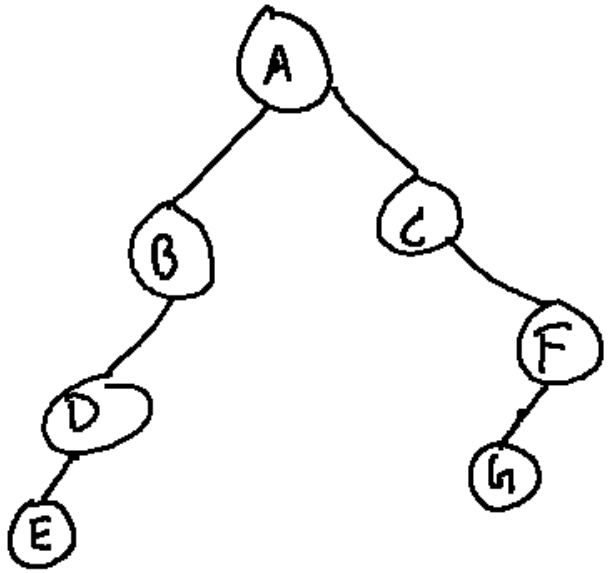
- We first next process the right subtree of A.
- Continuing further, we first should process the node C.
- Then come F and G in that order.
- The R part of the traversal is thus C F G.



Preorder: A B D E C F G

The Preorder Traversal

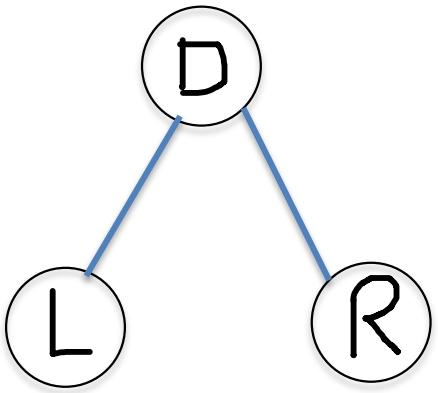
```
Procedure Preorder(T)
begin
        if T == NULL return;
        print(T->data);
        Preorder(T->left);
        Preorder(T->right);
end
```



Preorder: A B D E C F G

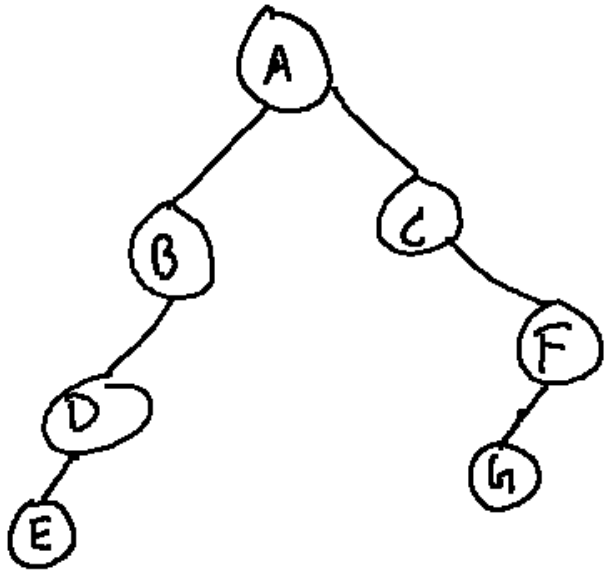
The Postorder Traversal (LRD)

- The traversal that first completes L, then traverses R, and then prints D.
- To traverse L, use the same order.
  - First the left subtree of L, then the right subtree of R, and then the root of L.



The Postorder Traversal

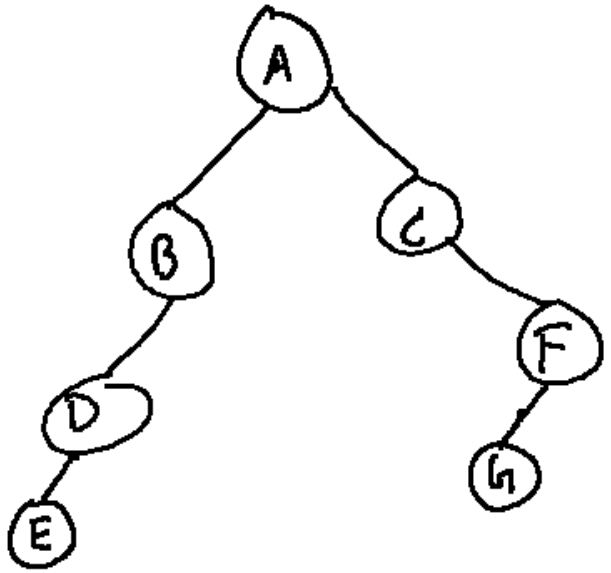
- We next process the right subtree of A.
- Continuing further, we first should process the node C.
- Then come G and F.
- The R part of the traversal is thus G F C.
- Then comes the root node A.



postorder: E D B G F C A

The Postorder Traversal

```
Procedure postorder(T)
begin
        if T == NULL return;
        Postorder(T->left);
        Postorder(T->right);
        print(T->data);
end
```



postorder: E D B G F C A

Another Kind of Traversal

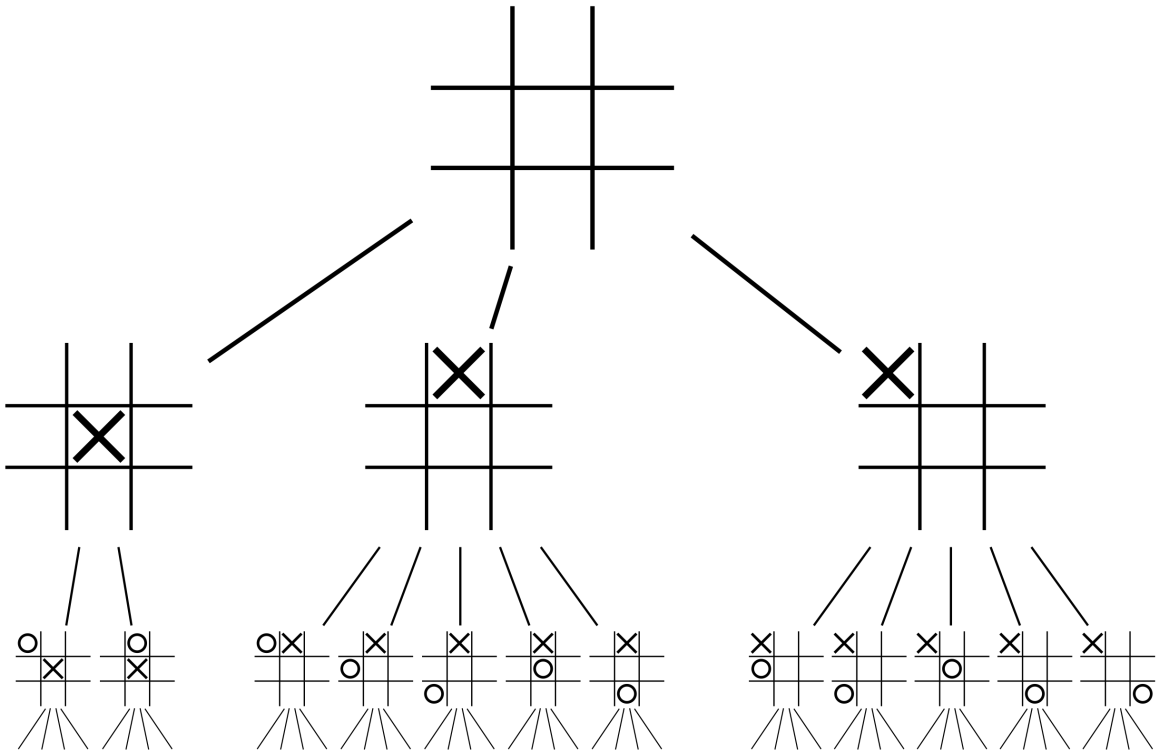
- When left and right subtree nodes can be intermixed.
- One useful traversal in this mode is the level order traversal.
- The idea is to print the nodes in a tree according to their level starting from the root.

Another Kind of Traversal

- Why would any one want to do that?
- One example:
  - Think of printing the organization chart.
  - Start with the CEO, there are CTO, CFO, and COO, say.
  - Then, five managers under the CTO, 2 managers under the CFO, and so on,
  - Each manager has more Assistant Managers who work with a team.
  - Want to list this in that order.
- There are other such examples too
  - Game trees

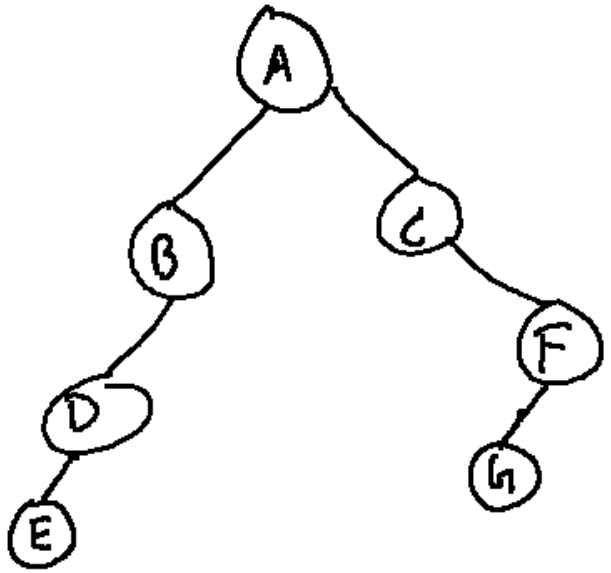
How to Perform a Depth Order Traversal

- Game Tree Example



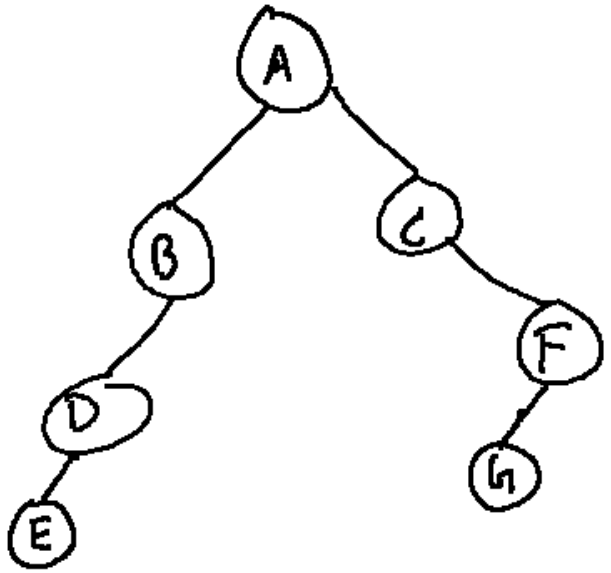
How to Perform a Depth Order Traversal

- Consider the same example tree.
- Starting from the root, so A is printed first.
- What should be printed next?
- Assume that we use the left before right convention.
- So, we have to print B next.
- How to remember that C follows B.
- And then D should follow C?



How to Perform a Depth Order Traversal

- Indeed, can remember that B and C are children of A.
- But, have to get back to children of B after C is printed.
- For this, one can use a queue.
  - Queue is a first-in-first-out data structure.



How to Perform a Depth Order Traversal

- The idea is to queue-up children of a parent node that is visited recently.
- The node to be visited recently will be the one that is at the front of the queue.
  - That node is ready to be printed.
- How to initialize the queue?
  - The root node is ready!

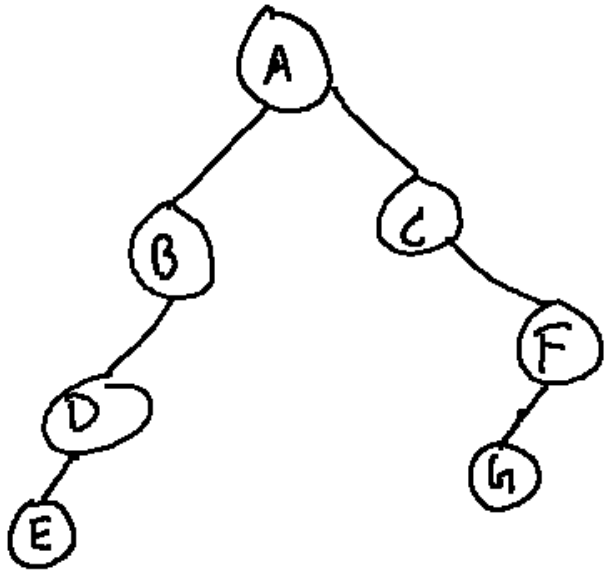
How to Perform a Depth Order Traversal

```
Procedure DepthOrder(T)
begin
    Q = queue;
    insert root into the queue;
    while Q is not empty do
        v = delete();
        print v->data;
        if v->left is not NULL insert v->left into Q;
        if v->right is not NULL insert v->right into Q;
    end-while
end
```

How to Perform a Depth Order Traversal

- Queue and output are shown at every stage.

Queue	Output
A	
B C	A
C D	B
D F	C
F E	D
E G	F
G	E
EMPTY	G



Analysis of Traversal Techniques

- For inorder, preorder, and postorder traversal, let the tree have n nodes of which n_1 are in the left subtree and the rest in the right subtree.
- Recurrence relation:

$$T(n) = T(n_1) + T(n-n_1-1) + O(1)$$

- Can solve by guessing that $T(n) \leq cn$ for constant c.
- Verify.

$$T(n) \leq cn_1 + c(n-n_1-1) + O(1) \leq cn, \text{ provided c is large enough.}$$

Analysis – Depth Order Traversal

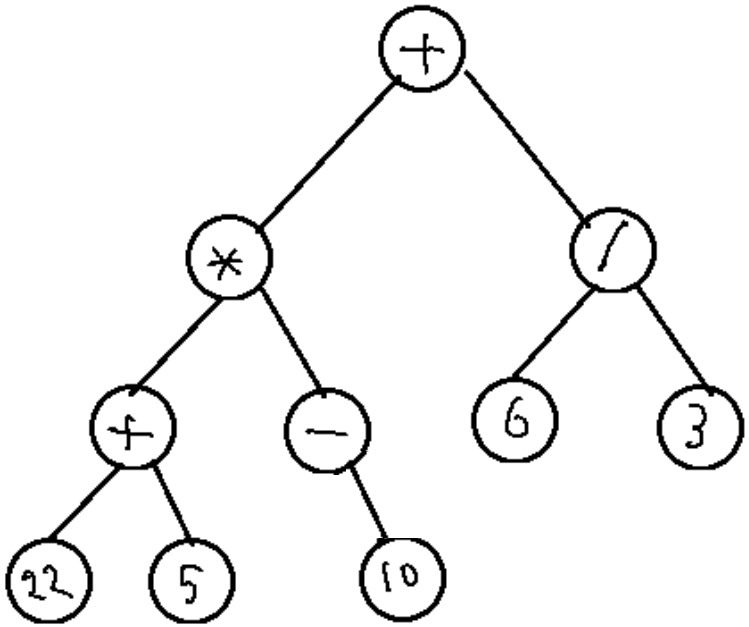
- How to analyze this traversal?
- Assume that the tree has n nodes.
- Each node is placed in the queue exactly once.
- The rest of the operations are all O(1) for every node.
- So the total time is O(n).
- This traversal can be seen as forming the basis for a graph traversal.

Application to Expression Evaluation

- We know what expression evaluation is.
- We deal with binary operators.
- An expression tree for a expression with only unary or binary operators is a binary tree where the leaf nodes are the operands and the internal nodes are the operators.

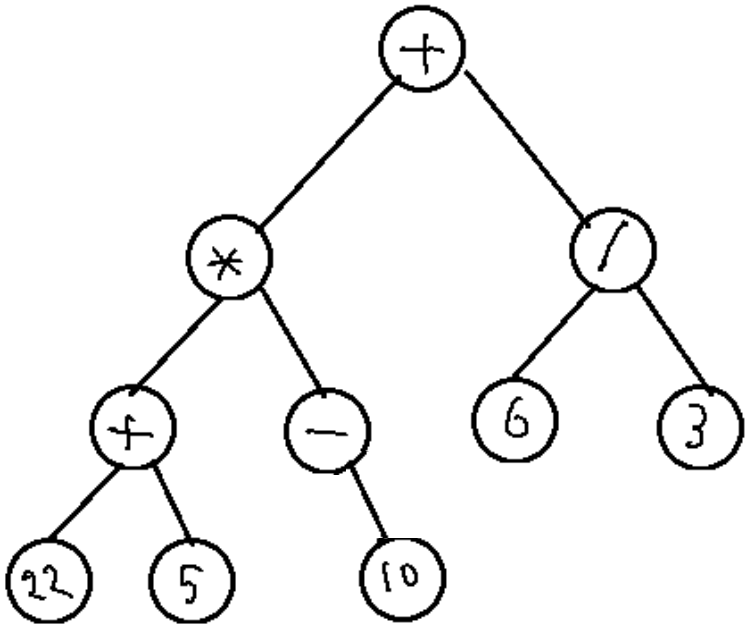
Example Expression Tree

- See the example to the right.
- The operands are 22, 5, 10, 6, and 3.
- These are also leaf nodes.



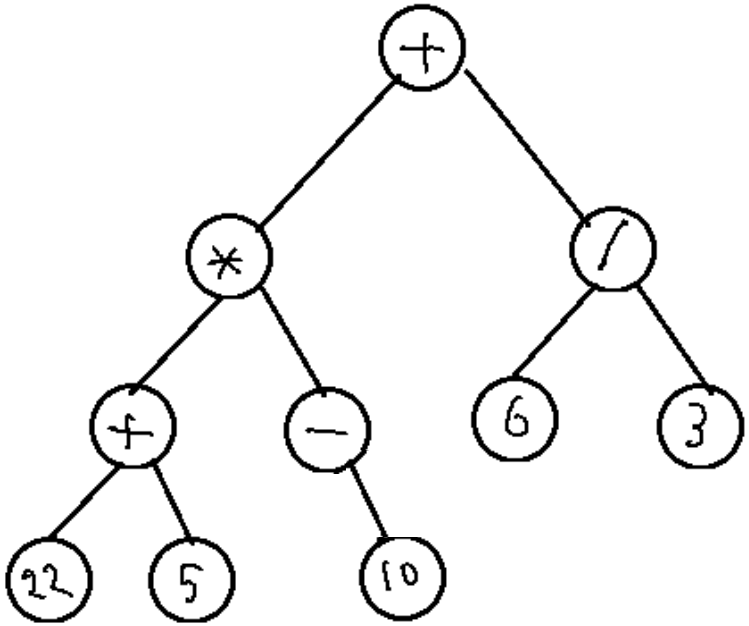
Questions wrt Expression Tree

- How to evaluate an expression tree?
  - Meaning, how to apply the operators to the correct set of operands.
- How to build an expression tree?
  - Given an expression, how to build an equival expression tree?



Questions wrt Expression Tree

- Notice that an inorder traversal of the expression tree gives an expression in the infix notation.
  - The above tree is equivalent to the expression ((22 + 5) × (−10)) + (6/3)
- What does a postorder and preorder traversal of the tree give?
  - Answer: ??



Why Expression Trees?

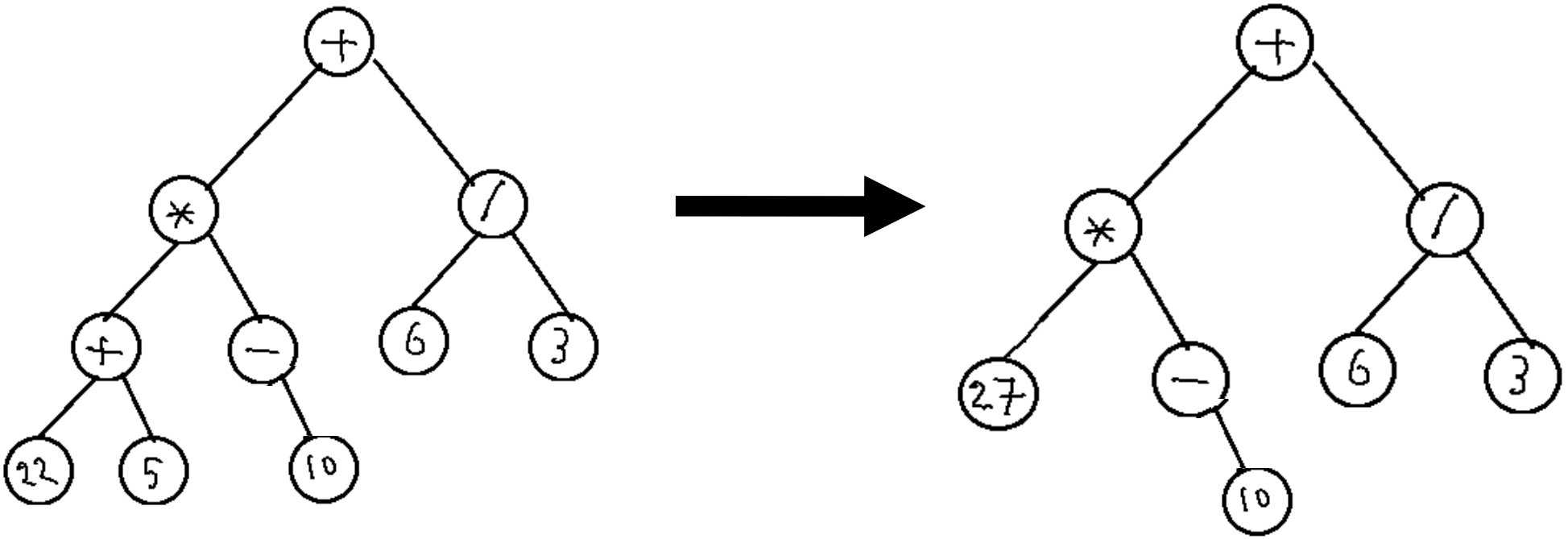
- Useful in several settings such as
  - compliers
  - can verify if the expression is well formed.

How to Evaluate using an Expression Tree

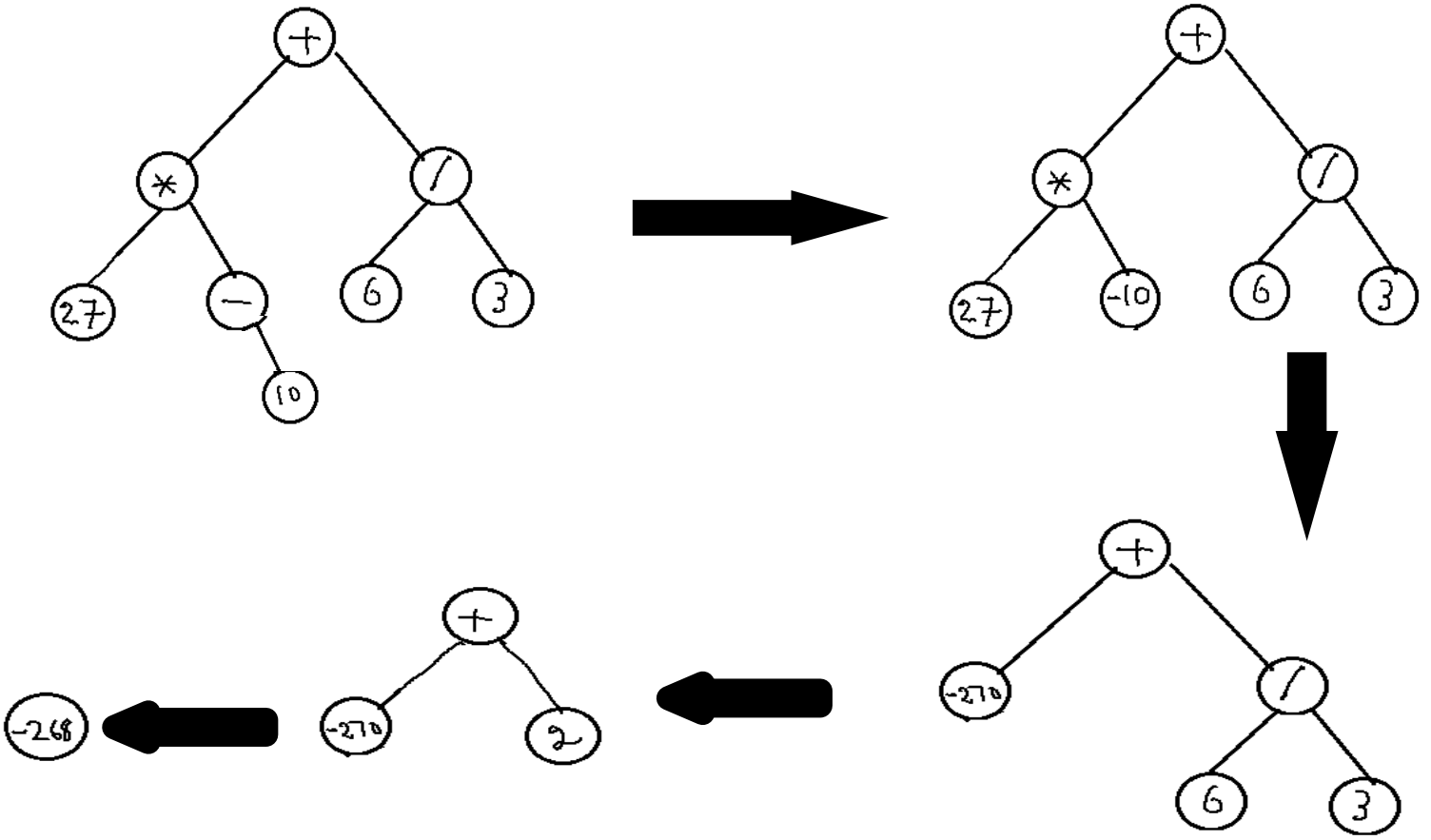
- Essentially, have to evaluate the root.
- Notice that to evaluate a node, its left subtree and its right subtree need to be operands.
- For this, may have to evaluate these subtrees first, if they are not operands.
- So, Evaluate(root) should be equivalent to:
  - Evaluate the left subtree
  - Evaluate the right subtree
  - Apply the operator at the root to the operands.

How to Evaluate using an Expression Tree

- This suggests a recursive procedure that has the above three steps.
- Recursion stops at a node if it is already an operand.



How to Evaluate using an Expression Tree



Pending Question

- How to build an expression tree?
- Start with an expression in the infix notation.
- Recall how we converted an infix expression to a postfix expression.
- The idea is that operators have to wait to be sent to the output.
  - A similar approach works now.

Building an Expression Tree

- Let us start with a postfix expression.
- The question is how to link up operands as (sub)trees.
- As in the case of evaluating a postfix expression, have to remember operators seen so far.
  - need to see the correct operands.
- A stack helps again.
- But instead of evaluating subexpression, we have to grow them as trees.
  - Details follow.

Building an Expression Tree

- When we see an operand :
  - That could be a leaf node...Or a tree with no children.
  - What is its parent?
  - Some operator.
  - In our case, operands can be trees also.
- The above observations suggest that operands should wait on the stack.
  - Wait as trees.

Building an Expression Tree

- What about operators?
- Recall that in the postfix notation, the operands for an operator are available in the immediate preceding positions.
- Similar rules apply here too.
- So, pop two operands (trees) from the stack.
- Need not evaluate, but create a bigger (sub)tree.

Building an Expression Tree

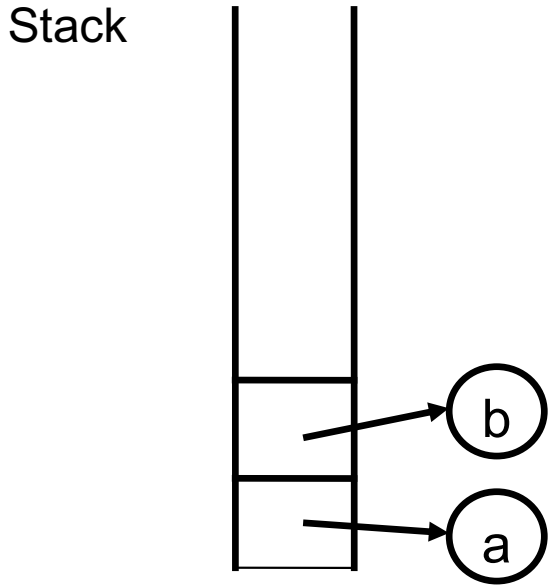
```
Procedure ExpressionTree(E)
//E is an expression in postfix notation.
begin
    for i=1 to |E| do
        if E[i] is an operand then
            create a tree with the operand as the only node;
            add it to the stack
        else if E[i] is an operator then
            pop two trees from the stack
            create a new tree with E[i] as the root and the two trees popped as its children;
            push the tree to the stack
    end-for
end
```

Building an Expression Tree

- Consider the expression (a + b – f) / (c x d + e)
- The postfix of the expression is a b + f – c d × e + /
- Let us follow the above algorithm.

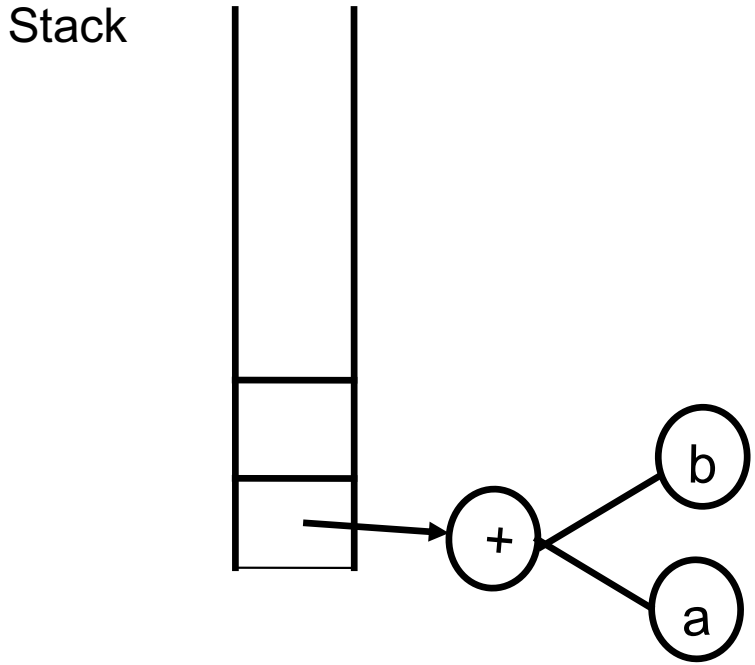
Building an Expression Tree

+ f – c d × e + /



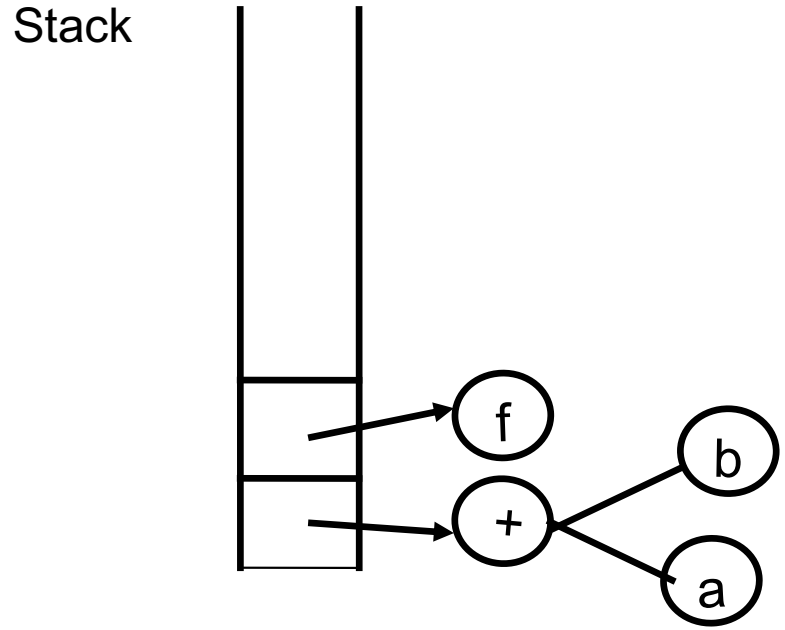
Building an Expression Tree

f – c d × e + /



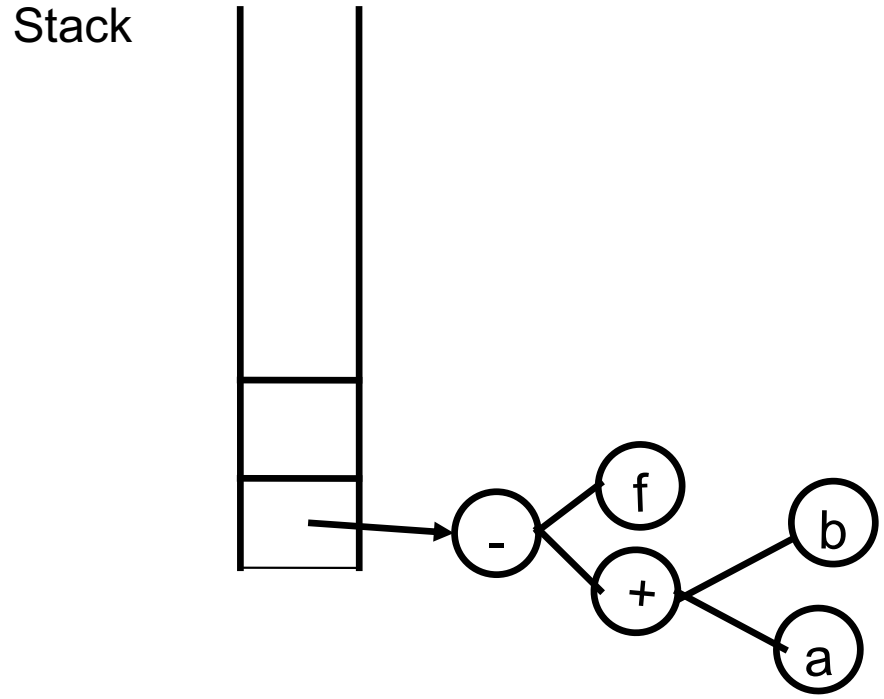
Building an Expression Tree

– c d × e + /

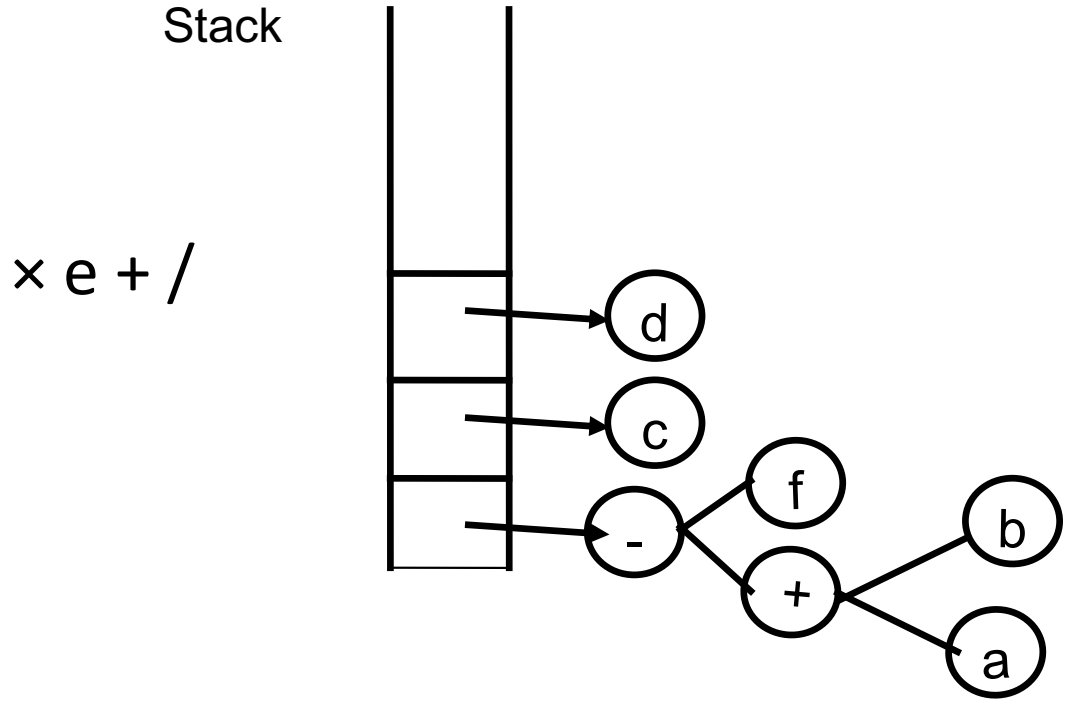


Building an Expression Tree

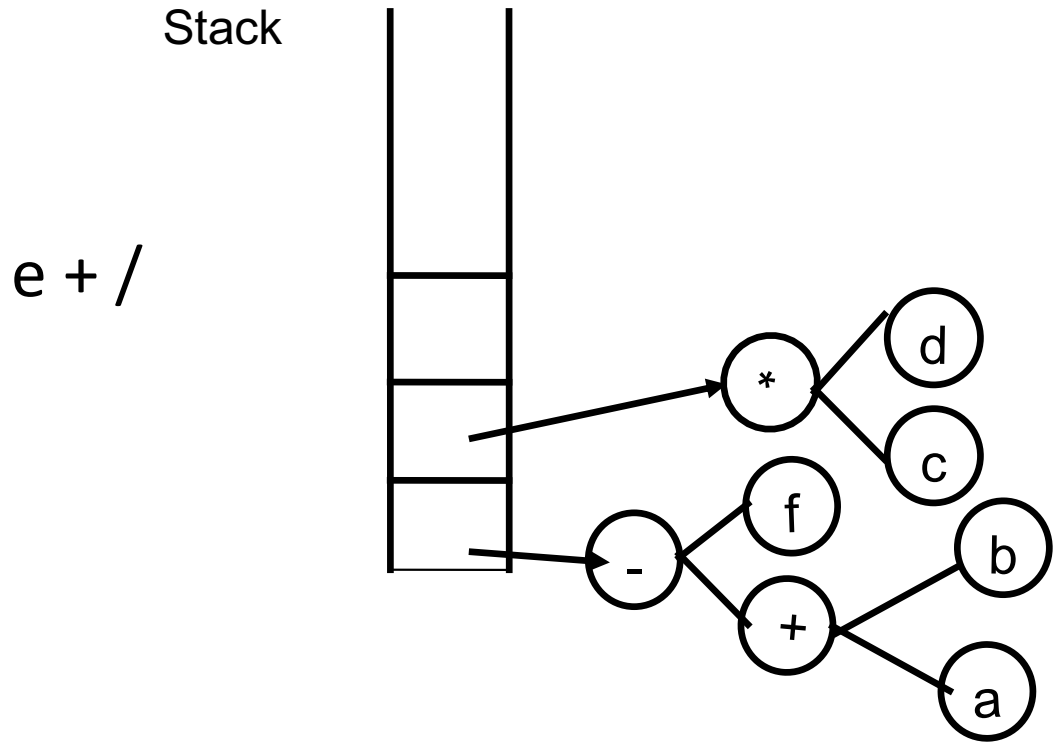
c d × e + /



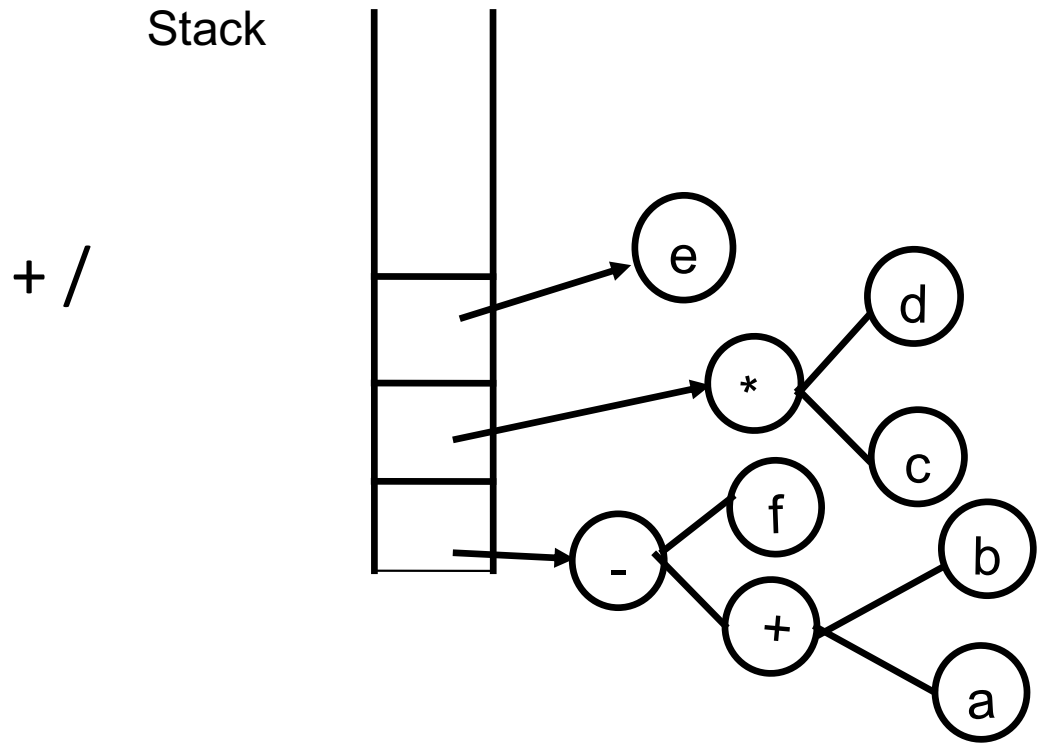
Building an Expression Tree



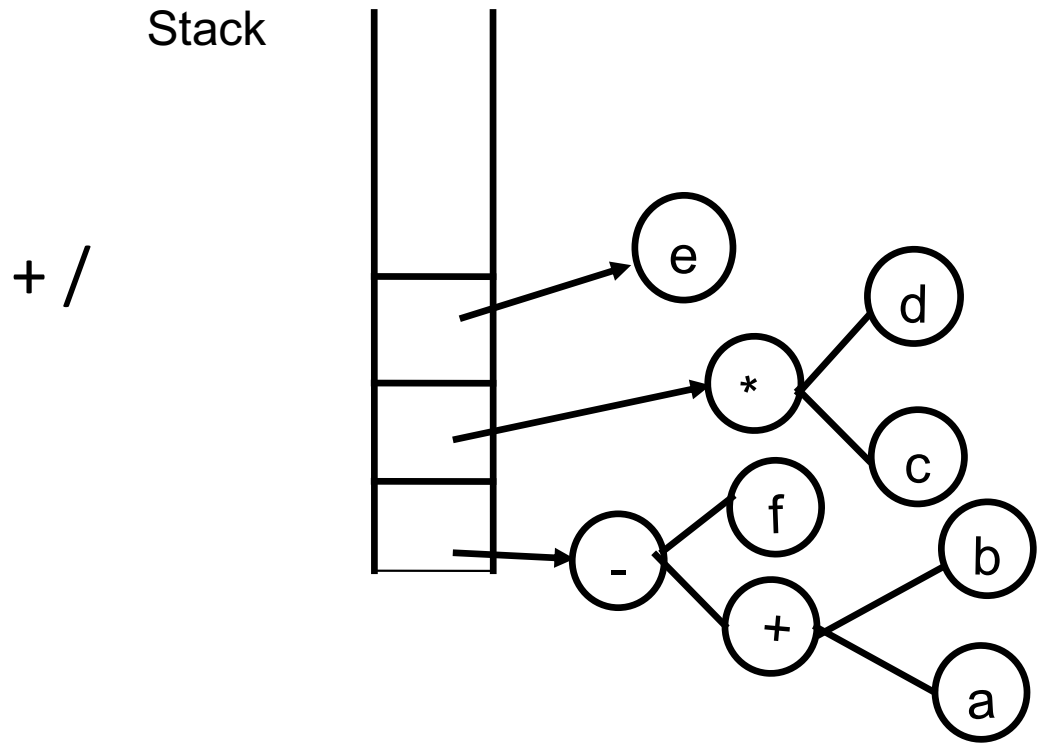
Building an Expression Tree



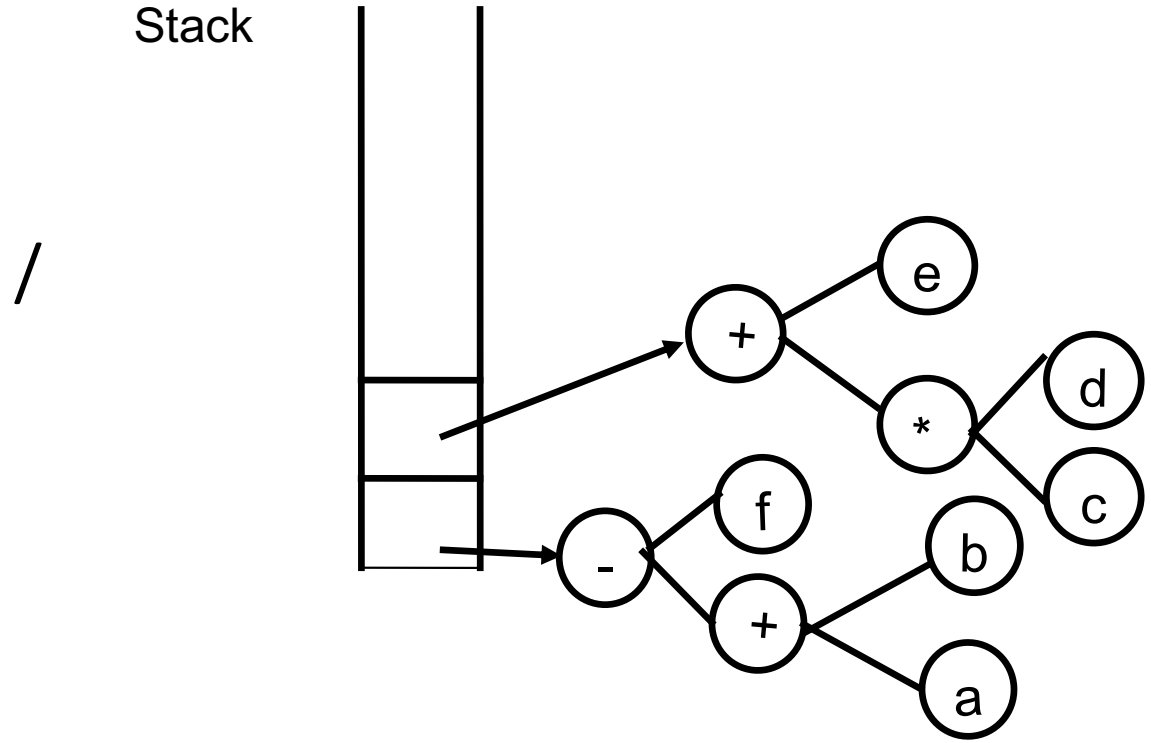
Building an Expression Tree



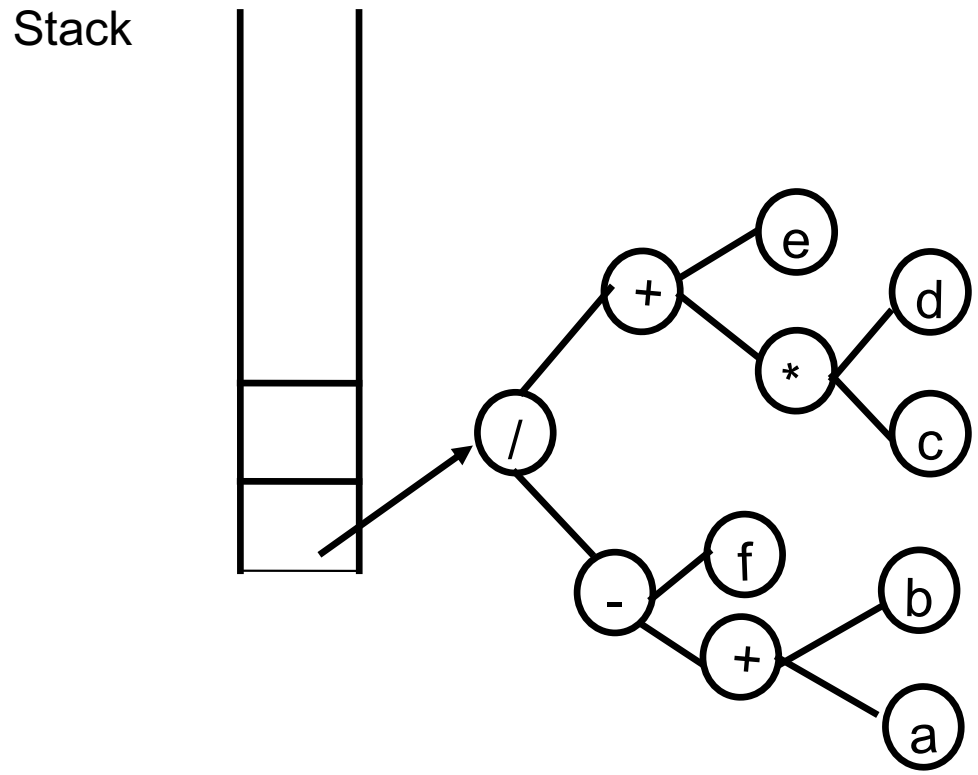
Building an Expression Tree



Building an Expression Tree



Building an Expression Tree



Another Application – Dictionary Operations

- Consider designing a data structure for primarily three operations:
  - insert,
  - delete, and
  - search.

Dictionary Operations

- Further extend the repertoire of operations to standard dictionary operations also such as findMin and findMax.
- Specifically, our data structure shall support the following operations.
  - Create()
  - Insert()
  - FindMin()
  - FindMax()
  - Delete(), and
  - Find()

Binary Search Tree

- Our data structure shall be a binary tree with a few modifications.
- Assume that the data is integer valued for now.
- Search Invariant:

  The data at the root of any binary search tree is larger than all elements in the left subtree and is smaller than all elements in the right subtree.

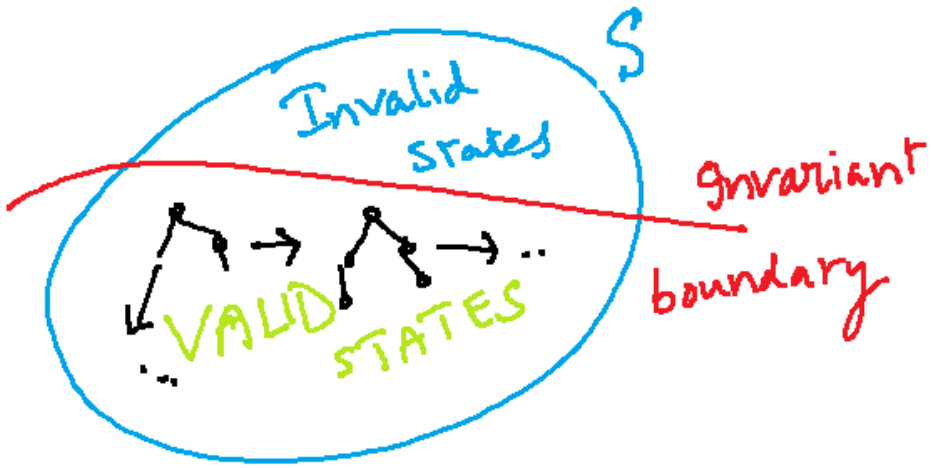
Data Structure Design via Invariants

- Our data structure shall be a binary tree with a few modifications.
- Assume that the data is integer valued for now.
- Search Invariant:

  The data at the root of any binary search tree is larger than all elements in the left subtree and is smaller than all elements in the right subtree.

Data Structure Design via Invariants

An ***invariant*** is just a set of conditions that will hold before and after every "step" of your program/algorithm.

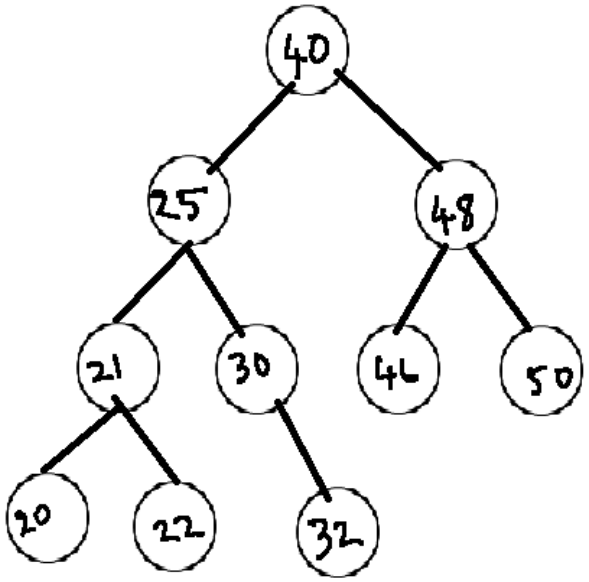


https://fgiesen.wordpress.com/2010/09/27/data-structures-and-invariants/

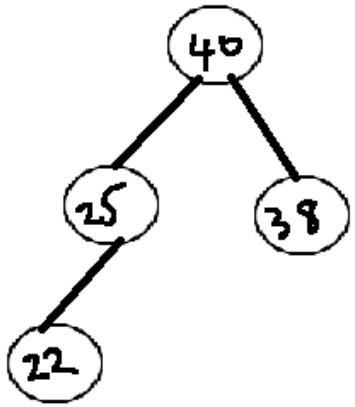
Binary Search Tree

- The search invariant has to be maintained at all times, after any operation.
- This invariant can be used to design efficient operations, and
- Also obtain bounds on the runtime of the operations.

Binary Search Tree



A binary search tree



Not a binary search tree

Operations

- Let us start with the operation Find(x).
- We are given a binary search tree T.
- Answer YES if x is in T, and answer NO otherwise.
- Throughout, let us call a node deficient, if it misses at least one child.
  - So a leaf node is also deficient.
  - So is an internal node with only one child.

Find(x)

- Let us compare x with the data at the root of T.
- There are three possibilities
  - x = T->data : Answer YES. Easy case.
  - x < T->data : Where can x be if it is in T? Left subtree
  - x > T->data : Where can x be if it is in T? Right subtree
- So, continue search in the left/right subtree.
- When to stop?

Find(x)

- Let us compare x with the data at the root of T.
- There are three possibilities
  - x = T->data : Answer YES. Easy case.
  - x < T->data : Where can x be if it is in T? Left subtree
  - x > T->data : Where can x be if it is in T? Right subtree
- So, continue search in the left/right subtree.
- When to stop?
  - Successful search stops when we find x.

Find(x)

- Let us compare x with the data at the root of T.
- There are three possibilities
  - x = T->data : Answer YES. Easy case.
  - x < T->data : Where can x be if it is in T? Left subtree
  - x > T->data : Where can x be if it is in T? Right subtree
- So, continue search in the left/right subtree.
- When to stop?
  - Successful search stops when we find x.
  - Unsuccessful search stops when we reach a deficient node without finding x.

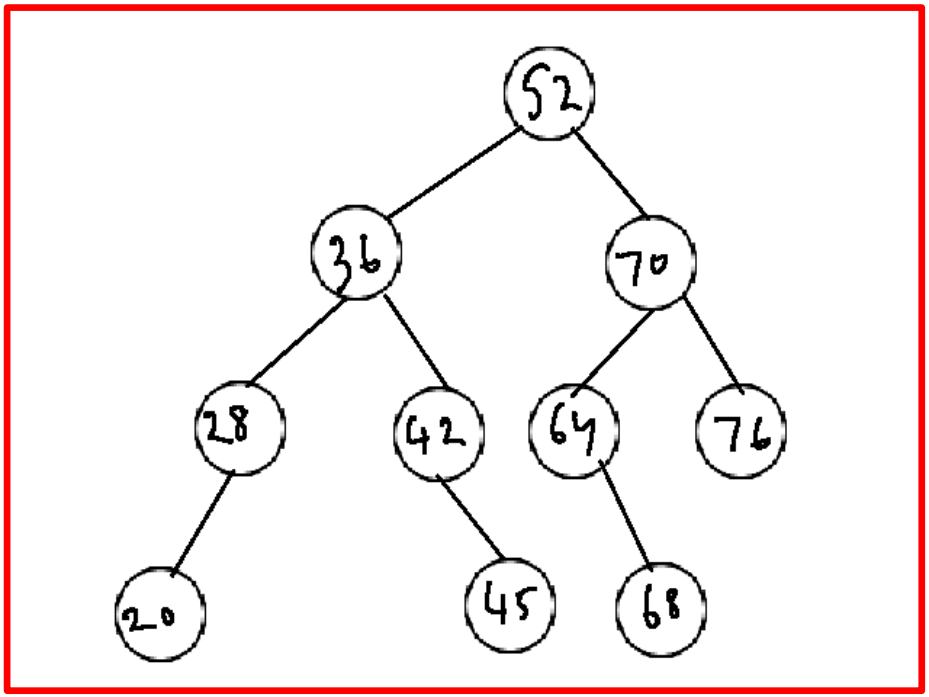
Find(x)

- Notice the similarity to binary search.
- In both cases, we continue search in a subset of the data.
  - In the case of binary search the subset size is exactly half the size of the current set.
  - Is that so in the case of a binary search tree also?
  - May not always be true.

Find(x)

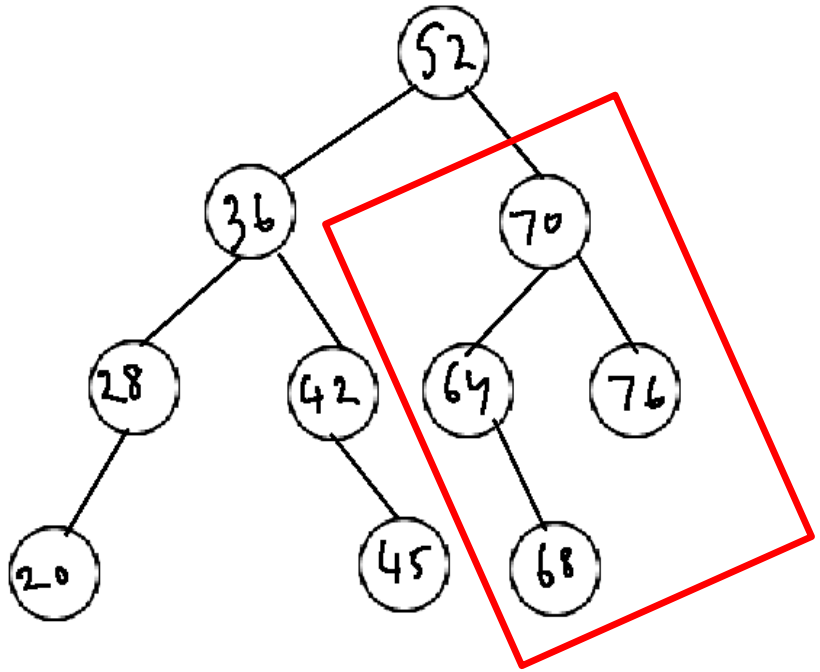
- How to analyze the runtime?
- Number of comparisons is a good metric.
- Notice that for a successful or an unsuccessful search, the worst case number of comparisons is equal to the height of the tree.
- What is the height of a binary search tree?
  - We'll postpone this question for now.

Find(x)



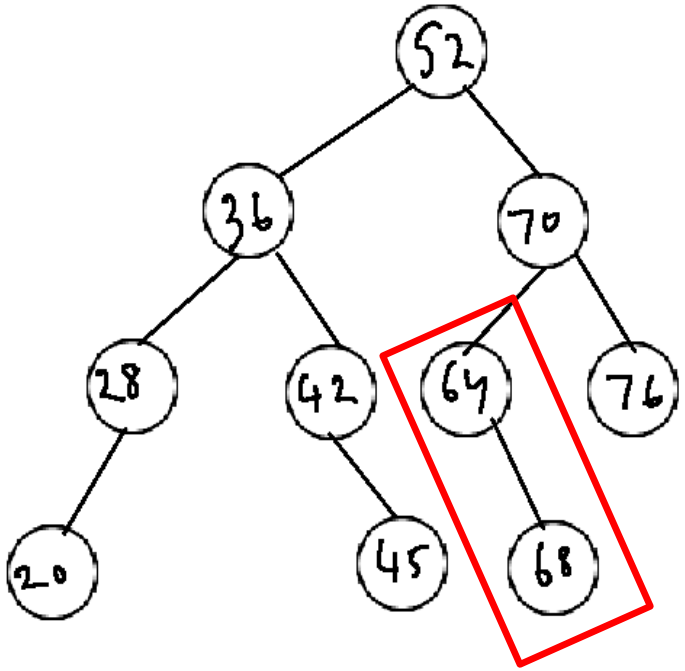
- Search for 68.
- Since 52 < 68, we search in the right subtree.

Find(x)



- Search for 68.
- Since 52 < 68, we search in the right subtree.
- Since 68 < 70, again search in the left subtree.

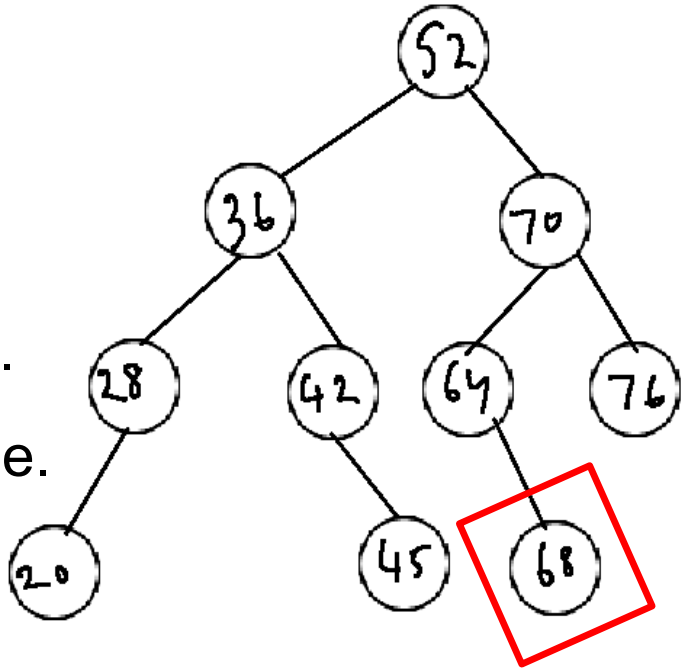
Find(x)



- Search for 68.
- Since 52 < 68, we search in the right subtree.
- Since 68 < 70, again search in the left subtree.
- Since 68 > 64, again search in the right subtree.

Find(x)

- Search for 68.
- Since 52 < 68, we search in the right subtree.
- Since 68 < 70, again search in the left subtree.
- Since 68 > 64, again search in the right subtree.
- Finally, find 68 as a leaf node.
- Now, try Find(48)



Find(x) Pseudocode

```
procedure Find(x, T)
begin
        if T == NULL return NO;
        if T->data == x return YES;
        else if T->data < x
                return Find(x, T->right);
        else
                return Find(x, T->left);
end
```

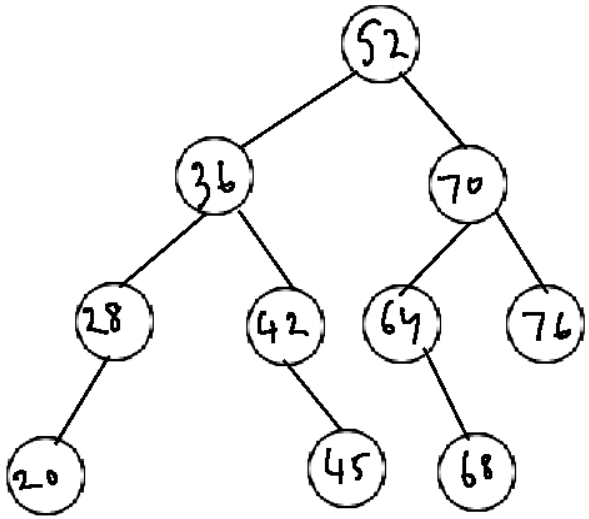
Observation on Find(x)

- Travel along only one path of the tree starting from the root.
- Hence, important to minimize the length of the longest path.
  - This is the depth/height of the tree.

Operation FindMin and FindMax

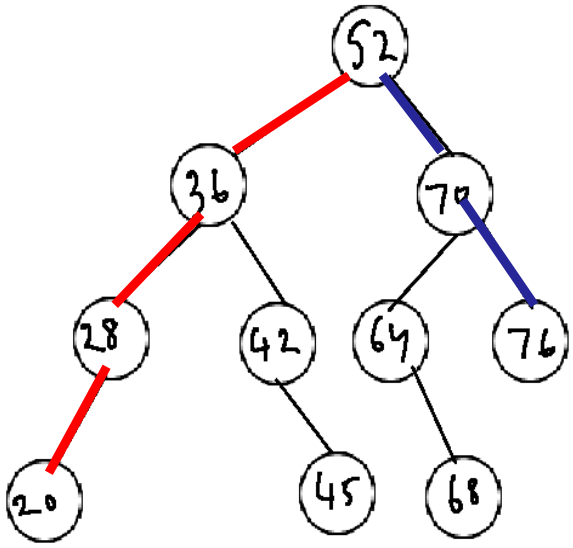
- Consider FindMin.
- Where is the smallest element in a binary search tree?
- Recall that values in the left subtree are smaller than the root, at every node.
- So, we should travel leftward.
  - stop when we reach a leaf or
  - a node with no left child.
  - Essentially, a deficient node missing a left child.
- FindMax is similar. How should we travel?

Operation FindMin and FindMax



- On the above tree, findMin will travese the path shown in red.
- FindMax will travel the path shown in blue.

Operation FindMin and FindMax



- On the above tree, findMin will travese the path shown in red.
- FindMax will travel the path shown in blue.

Operation FindMin and FindMax

```
procedure FindMin(T)
begin
        if T = NULL return null;
        if T–> left = NULL return T;
        return FindMin(T–>left);
end
```

- Both these operations also traverse one path of the tree.
- Hence, the time taken is proportional to the depth of the tree.
- Notice how the depth of the tree is important to these operations also.

Insert(x)

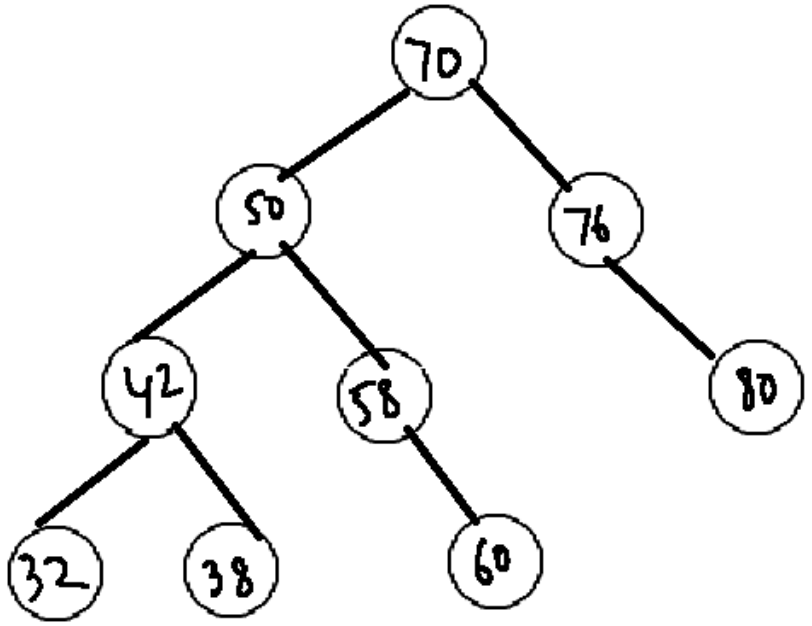
- Where should x be inserted?
- Should satisfy the search invariant.
  - So, if x is larger than the root, insert in the right subtree
  - if x is smaller than the root, insert in the left subtree.
- Repeat the above till we reach a deficient node.
- Can always add a new child to a deficient node.
- So, add node with value x as a child of some deficient node.

Insert(x)

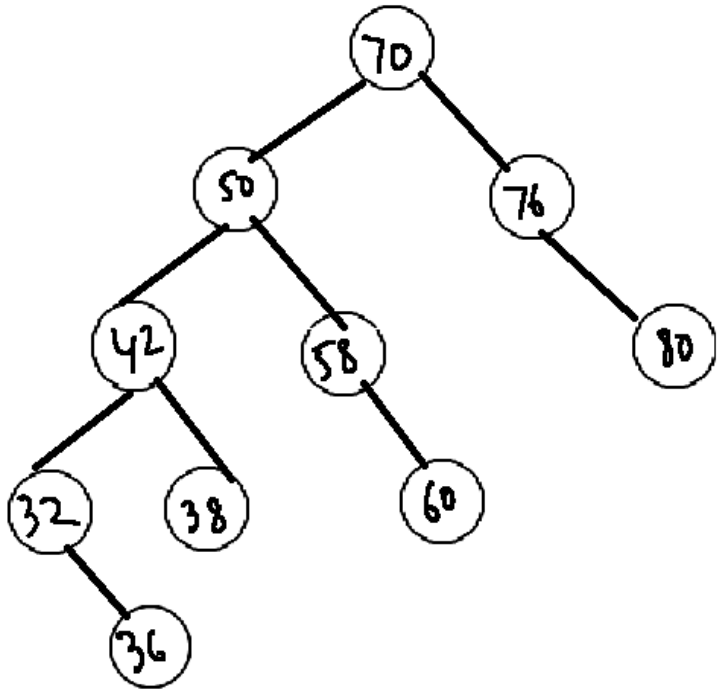
- Notice the analogy to Find(x)
- If x is not in the tree, Find(x) stops at a deficient node.
- Now, we are inserting x as a child of the deficient node last visited by Find(x).
- If the tree is presently empty, then x will be the new root.
- Let us consider a few examples.

Insert(x)

- Consider the tree shown and inserting 36.
- We travel the path 70 – 50 – 42 – 32.
- Since 32 is a leaf node, we stop at 32.



Insert(x)



- Now, 36 > 32. So 36 is inserted as a right child of 32.

Insert(x)

- Show the binary search tree obtained after inserting the following values in that order starting with an empty binary search tree.

  32, 28, 22, 38, 42, 51, 18, 37, 12

- Delete 32 from the resulting tree by removing the smallest node in the right subtree of 32.

Insert(x)

```
Procedure insert(x)
begin
T’ = T;
if T’ =  NULL then
        T’ = new Node(x, Null, Null);
else
        while (1)
                if T’–> data > x then
                                If T'->left then T’ = T’–> left;
                                else Add x as a left child of T'
                                break;
                else
                                If T'->right then T’ = T’–> right;
                                else Add x as a right child of T'
                                 break;
        end-while;
End.
```

Insert(x)

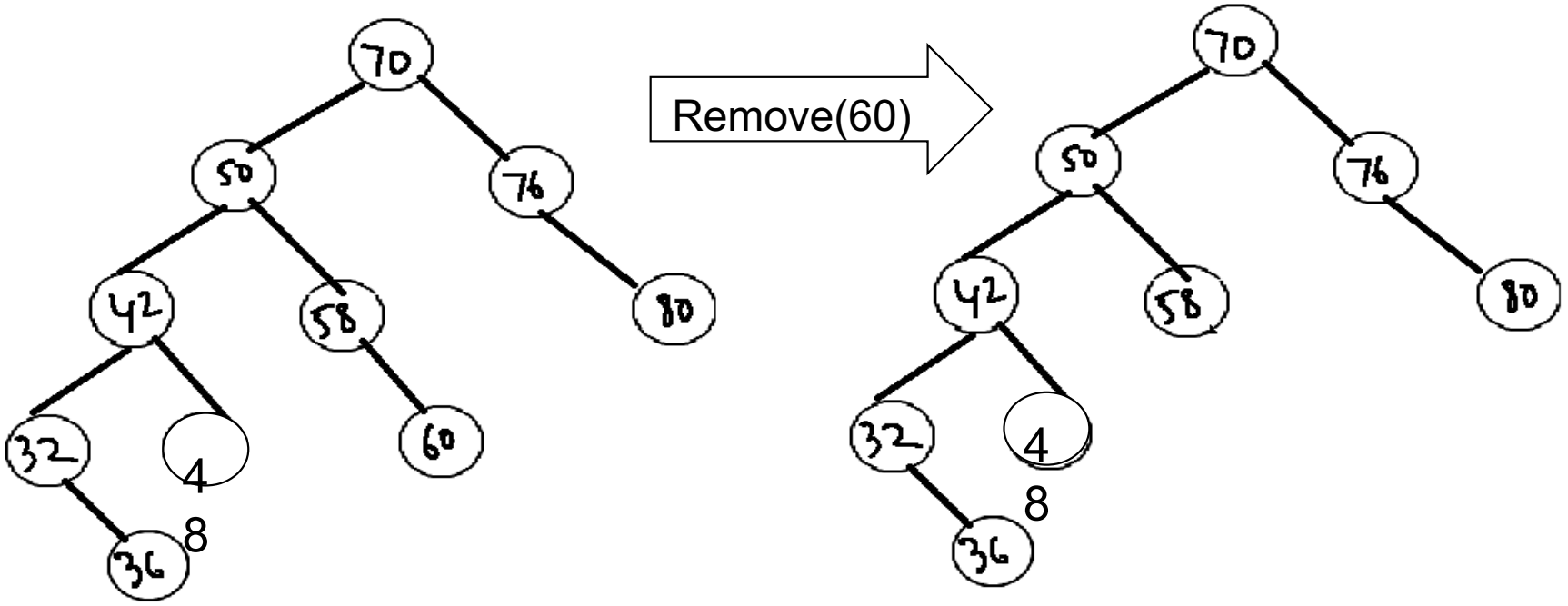
- New node always inserted as a leaf.
- To analyze the operation insert(x), consider the following.
  - Operation similar to an unsuccessful find operation.
  - After that, only O(1) operations to add x as a child.
- So, the time taken for insert is also proportional to the depth of the tree.

Remove(x)

- Finally, the remove operation.
- Difficult compared to insert
  - new node inserted always as a leaf.
  - but can also delete a non-leaf node.
- We will consider several cases
  - when x is a leaf node
  - when x has only one child
  - when x has both children

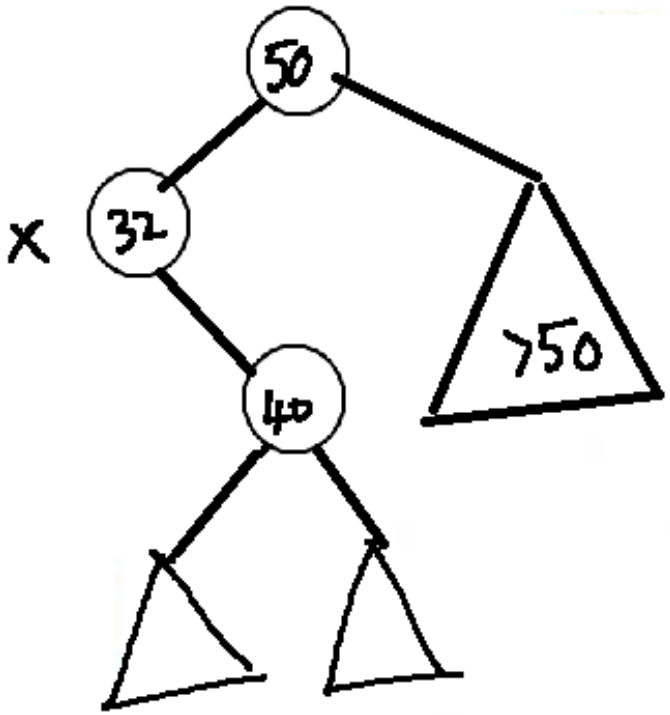
Remove(x)

- If x is a leaf node, then x can be removed easily.
  - parent(x) misses a child.

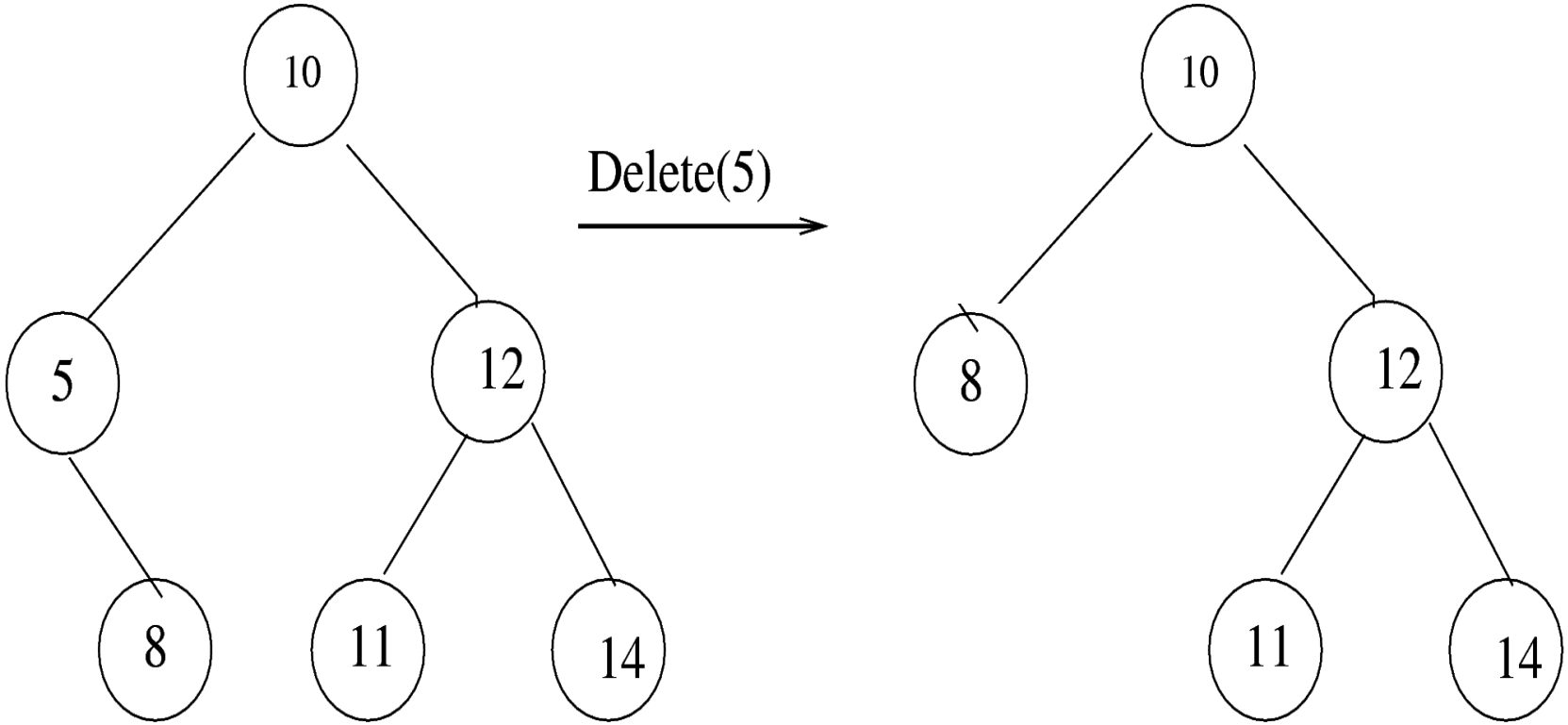


Remove(x)

- Suppose x has only one child, say right child.
- Say, x is a left child of its parent.
- Notice that x < parent(x) and child(x) > x, and also child(x) < parent(x).
- So, child(x) can be a left child of parent(x), instead of x.
- In essence, promote child(x) as a child of parent(x).



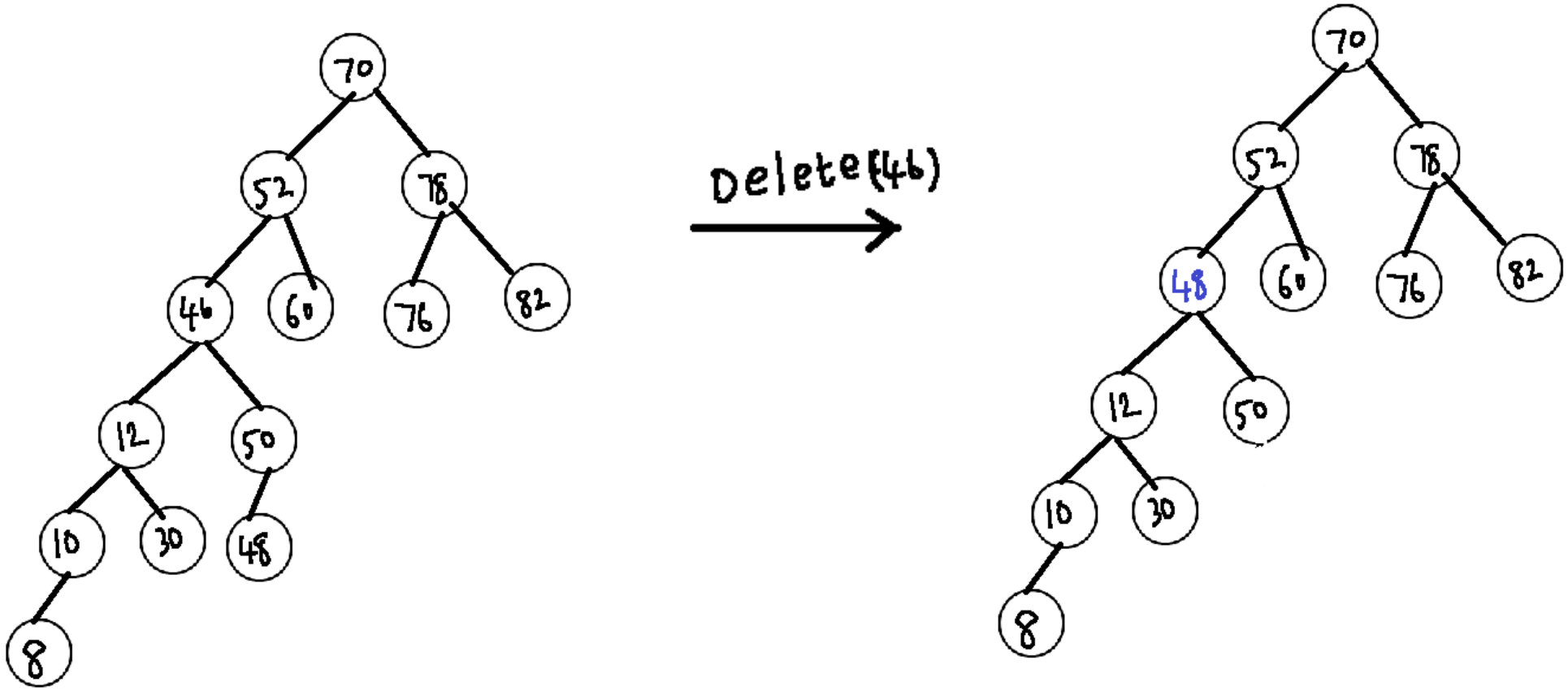
Remove(x)



Remove(x)

- One possibility is to consider the maximum valued node in the left subtree of x.
- Equivalently, can also consider the node with the minimum value in the right subtree of x.
- Notice that both these replacement nodes are deficient nodes. Hence easy to remove them.
- In a way, to remove x, we physically remove a deficient node.

Remove(x)


70
52
78
46
60
76
82
12
50
10
30
48
8
Delete(46)
70
52
78
48
60
76
82
12
50
10
30
8

Remove(x)

- From the tree shown above, delete nodes 30, 78, and 12 in that order.

Remove(x)

```
Procedure Delete(x, T)
begin
if T = NULL then return NULL;
T' = Find(x);
if T' has only one child then
        adjust the parent of the  remaining child;
else
        T'' = FindMin(T'–> right);
        Remove T'' from the tree;
        T'–> value = T''–> value;
End-if
End.
```

Remove(x)

- Time taken by the remove() operation also proportional to the depth of the tree.

Thank You