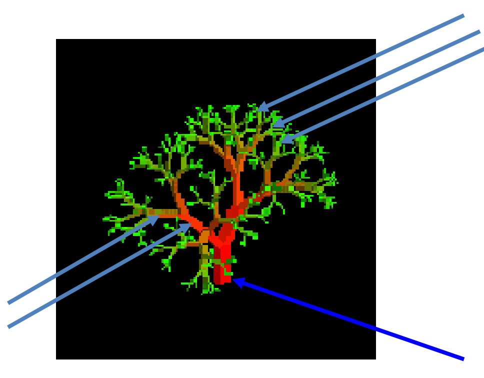
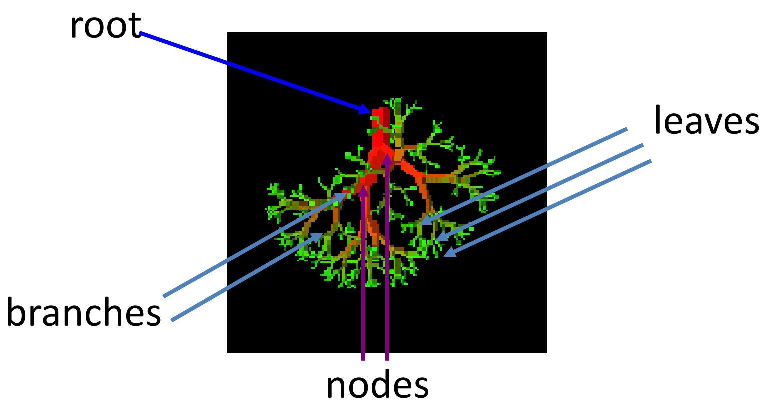
Trees and
Binary Trees

Nature View of a Tree

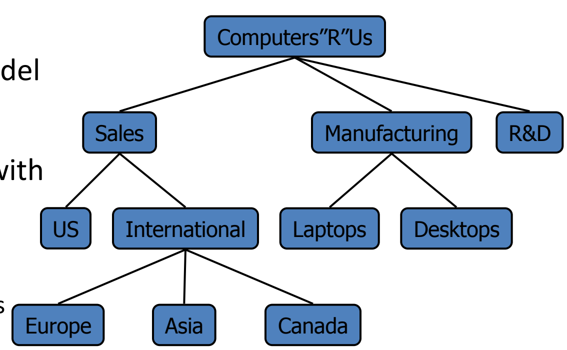


Computer Scientist’s View



What is a Tree

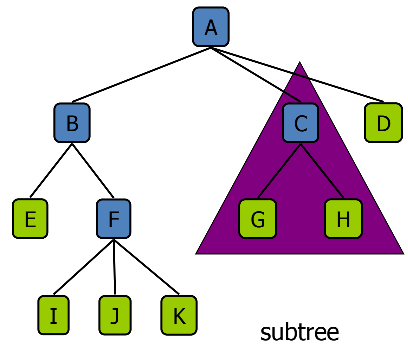
- A tree is a finite nonempty set of elements.
- It is an abstract model of a hierarchical structure.
- Consists of nodes with a parent-child relation.
- Applications:
  - Organization charts
  - File systems
  - Programming environments



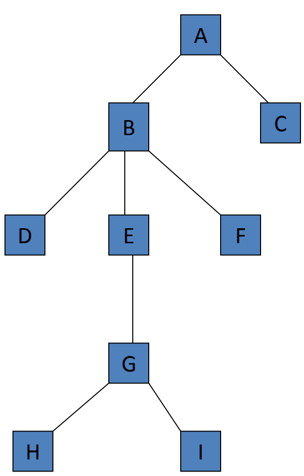
Tree Terminology

- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node (3)
- **Degree** of a node: the number of its children
- **Degree** of a tree: the maximum degree of its node.

- **Subtree**: tree consisting of a node and its descendants

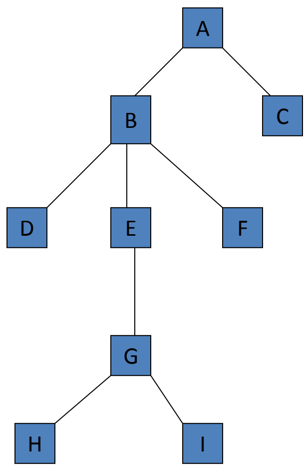


Tree Properties



Property	Value
Number of nodes	
Height	
Root Node	
Leaves	
Interior nodes	
Ancestors of H	
Descendants of B	
Siblings of E	
Right subtree of A	
Degree of this tree	

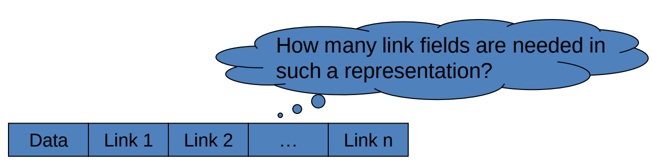
Tree Properties



Property	Value
Number of nodes :	9
Height :	4
Root Node :	A
Leaves :	5
Interior nodes :	4
Ancestors of H :	G, E,B, A
Descendants of B :	E, G, H, I
Siblings of E :	D, F
Right subtree of A :	C
Degree of this tree :	3

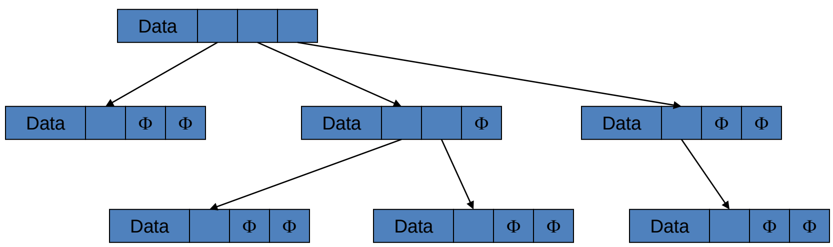
Intuitive Representation of Tree Node

- List Representation
  - (A (B (E (K, L), F), C (G), D (H (M), I, J)))
  - The root comes first, followed by a list of links to sub-trees



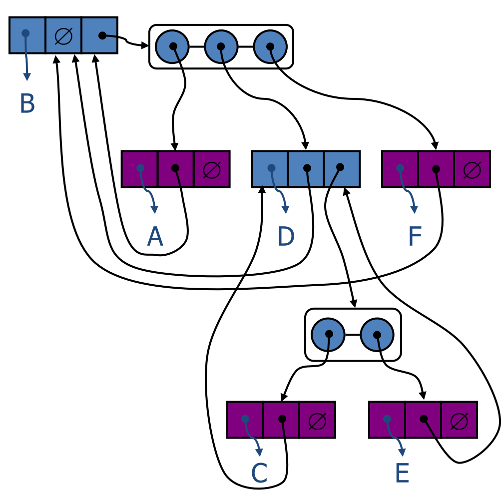
Trees

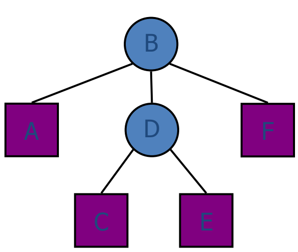
- Every tree node:
  - object – useful information
  - children – pointers to its children



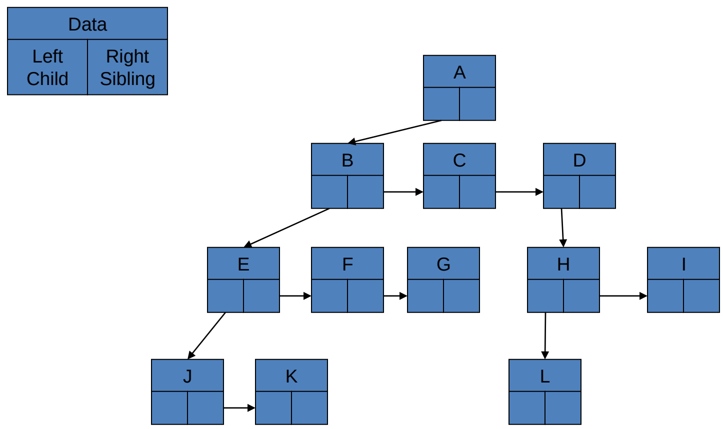
A Tree Representation

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes





Left Child, Right Sibling Representation



11

TREE ADT: Operations

- Operations
  - Traversal
  - Insertion
  - Deletion
  - Search
  - Copy
  - ….

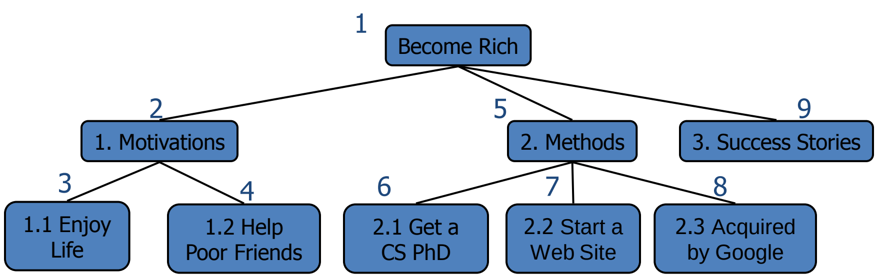
12

Tree Traversal

- Two main methods:
  - Preorder
  - Postorder
- Recursive definition
- Preorder:
  - visit the root
  - traverse in preorder the children (subtrees)
- Postorder
  - traverse in postorder the children (subtrees)
  - visit the root

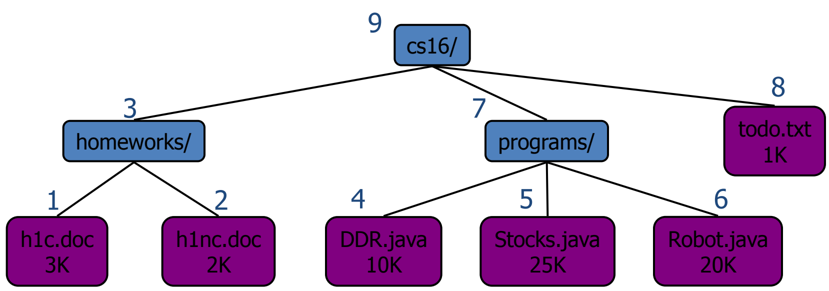
Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories



15

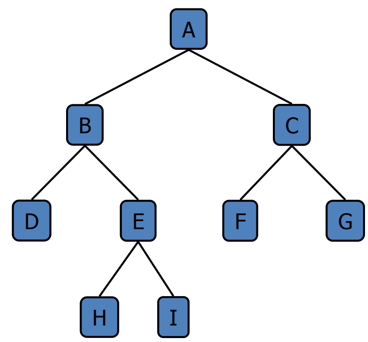
Binary Trees

16

Binary Tree

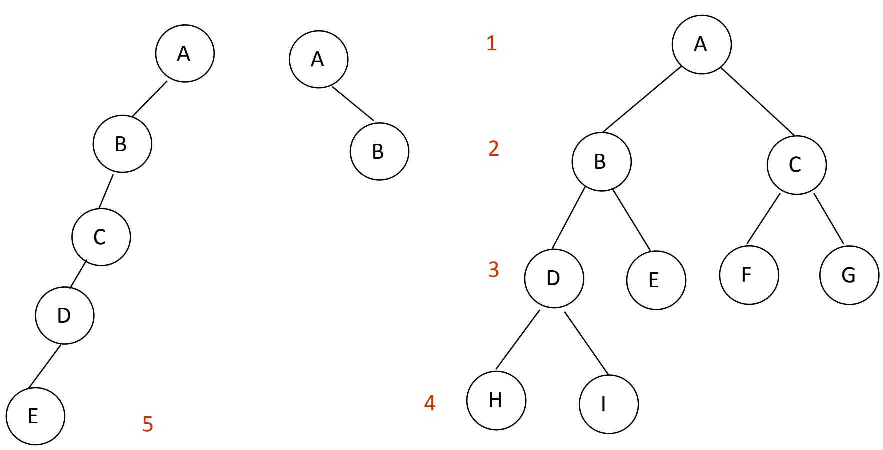
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (degree of two)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, OR
  - a tree whose root has an ordered pair of children, each of which is a binary tree

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



17

Examples of the Binary Tree



18

Difference Between A Tree and A Binary Tree

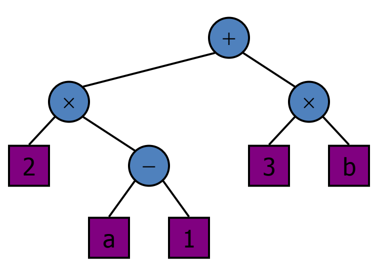
- The subtrees of a binary tree are ordered; those of a tree are not ordered.

- Are different when viewed as binary trees.
- Are the same when viewed as trees.

19

Arithmetic Expression Tree

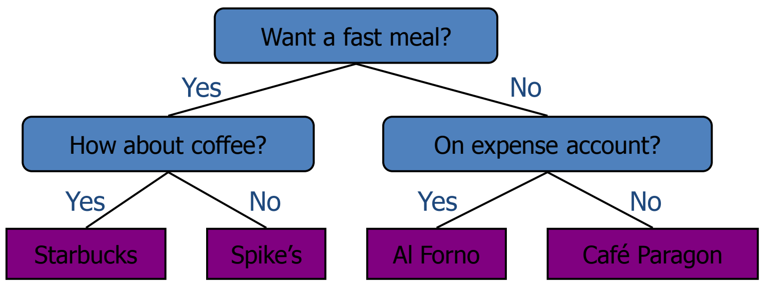
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



20

Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision

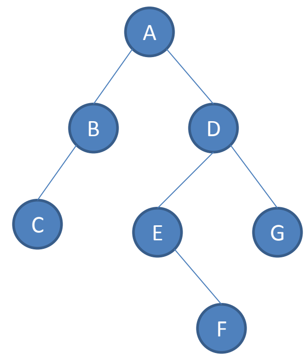


Binary Tree Traversal

- Traversal
  - Each node in a tree is processed exactly once in a systematic manner
- Three main ways of tree traversal
  - Preorder
  - Inorder
  - Postorder

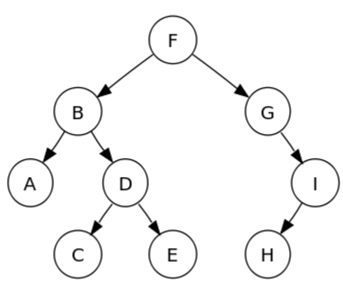
Binary Tree Traversal…

- The easiest way to define each order is by using recursion
- Preorder traversal (Rlr)
  - Process the root node
  - Traverse the left subtree in preorder
  - Traverse the right subtree in preorder
- Preorder traversal: ABCDEFG



23

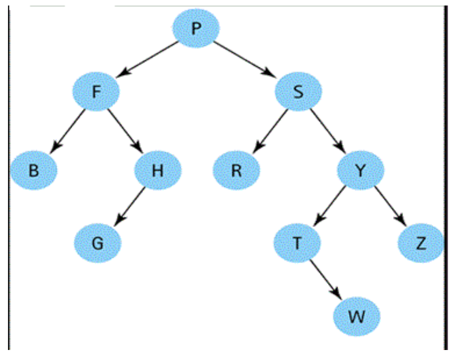
Assignment: Preorder Traversal



Preorder: FBADCEGIH

24

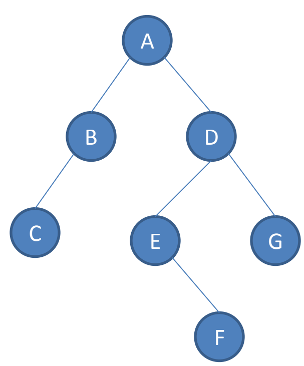
Assignment: Preorder Traversal



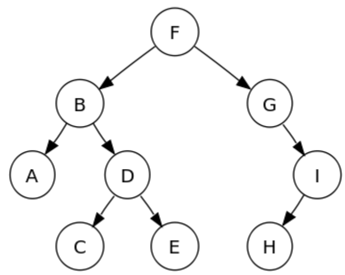
Preorder: PFBHGSRYTWZ

Binary Tree Traversal…

- Inorder traversal (lRr)
  - Traverse the left subtree in Inorder
  - Process the root node
  - Traverse the right subtree in Inorder
- Inorder traversal: CBAEFDG

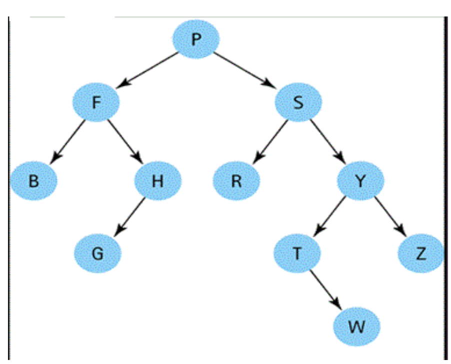


Assignment: Inorder Traversal



Inorder: ABCDEFGHI

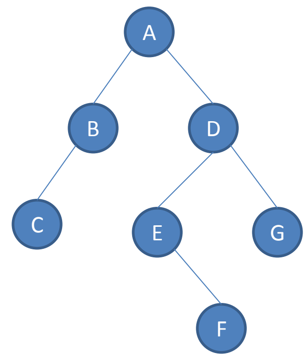
Assignment: Inorder Traversal



Inorder: BFGHPRSTWYZ

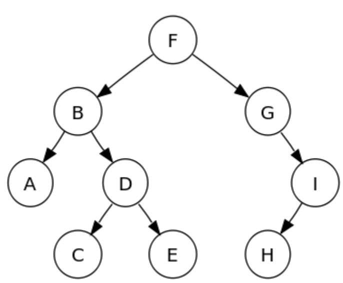
Binary Tree Traversal…

- Postorder traversal (lrR)
  - Traverse the left subtree in postorder
  - Traverse the right subtree in postorder
  - Process the root node
- Postorder traversal: CBFEGDA



29

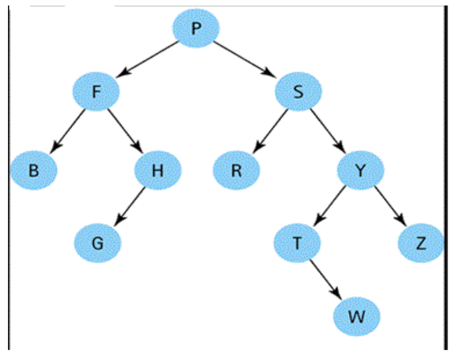
Assignment: Postorder Traversal



Postorder: ACEDBHIGF

30

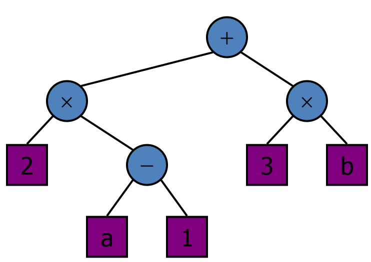
Assignment: Postorder Traversal



Postorder: BGHFRWTZYSP

Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print “(“ before traversing left subtree
  - print “)“ after traversing right subtree



$((2 \times (a - 1)) + (3 \times b))$

Tutorial: Algorithm to Print Arithmetic Expression using Binary Tree

33

Linked storage representation for binary tree

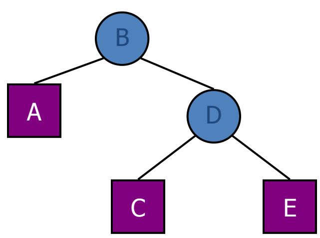
LPTR	DATA	RPTR

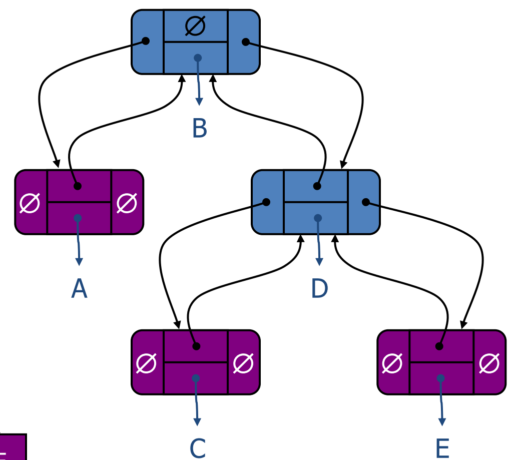
- DATA
  - Information associated with node
- LPTR
  - Address of location of left subtree
- RPTR
  - Address of location of right subtree

34

Linked storage representation for binary tree: option 2

- A node is represented by an object storing
  - Data
  - Parent
  - Left child
  - Right child





Binary tree traversal: algorithms

- Recursive
- Iterative

Preorder traversal: iterative algorithm

Algorithm PREORDER(T)
Input: A binary tree whose root node address is given by a pointer variable T
Output: Preorder traversal of a tree
[Initialize]
if T=NULL **then**
    return false {Empty Tree}
else
    TOP← -1
    push(S, T)
[Process each stacked branch address]
Repeat while TOP>=0
[Get stored address and branch left]
    P← pop(S)
    Repeat while P != NULL
        write DATA(P)
        if RPTR(P) != NULL **then**
            push(S,RPTR(P)) *[Store address of nonempty right subtree]*
            P← LPTR(P) *[branch left]*
[Finished]
return

Preorder traversal: recursive algorithm

Algorithm RPREORDER(T)
Input: A binary tree whose root node address is given by a pointer variable T
Output: Preorder traversal of a tree
[Process the root node]
if T != NULL **then**
    write DATA(T)
else
    return false {Empty Tree}
[Process the left subtree]
if LPTR(T) != NULL **then**
RPREORDER(LPTR(T))
[Process the right subtree]
if RPTR(T) != NULL **then**
RPREORDER(RPTR(T))
[Finished]
return

Postorder traversal: iterative algorithm

Algorithm POSTORDER(T)
Input: A binary tree whose root node address is given by a pointer variable T
Output: Postorder traversal of a tree
[Initialize]
if T=NULL **then**
    return false {Empty Tree}
else
    TOP← -1
    P←T
[Traverse in postorder]
Repeat while true
[Descend left]
    Repeat while P != NULL
        push(S,P)
        P← LPTR(P)
[Process a node whose left and right subtree have been traversed]
    Repeat while S[TOP] <0
        P← POP(S)
        write DATA(P)
        if TOP = -1 **then**
        return
[Branch right and then mark node from which we branched]
    P← RPTR(S[TOP])
    S[TOP] ← -S[TOP]
[Finished]
return

Postorder traversal: recursive algorithm

Algorithm RPOSTORDER(T)
Input: A binary tree whose root node address is given by a pointer variable T
Output: Postorder traversal of a tree
[Initialize]
if T=NULL **then**
    return false {Empty Tree}
[Traverse in postorder the left subtree]
if LPTR(T) != NULL **then**
    RPOSTORDER(LPTR(P))
[Traverse in postorder the left subtree]
if RPTR(T) != NULL **then**
    RPOSTORDER(RPTR(P))
[Process the root node]
write DATA(P)
[Finished]
return

Inorder traversal: iterative algorithm

Algorithm INORDER(T)
Input: A binary tree whose root node address is given by a pointer variable T
Output: Inorder traversal of a tree

Inorder traversal: recursive algorithm

Algorithm RINORDER(T)
Input: A binary tree whose root node address is given by a pointer variable T
Output: Inorder traversal of a tree

43

Making a duplicate copy of a tree

Algorithm COPY(T)
Input: A binary tree whose root node address is given by a pointer variable T
Output: address of the root node of a new tree which is the copy of the tree with root T

```
[Null pointer?]
if T = NULL then
        return false {Empty tree}
[Create a new node]
        NEW ← NODE
[Copy information field]
        DATA(NEW)← DATA(T)
[Set structural links]
        LPTR(NEW) ← COPY(LPTR(T))
        RPTR(NEW) ← COPY(RPTR(T))
[Return address of a new node]
return NEW
```

44

Delete a node from Binary tree

- If a node to be deleted has no offspring, it can simply be deleted
- If a node has either a right or left empty subtree, the nonempty subtree can be appended to its grandparent node
- If the node has both a right and left subtree, the deletion strategy…

45

Delete a node from Binary tree…

- If the node has both a right and left subtree…
  - First obtain the inorder successor of the node to be deleted
  - Then the right subtree of this successor node is appended to its grandparent node
  - Then the node to be deleted is replaced by its inorder successor

46

```
Algorithm DELETE(HEAD,X)
Input: A binary tree whose root node address is given by a pointer variable HEAD, X is the
data value of the node marked for deletion
Output: address of the root node of a new tree in which node with data value X is deleted
[Initialize]
if LPTR(HEAD) != HEAD then
        CUR ← LPTR(HEAD)
        PARENT ← HEAD
        D = 'L'
else
        return false {node not found}
[Search for the node marked for deletion]
FOUND← false
Repeat while not FOUND and CUR != NULL
        if DATA(CUR)=X then
                FOUND=true
        else
                if X < DATA(CUR) then
                        PARENT← CUR
                        CUR ← LPTR(CUR)
                        D = 'L'
                else
                        PARENT← CUR
                        CUR← RPTR(CUR)
                        D= 'R'
```

```
if FOUND = false then
        return false {node not found}
[perform the indicated deletion and restructure the tree]
if LPTR(CUR) = NULL then
        Q← RPTR(CUR)
else
        if RPTR(CUR) = NULL then
                Q← LPTR(CUR)
        else   [check the right child of successor]
                SUC ← RPTR(CUR)
                if LPTR(SUC) = NULL then
                        LPTR(SUC) ← LPTR(CUR)
                        Q ← SUC
                else    [search for the successor of CUR]
                        PRED ← RPTR(CUR)
                        SUC ← LPTR(PRED)
                        Repeat while LPTR(SUC) != NULL
                                PRED ← SUC
                                SUC ← LPTR(PRED)
                        [connect successor]
                        LPTR(PRED) ← RPTR(SUC)
                        LPTR(SUC)← LPTR(CUR)
                        RPTR(SUC)← RPTR(CUR)
                        Q← SUC
```

Delete a node from Binary tree…

```
[connect parent of X to its replacement]
if D = 'L' then
        LPTR(PARENT)  ← Q
else
        RPTR(PARENT) ← Q
```

49

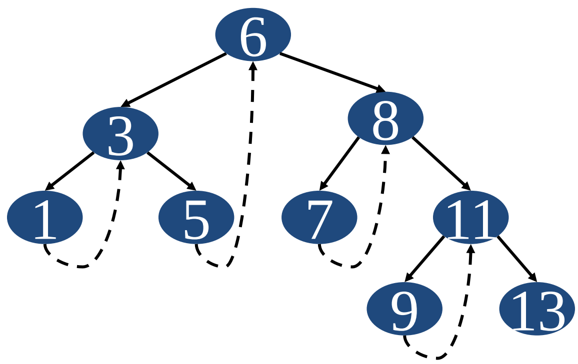
Threaded Storage Representation for Binary Trees

- Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers
- We can use these pointers to help us in inorder traversals
- We have the pointers reference the next node in an inorder traversal; called threads
- We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer

Threaded tree node structure

```
struct node
{
    struct node *lptr, *rptr;
    bool lthread, rthread;
};
```

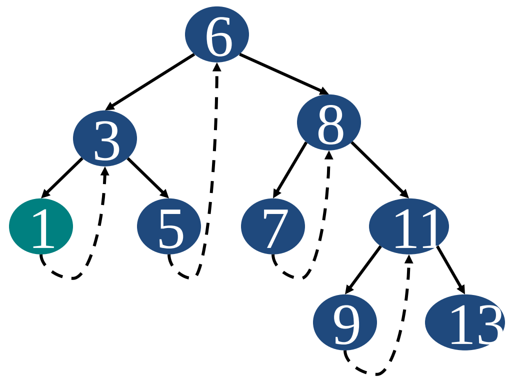
Threaded Tree Example



Threaded Tree Traversal

- We start at the leftmost node in the tree, print it, and follow its right thread
- If we follow a thread to the right, we output the node and continue to its right
- If we follow a link to the right, we go to the leftmost node, print it, and continue

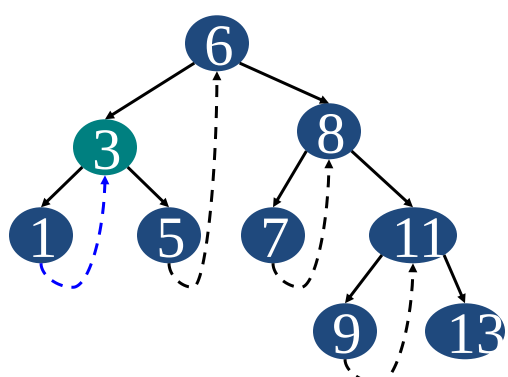
Threaded Tree Traversal



Output
1

Start at leftmost node, print it

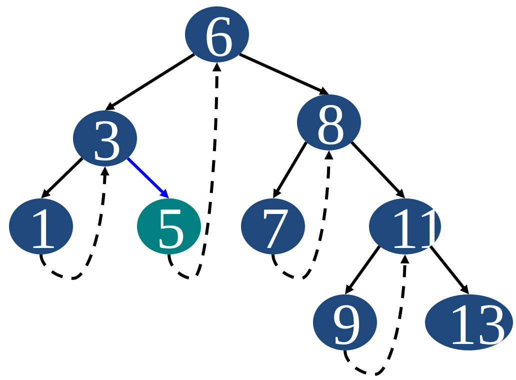
Threaded Tree Traversal



Output
1
3

Follow thread to right, print node

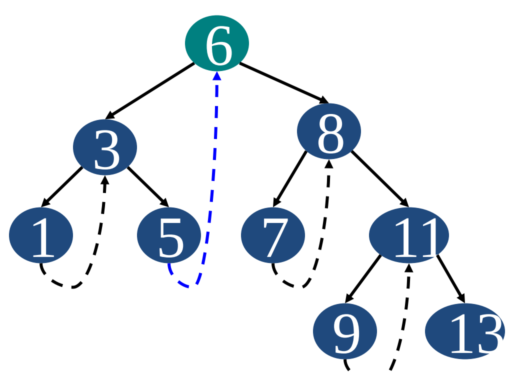
Threaded Tree Traversal



Output
1
3
5

Follow link to right, go to leftmost node and print

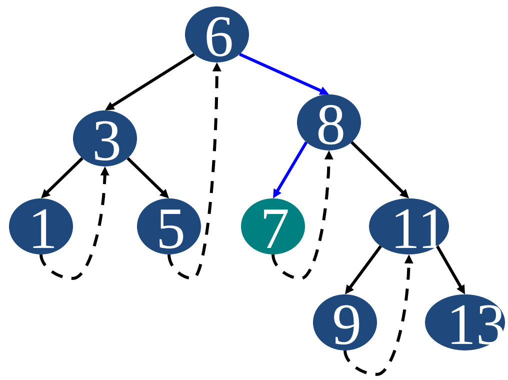
Threaded Tree Traversal



Output
1
3
5
6

Follow thread to right, print node

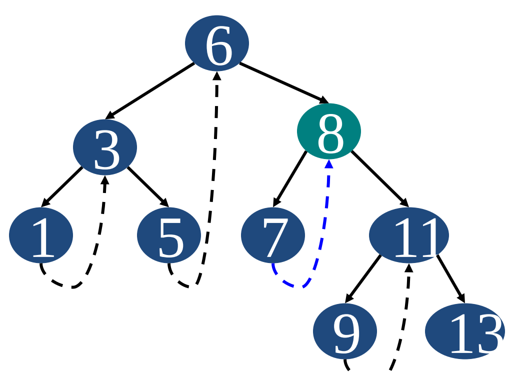
Threaded Tree Traversal



Output
1
3
5
6
7

Follow link to right, go to leftmost node and print

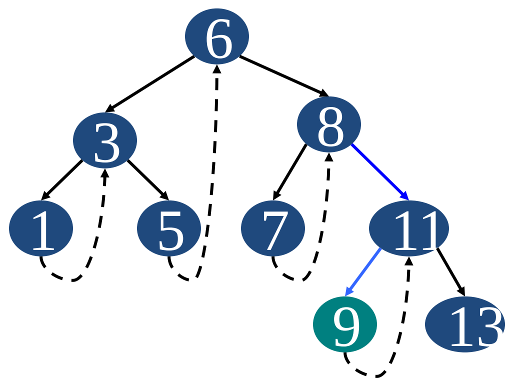
Threaded Tree Traversal



Output
1
3
5
6
7
8

Follow thread to right, print node

Threaded Tree Traversal

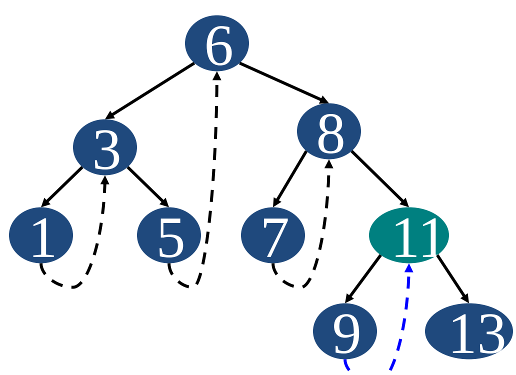


Output
1
3
5
6
7
8
9

Follow link to right, go to leftmost node and print

60

Threaded Tree Traversal

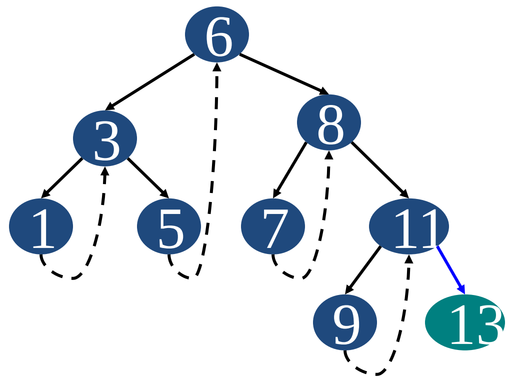


Output
1
3
5
6
7
8
9
11

Follow thread to right, print node

61

Threaded Tree Traversal



Output
1
3
5
6
7
8
9
11
13

Follow link to right, go to leftmost node and print

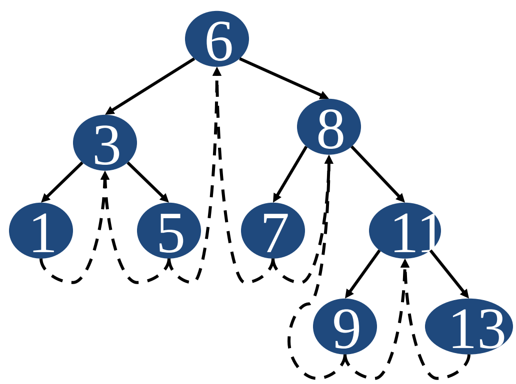
62

Threaded Tree modification

- We're still wasting pointers, since half of our leafs' pointers are still null
- We can add threads to the previous node in an inorder traversal as well, which we can use to traverse the tree backwards or even to do postorder traversals

63

Threaded Tree Modification



Find Inorder Successor

Algorithm INS(X)
Input: X, the address of a node in a threaded binary tree
Output: address of the inorder successor of X
[Return the right pointer of the given node if a thread]
P ← RPTR(X)
if RTHREAD(X) = true
then return P
[Branch left repeatedly until a left thread]
repeat while LTHREAD(P) != true
    P ← LPTR(P)
[Return address of successor]
return P

Find Inorder Predecessor

Algorithm INP(X)
Input: X, the address of a node in a threaded binary tree
Output: address of the inorder predecessor of X
[Return the left pointer of the given node if a thread]
P ← LPTR(X)
if LTHREAD = true
then **return** P
[Branch right repeatedly until a right thread]
repeat while RTHREAD(P) != false
    P ← RPTR(P)
[Return address of successor]
return P

66

Traverse Threaded Binary Tree in Inorder

Algorithm TINORDER(HEAD)
Input: address of a list head (HEAD) in a threaded binary tree, a subalgorithm INS previously discussed
Output: inorder traversal of a tree
[Initialize]
P ← HEAD
[Traverse threaded tree in inorder]
repeat while true
    P ← INS(P)
    if P=HEAD
    then **return**
    else Write(DATA(P))

67

Inserting a node into a Threaded Binary Tree

Algorithm LEFT(X, INFO)
Input: address of a designated node X in an inorder threaded binary tree and the information associated with the new node (INFO)
Output: Inserts a new node to the left of X
[Create a new node]
P ← NODE
DATA(P) ← INFO
[Adjust pointer fields]
    LPTR(P) ← LPTR(X)
    LPTR(X) ← P
    RPTR(P) ← -X
[Reset predecessor thread if required]
    if LPTR(P) > 0
    then RPTR(INP(P)) ← -P
    return

68

Threaded Binary Tree:Advantages

- Inorder traversal is somewhat faster than that of its unthreaded version
  - Because stack is not required in the former
- Efficient determination of the predecessor and successor nodes for any node P
- What are the disadvantages????

69

Conversion of General tree forest to Binary Tree

ALGORITHM CONVERT
[Given a forest of trees in preorder sequence, it is required to convert this forest into an equivalent binary tree with a list head (HEAD)].

1. [*Initialize*]

HEAD ← NODE
LPTR(HEAD) ← NULL
RPTR(HEAD) ← HEAD
LEVEL[1] ← 0
LOCATION TOP ← 1

2. [*Process the input*]

Repeat thru step 6 while input is there.

3. [*Input a node*]

Read(LEVEL,INFO).

70

Conversion of General tree forest to Binary Tree

4. [*Create a tree node*]
NEW <-- NODE
LPTR(NEW) <-- RPTR(NEW) <-- NULL
DATA(NEW) <-- INFO.

5. [*Compare levels*]

PRED_LEVEL ← LEVEL[TOP]
PRED_LOC ←LOCATION[TOP]
if LEVEL > PRED_LEVEL
then LPTR(PRED_LOC) ← NEW
else if LEVEL = PRED_LEVEL
RPTR(PRED_LOC) ← NEW
TOP ← TOP – 1
else
Repeat while LEVEL != PRED_LEVEL
TOP ← TOP – 1
PRED_LEVEL ← LEVEL[TOP]
PRED_LOC ← LOCATION[TOP]
if PRED_LEVEL ← LEVEL
then write (“Invalid Input”)
return

71

Conversion of General tree forest to Binary Tree

RPTR(PRED_LOC) <-- NEW
TOP <-- TOP – 1.

6. [*Pushing values in stack*]

TOP <-- TOP + 1
LEVEL[TOP] <-- LEVEL
LOCATION[TOP] <-- NEW.

7. [*FINISH*]
return.