Pontificia Universidad JAVERIANA Colombia

Estructuras de Datos

Tree Data Structure

In This Session

- **Tree Data Structure**
  - ▷ Terminology
  - ▷ Binary Trees
    - ◇ Properties
    - ◇ Array-based Representation
    - ◇ Linked Representation
    - ◇ Traversals
    - ◇ An Application

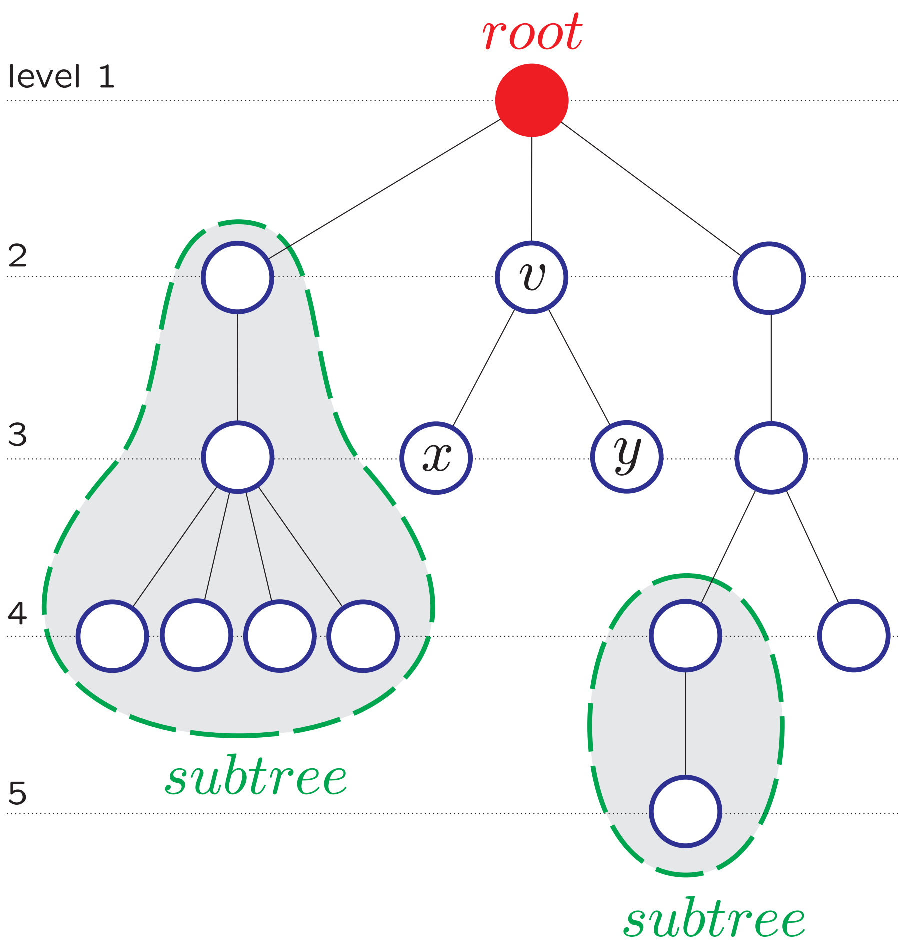
Tree Data Structure

- Until now: linear and tabular data
- How can we represent hierarchical data?
  - somebody's descendants
  - governmental/company subdivisions
  - modular decomposition of programs
- Answer: using a **Tree Data Structure**

A *tree* is a data structure that organizes information like an upsidedown tree

Terminology

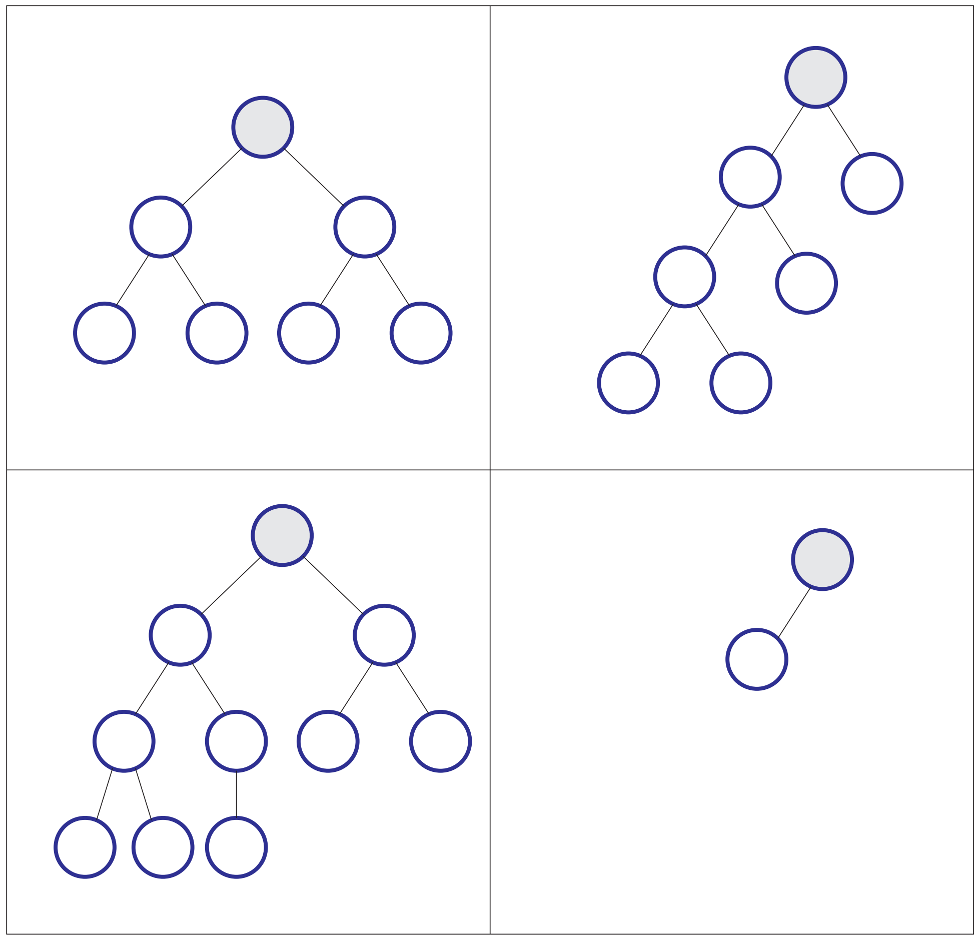
A **tree** is a finite nonempty set of elements



- x, y are **children** of v; v is a **parent** of x, y
- x, y are **siblings**
- Elements with no children are called **leaves**
- **Level:** root=level 1; children=level 2,3, ...
- **Degree of an element:** number of children
- **Height or Depth:** number of levels

Binary Trees

A **binary tree** is a tree (possible empty) in which every element has degree ≤ 2



Properties of Binary Trees

P1: *Every binary tree with n elements, $n > 0$, has exactly $n - 1$ edges.*

<u>Proof</u>: Each element (except the root) has one parent. $\exists$ exactly one edge between each child and its parent. Hence, $\exists n - 1$ edges. $\square$

P2: *The number of elements at level i is $\leq 2^{i-1}, i > 0$.*

<u>Proof:</u> By induction on i.
Basis: $i = 1$; number of elements $= 1 = 2^0$
Ind. Hypothesis: $i = k$; number of elements at level $k \leq 2^{k-1}$.
Look at level $i = k + 1$
(number of elements at level $k + 1$) $\leq 2 \cdot$(number of elements at level k)
$\leq 2 \times 2^{k-1} = 2^k$. $\square$

P3: *A binary tree of height h, $h > 0$, has at least h and at most $2^h - 1$ elements.*

Poof: Let n be the number of elements. $\exists$ must be ≥ 1 elements at each level, hence, $n \geq h$.
Now, if $h = 0$, then $n = 0 = 2^0 - 1$.
For $h > 0$, we have by P2 that

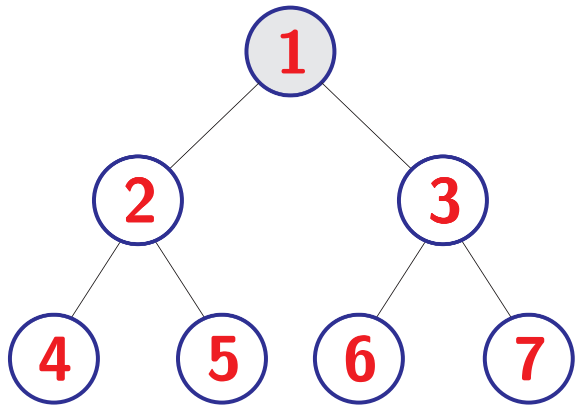
$$n \leq \sum_{i=1}^{h} 2^{i-1} = 2^h - 1$$

$\square$

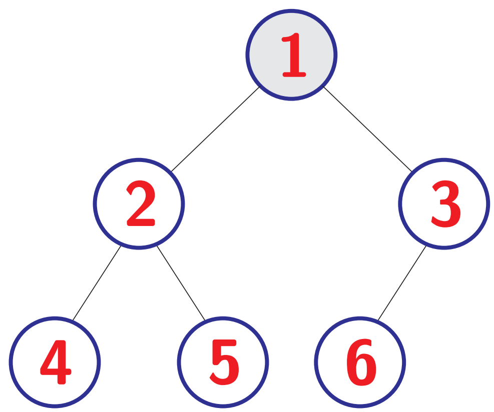
P4: *Let h be the height of an n-elements binary tree, $n \geq 0$. Then,* $\lceil \log_2(n+1) \rceil \leq h \leq n$

Proof: $\exists$ must be ≥ 1 element at each level, hence, $h \leq n$.
P3 $\Rightarrow n \leq 2^h - 1 \Rightarrow 2^h \geq n + 1 \Rightarrow h \geq \log_2(n+1)$.
Since h is an integer, we have that $h \geq \lceil \log_2(n+1) \rceil$. $\square$

Full binary tree: A binary tree of height h is *full* if contains exactly $2^h - 1$ elements.



Complete binary tree: Is a binary tree of height h in which all levels (except perhaps for the last) have a maximum number of elements.



Number the elements from 1 through $2^h - k$, starting from level 1 and proceed in a left-to-right fashion, for some $k \geq 1$.

P5: *Let i, $1 \leq i \leq n$, be the number assigned to an element v of a complete binary tree. Then:*

(i) *If $i = 1$, then v is the root. If $i > 1$, then the parent of v has been assigned the number $\lfloor i/2 \rfloor$.*

(ii) *If $2i > n$, then v has no left child. Otherwise, its left child has been assigned the number $2i$.*

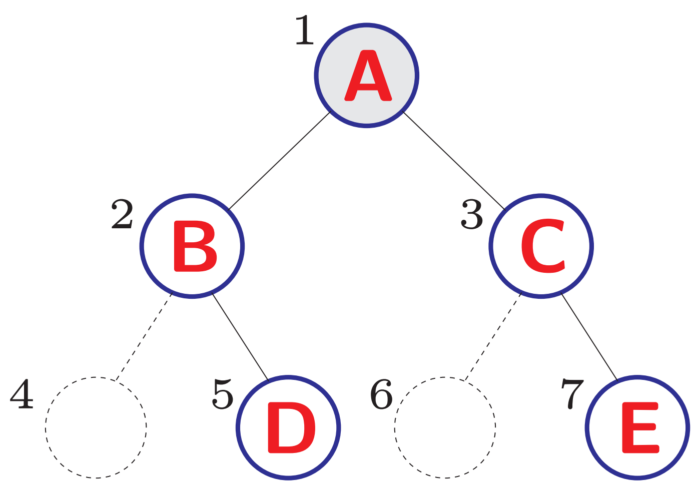
(iii) *If $2i + 1 > n$, then v has no right child. Otherwise, its right child has been assigned the number $2i + 1$.*

Proof: By induction on i. $\square$

Binary Tree Data Structure

– Array-based Representation –

Uses **P5**



1	2	3	4	5	6	7
A	**B**	**C**		**D**		**E**

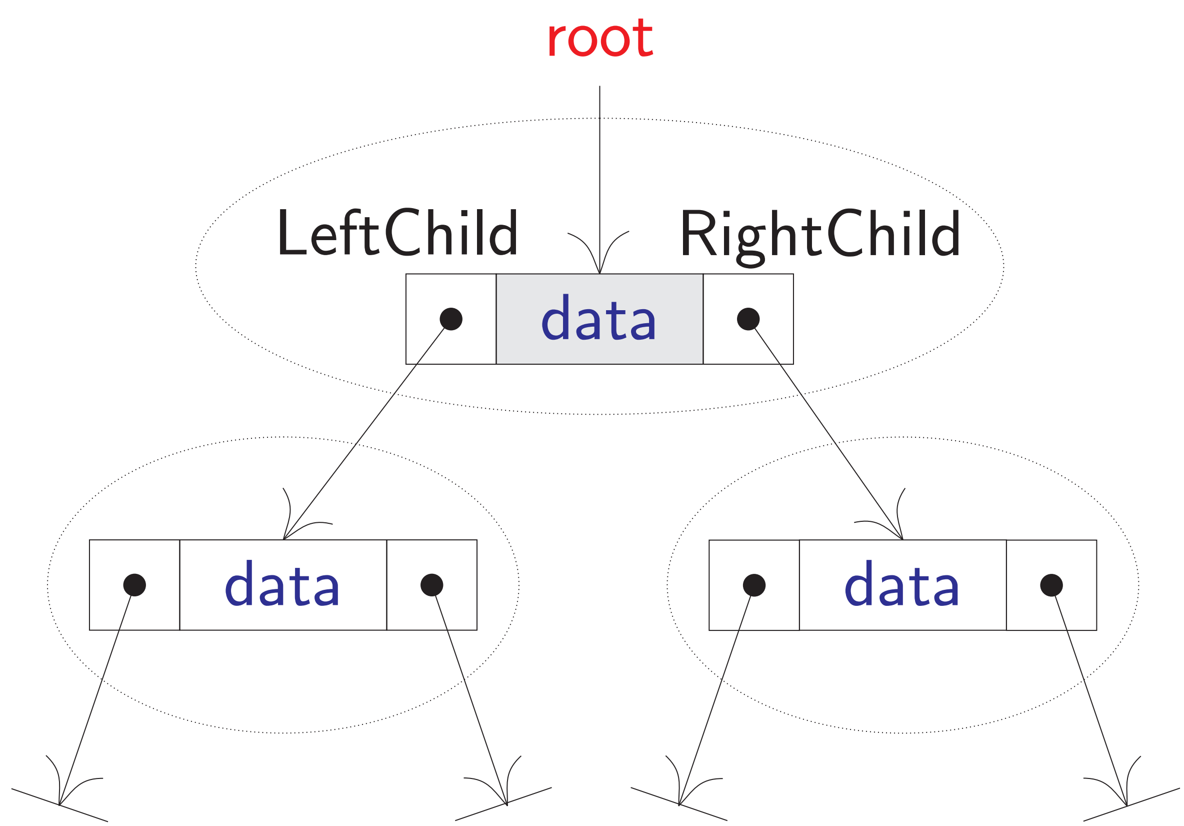
Note: An n–element binary tree may require an array of size $2^n - 1$ for its representation. $\Rightarrow$ Can be a waste of space.

Binary Tree Data Structure

– Linked Representation –

The most popular way to represent a binary tree is by using links or pointers. Each node is represented by three fields:

- data
- LeftChild
- RightChild

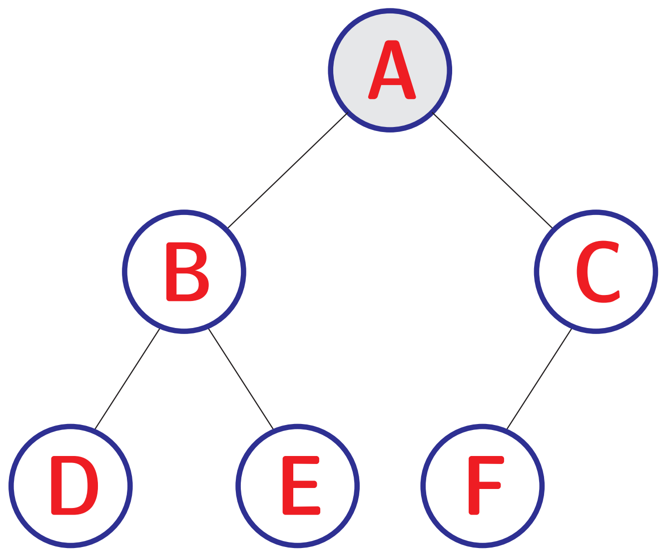


Binary Tree Traversal

There are four common ways to traverse a binary tree:

- **Preorder**: Visit-Left-Right (VLR)
- **Inorder**: Left-Visit-Right (LVR)
- **Postorder**: Left-Right-Visit (LRV)
- **Level order**

Example:



- Preorder: ABDECF
- Inorder: DBEAFC
- Postorder: DEBFCA
- Level order: ABCDEF

binarytree.h

```
#ifndef BINARYTREE
#define BINARYTREE

#include <stdio.h>
#include <stdlib.h>

typedef struct TreeNode TreeNode;
typedef struct Tree Tree;

struct TreeNode{
   int item;
   TreeNode * left;
   TreeNode * right;
};

struct Tree{
   TreeNode * root;
   void (*makeTree) ( Tree *, int, Tree *, Tree * );
   void (*preorder) ( TreeNode * );
   void (*inorder) ( TreeNode * );
   void (*postorder) ( TreeNode * );
   void (*levelorder) ( TreeNode * );
   int (*empty) ( TreeNode * );
   int (*size) ( TreeNode * );
   int (*height) ( TreeNode * );
};
```

```c
void makeTree( Tree * x, int e, Tree * l, Tree * r ){ ... }
void preorder( TreeNode * p ){ ... }
void inorder( TreeNode * p ){ ... }
void postorder( TreeNode * p ){ ... }
void givenLevel( TreeNode * p, int level ){ ... }
void levelorder( TreeNode * p ){ ... }
int empty( TreeNode * p ){ ... }
int size( TreeNode * p ){ ... }
int height( TreeNode * p ){ ... }
Tree createTree( ){ ... }

#endif
```

createTree

```
Tree createTree( ){
   Tree t;
   t.root = NULL;
   t.makeTree = &makeTree;
   t.preorder = &preorder;
   t.inorder = &inorder;
   t.postorder = &postorder;
   t.levelorder = &levelorder;
   t.empty = &empty;
   t.size = &size;
   t.height = &height;
   return t;
}
```

makeTree

```
void makeTree( Tree * x, int e, Tree * l, Tree * r ){
   x->root = malloc( sizeof( TreeNode ) );
   x->root->item = e;
   x->root->left = l->root;
   x->root->right = r->root;
   l->root = r->root = NULL;
   return;
}
```

preorder, inorder, postorder

```
void preorder( TreeNode * p ){
   if( p ){
      printf( "%d␣", p->item );
      preorder( p->left );
      preorder( p->right );
   }
}

void inorder( TreeNode * p ){
   if( p ){
      inorder( p->left );
      printf( "%d␣", p->item );
      inorder( p->right );
   }
}

void postorder( TreeNode * p ){
   if( p ){
      postorder( p->left );
      postorder( p->right );
      printf( "%d␣", p->item );
   }
}
```

levelorder

```
void givenLevel( TreeNode * p, int level ){
   if( !p )
      return;
   if( level == 1 )
      printf( "%d␣", p->item );
   else if( level > 1 ){
      givenLevel( p->left, level - 1 );
      givenLevel( p->right, level - 1 );
   }
}


void levelorder( TreeNode * p ){
   int h = height( p );
   int i;
   for( i = 1; i <= h; i++)
      givenLevel( p, i );
}
```

empty, size, height

```
int empty( TreeNode * p ){
   return p == NULL;
}

int size( TreeNode * p ){
   if( p )
      return 1 + size( p->left ) + size( p->right );
   else
      return 0;
}

int height( TreeNode * p ){
   int hl, hr;
   if( p ){
      hl = height( p->left );
      hr = height( p->right );
      if( hl > hr ) return ++hl;
      else return ++hr;
   }
   else
      return 0;
}
```

testbinarytree.c

```
#include <stdio.h>
#include "binarytree.h"

int main(){
   Tree a = createTree( );
   Tree x = createTree( );
   Tree y = createTree( );
   Tree z = createTree( );
   y.makeTree( &y, 1, &a, &a );
   z.makeTree( &z, 2, &a, &a );
   x.makeTree( &x, 3, &y, &z );
   y.makeTree( &y, 4, &x, &a );
   printf( "Preorder␣sequence␣is␣" );
   y.preorder( y.root );
   printf( "\n" );
   printf( "Inorder␣sequence␣is␣" );
   y.inorder( y.root );
   printf( "\n" );
   printf( "Postorder␣sequence␣is␣" );
   y.postorder( y.root );
   printf( "\n" );
   printf( "Level␣sequence␣is␣" );
```

```
  y.levelorder( y.root );
  printf( "\n" );
  printf( "empty␣=␣%d\n", y.empty( y.root ) );
  printf( "size␣=␣%d\n", y.size( y.root ) );
  printf( "height␣=␣%d\n", y.height( y.root ) );
  return 0;
}
```

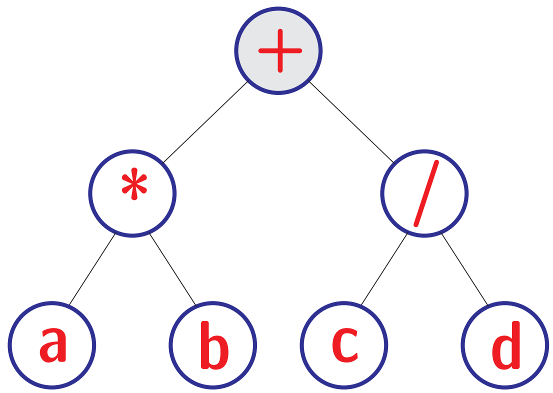
Compilation and Running

```
C:\Users\Yoan\Desktop\code\progs>gcc testbinarytree.c

C:\Users\Yoan\Desktop\code\progs>a
Preorder sequence is 4 3 1 2
Inorder sequence is 1 3 2 4
Postorder sequence is 1 2 3 4
Level sequence is 4 3 1 2
empty = 0
size = 4
height = 3

C:\Users\Yoan\Desktop\code\progs>
```

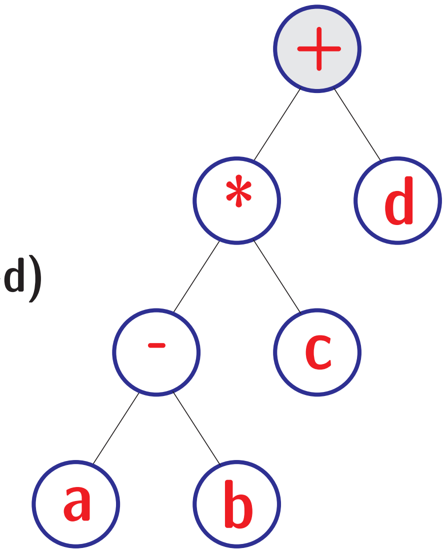
Binary Tree Application

– Expression Trees –

(a*b)+(c/d)



(((a-b)*c)+d)



Infix form:	a*b+c/d	a-b*c+d
Prefix form:	+*ab/cd	+*-abcd
Postfix form:	ab*cd/+	ab-c*d+

- Infix: ambiguous; pre/postfix: unambiguous
- Postfix evaluation:
  - ◇ Scan left-to-right
  - ◇ Put operator to a stack
  - ◇ Apply the encountered operator to the operands in the stack and delete them from the stack.