ต้นไม้แบบทวิภาค

(Binary Trees)

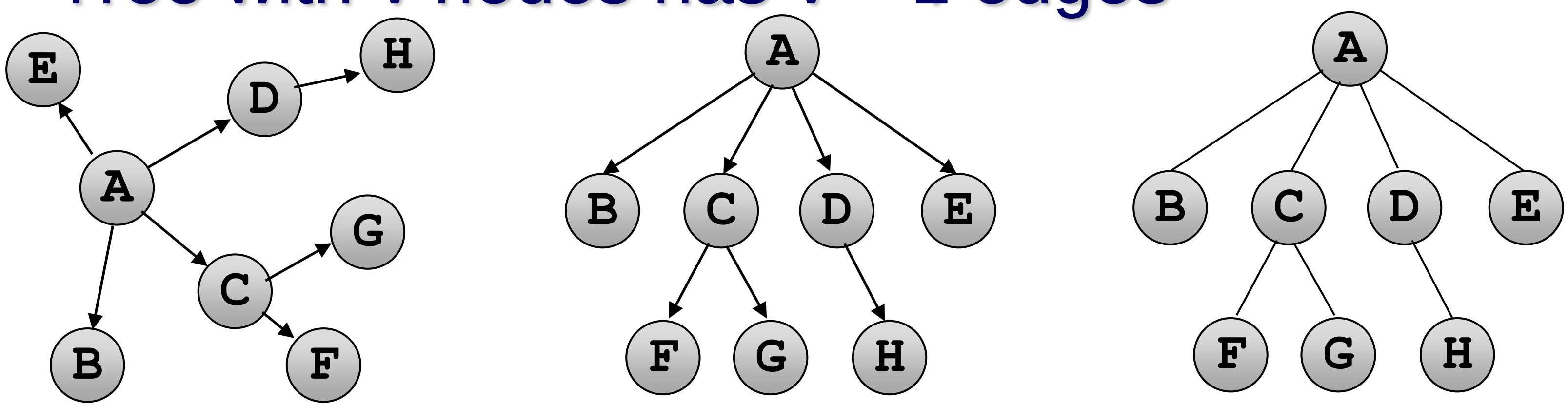
สมชาย ประสิทธิ์จูตระกูล
Translated to English by
Nuttapong Chentanez

Topics

- Tree Definition
- Tree Implementation
- Binary Tree
  - Huffman Tree
  - Expression Tree
  - Tree traversal
  - Expression Tree Evaluation
  - Expression Tree Differentiation

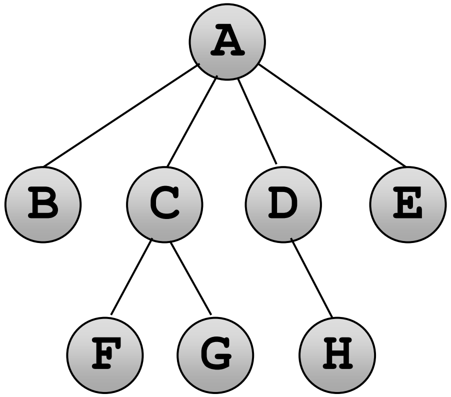
Tree

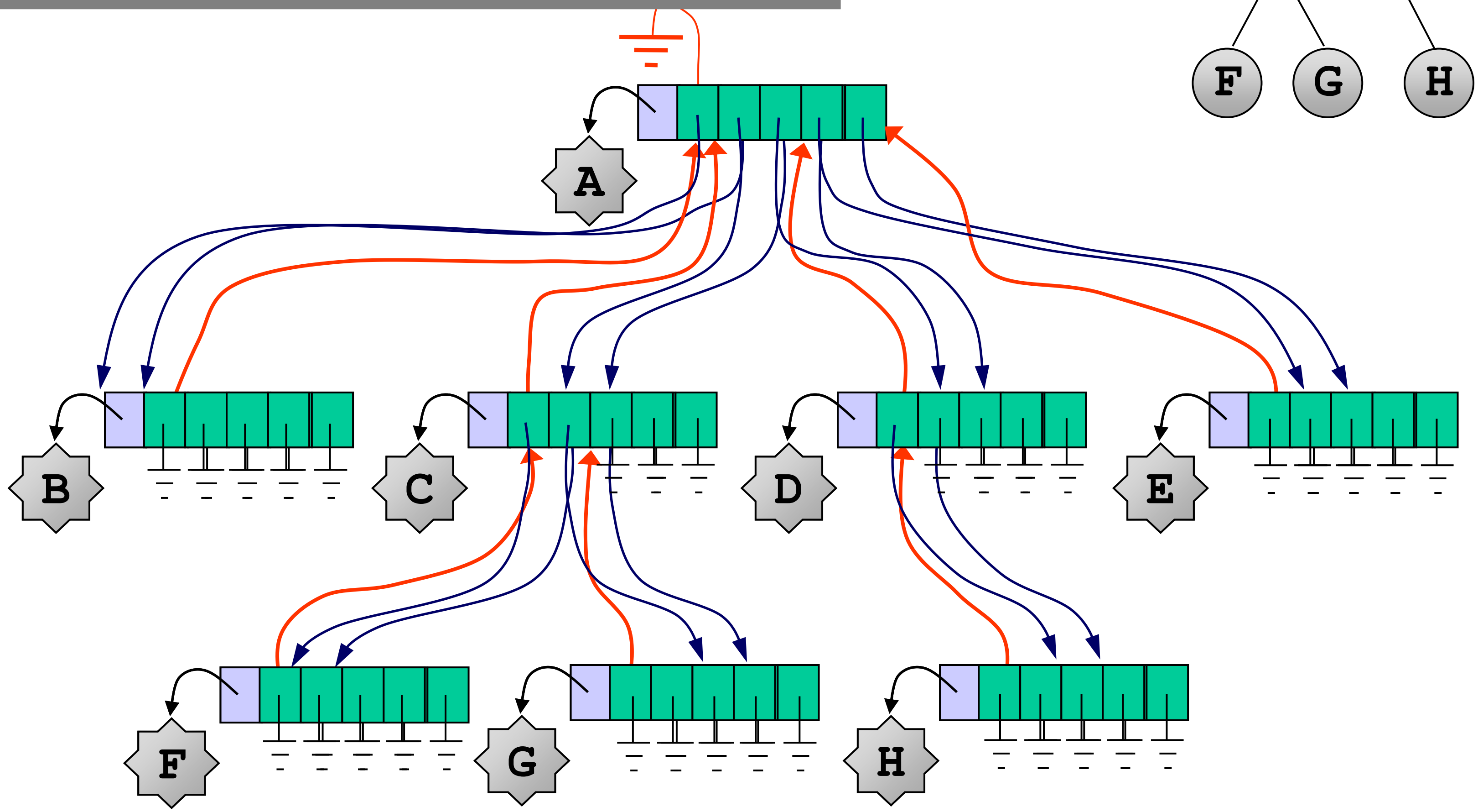
- Tree consists of nodes and edges
- Edge has direction (Directed Edge)
- A is the parent of B when there's an edge from A to B
- Each node has only one parent
  (Except root, which has no parent)
- Tree with v nodes has v – 1 edges



Tree Implementation : Use array to store children

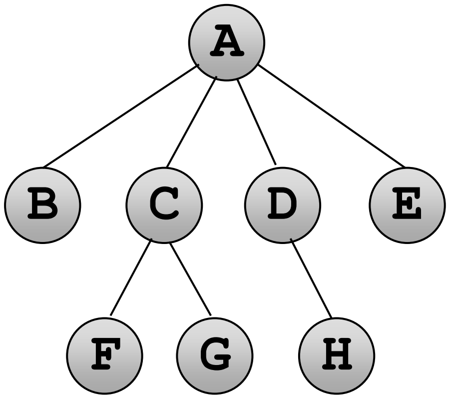
```
class node {
  T     data;
  node *children[4];
};
```

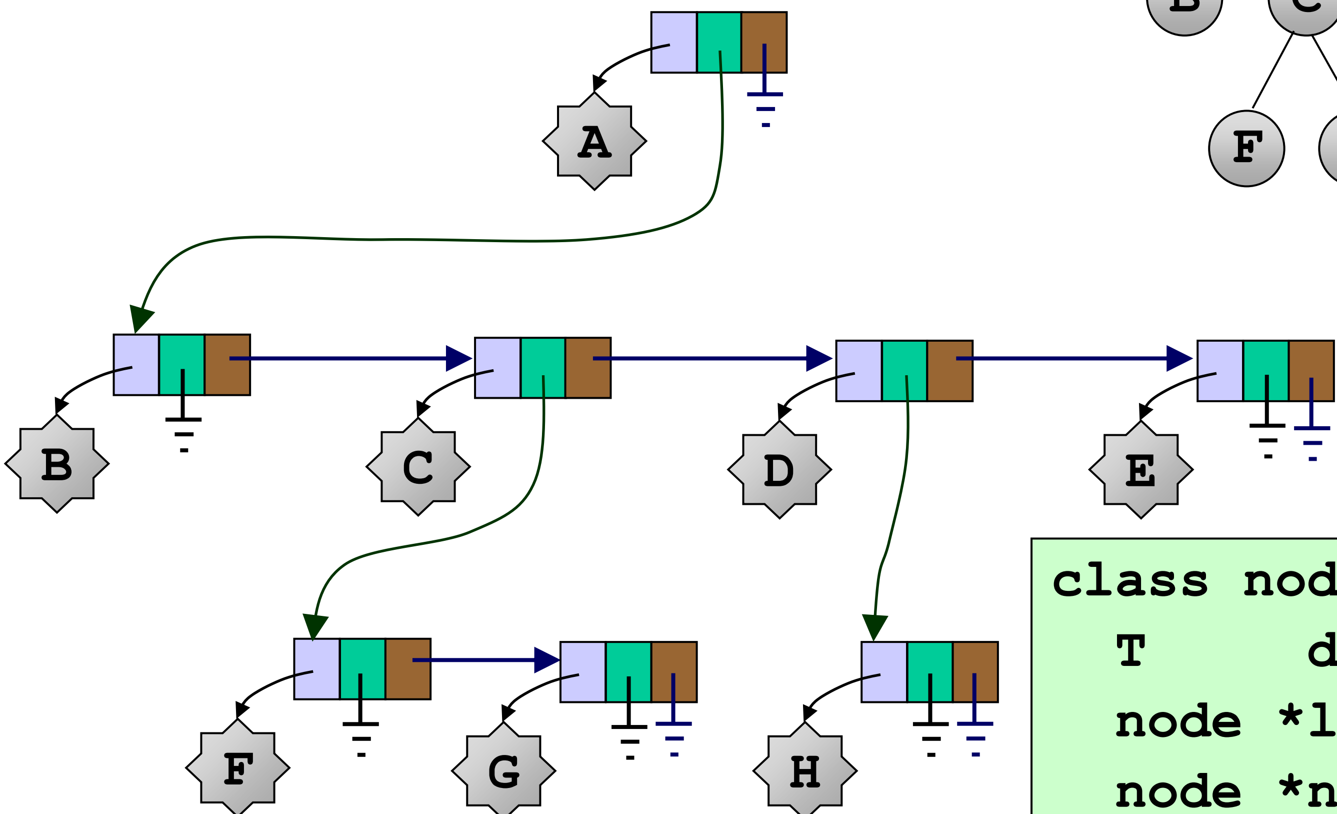




Tree Implementation : Use list to store children

- Store link to the left most child
- Each child link to the next child

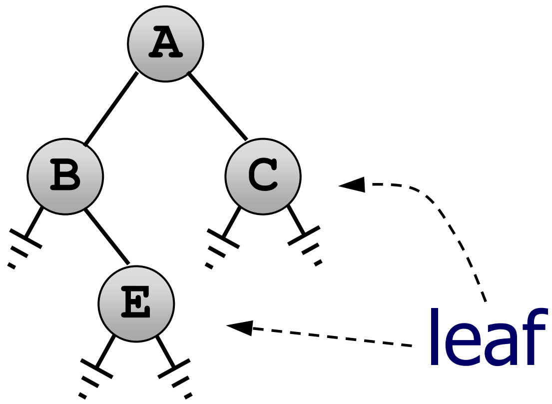


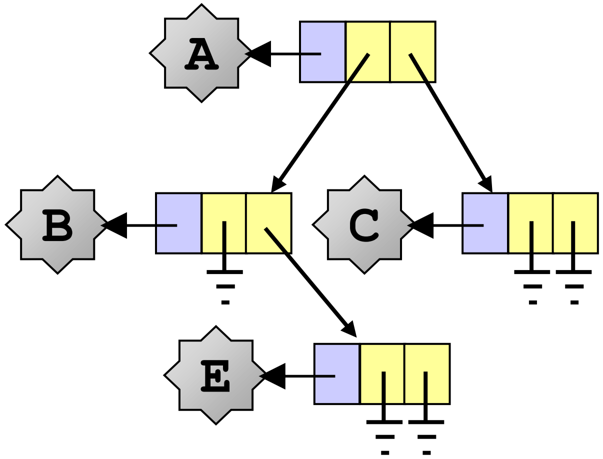



```
class node{
  T       data;
  node *leftChild;
  node *nextSibling;
};
```

Binary Tree

- Each node has two children: left and right






```
class node {
  T     data;
  node *left, *right;

  node(T data, node *left, node *right) :
    data(data), left(left), right(right)
  {}

  bool isLeaf() {
    return left==NULL && right==NULL;
};
```

Huffman Code

งึกงึกงักงักมันเป็นงึกงึกงักงัก
งึกงึกงักงักมันเป็นงึกงึกงักงัก
มันเป็นกะอึกกะอักมันเป็นจึกจึกจั๊กจั๊ก

ก	ง	◌ั	◌ึ	น	จ	ป	ม	เ	◌็	อ	ะ	◌๊
24	16	15	11	8	4	4	4	4	4	2	2	2
0000	0001	0010	1010	0011	0100	0101	0110	0111	1000	1001	1011	1100
11	010	000	001	0110	01110	01111	1000	1001	10100	10101	10110	10111

$4 \times (24 + 16 + 15 + 11 + 8 + 4 + 4 + 4 + 4 + 4 + 2 + 2 + 2) = 400$ bits

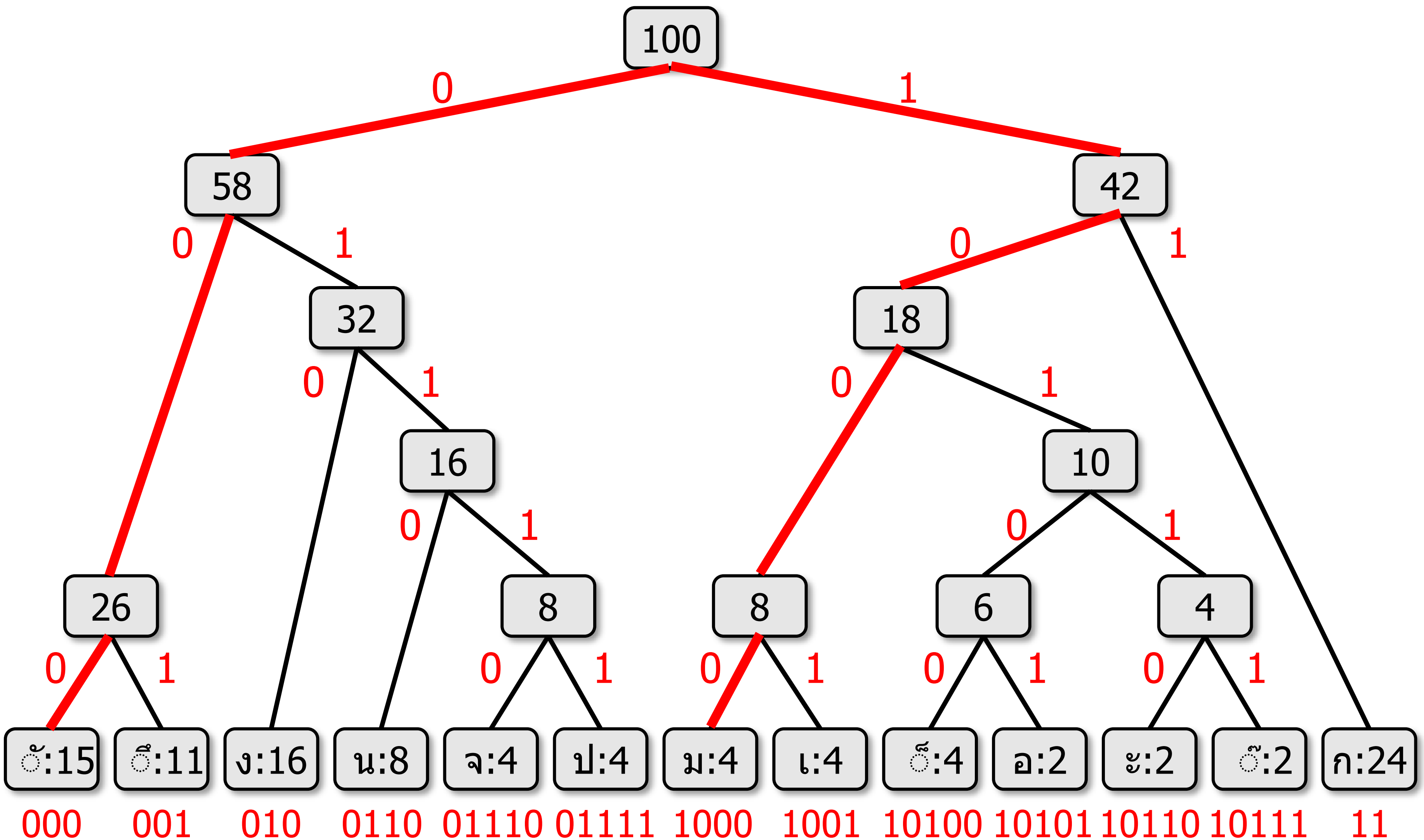
variable-length code

prefix-free code

$$\left.\begin{array}{l} 2 \times 24 + \\ 3 \times (16 + 15 + 11) + \\ 4 \times (8 + 4 + 4) + \\ 5 \times (4 + 4 + 4 + 2 + 2 + 2) \end{array}\right\} = 328 \text{ bits}$$

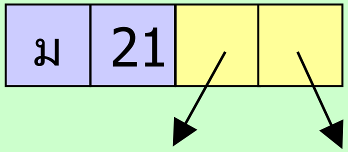
No code is a prefix of another code

How to find Huffman code?

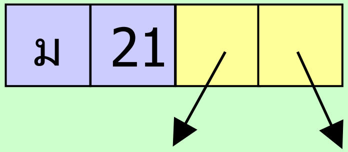


huffman_tree

```
class huffman_tree {
  class node {
    public:
      char c;
      int  freq;
      node *left, *right;

      node(char c, int freq, node *left, node *right) :
        c(c), freq(freq), left(left), right(right)
      {}
  };

  node *root;
```

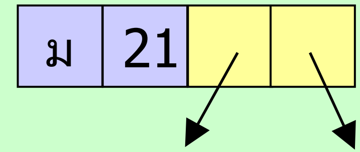


huffman_tree: ctor




```
class huffman_tree {
  class node { ... }
  node *root;
public:
  huffman_tree(char c[], int f[], size_t n) {
    class freq_comp {
      public:
        bool operator()(node *a, node *b) {
            return a->freq > b->freq;
        }
    };
    priority_queue<node*,vector<node*>, freq_comp > h;
    for (size_t i=0; i<n; ++i) {
      h.push(new node(c[i], f[i], NULL, NULL));
    }
    for (size_t i=0; i<n-1; ++i) {
      node *n1 = h.top(); h.pop();
      node *n2 = h.top(); h.pop();
      h.push(new node('*', n1->freq+n2->freq, n1, n2));
    }
    root = h.top();
  }
```

huffman_tree: dtor

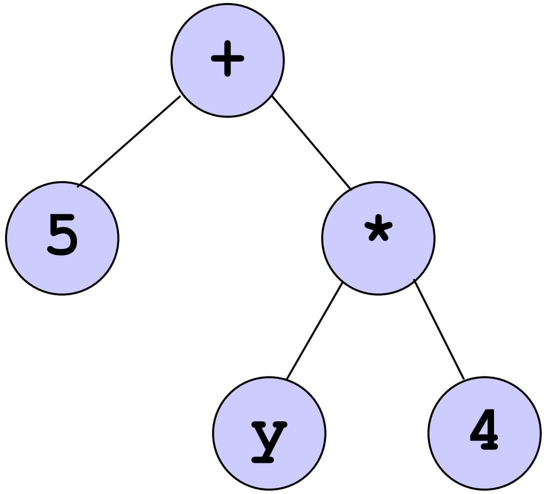


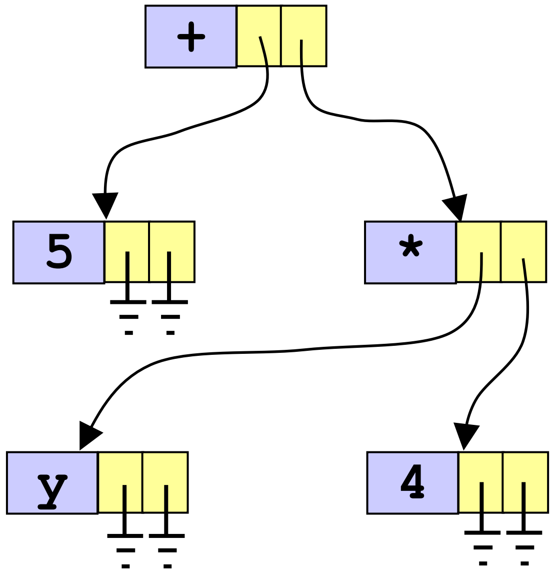

```
class huffman_tree {
  ...
  node *root;

public:
  huffman_tree(char c[], int f[], int n) { ... }

 ~huffman_tree() {
    delete_all_node( root );
  }
  void delete_all_nodes(hnode *r) {
    if (r == NULL) return;
    delete_all_nodes(r->left);
    delete_all_nodes(r->right);
    delete r;
  }
```

Expression Tree

- Represent expression as tree
- Leaf : operand
- Inner Node : operator

```
5 + y * 4
```

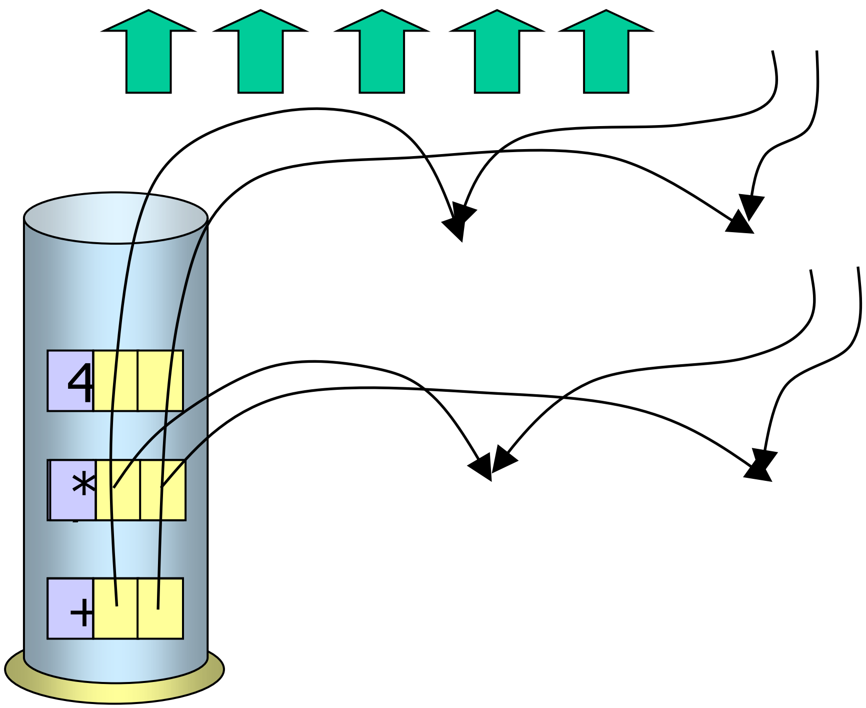




Expression Tree Construction

- Operand: push to stack
- Operator: pop two nodes from stack to be the children of the new node, then push to stack

```
   infix : 5  +  y  *  4
 postfix : 5  y  4  *  +
```

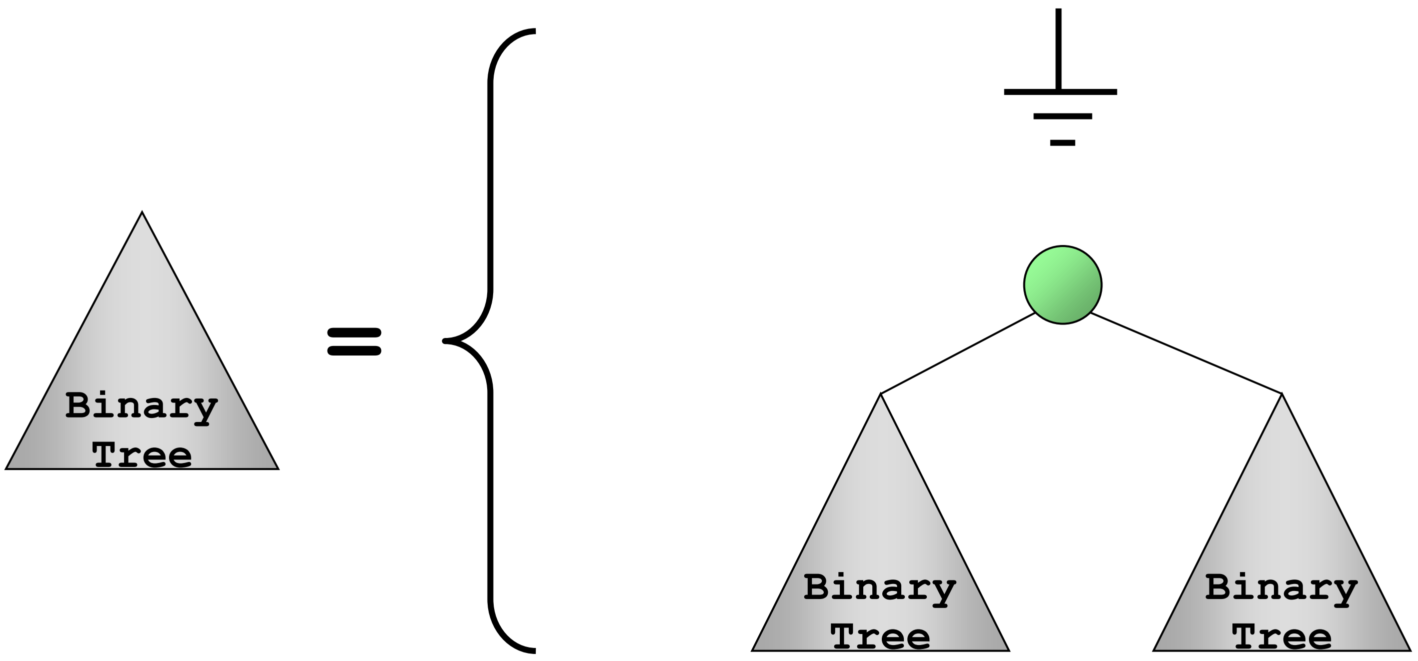


expression_tree

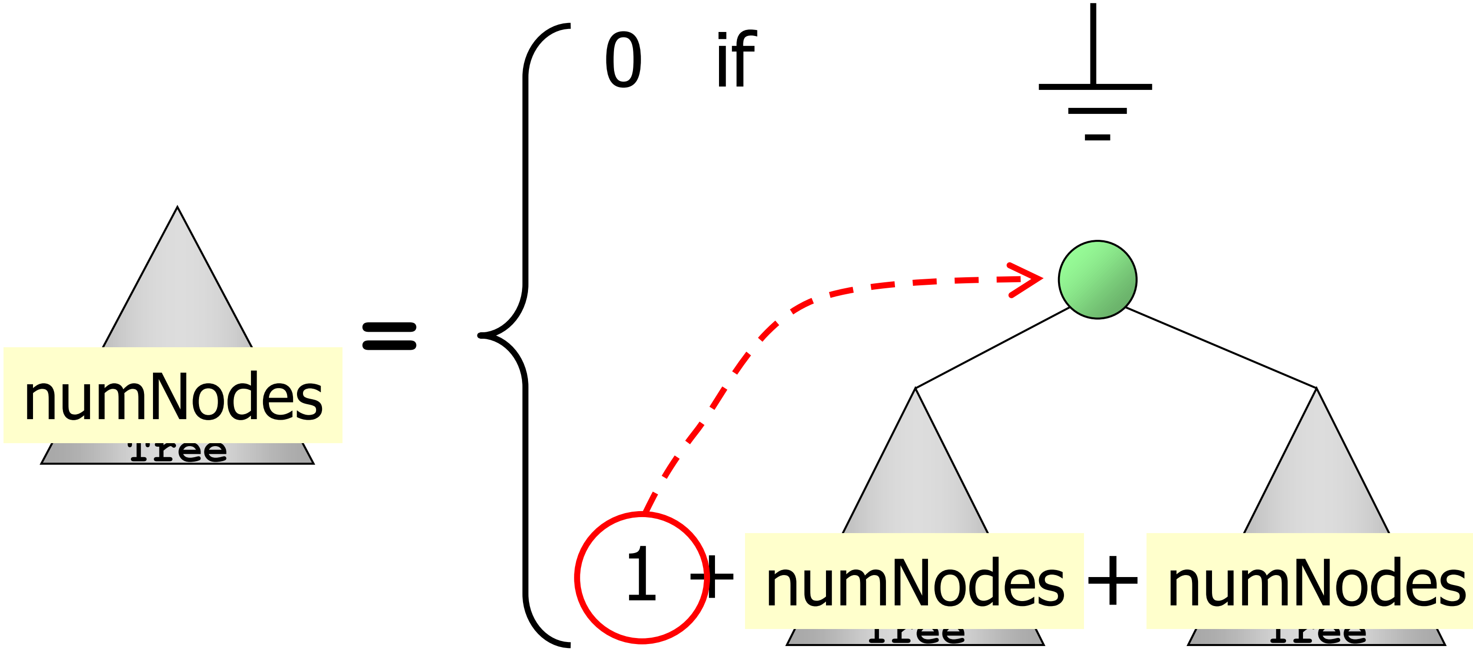
```
node *expression_tree(string &infix) {
    string postfix = infix2postfix(infix);
    stack<node*> s;
    for (size_t i=0; i<postfix.size(); i++) {
      char token = postfix[i];
      if (!isOperator(token)) {
        s.push(new node(token, NULL, NULL));
      } else {
        node *right = s.top(); s.pop();
        node *left  = s.top(); s.pop();
        s.push(new node(token, left, right));
      }
    }
    return s.top();
}
```

Binary tree: recursive view

- Binary tree is
  - Empty tree (NULL) or
  - A node with binary trees as left and right children

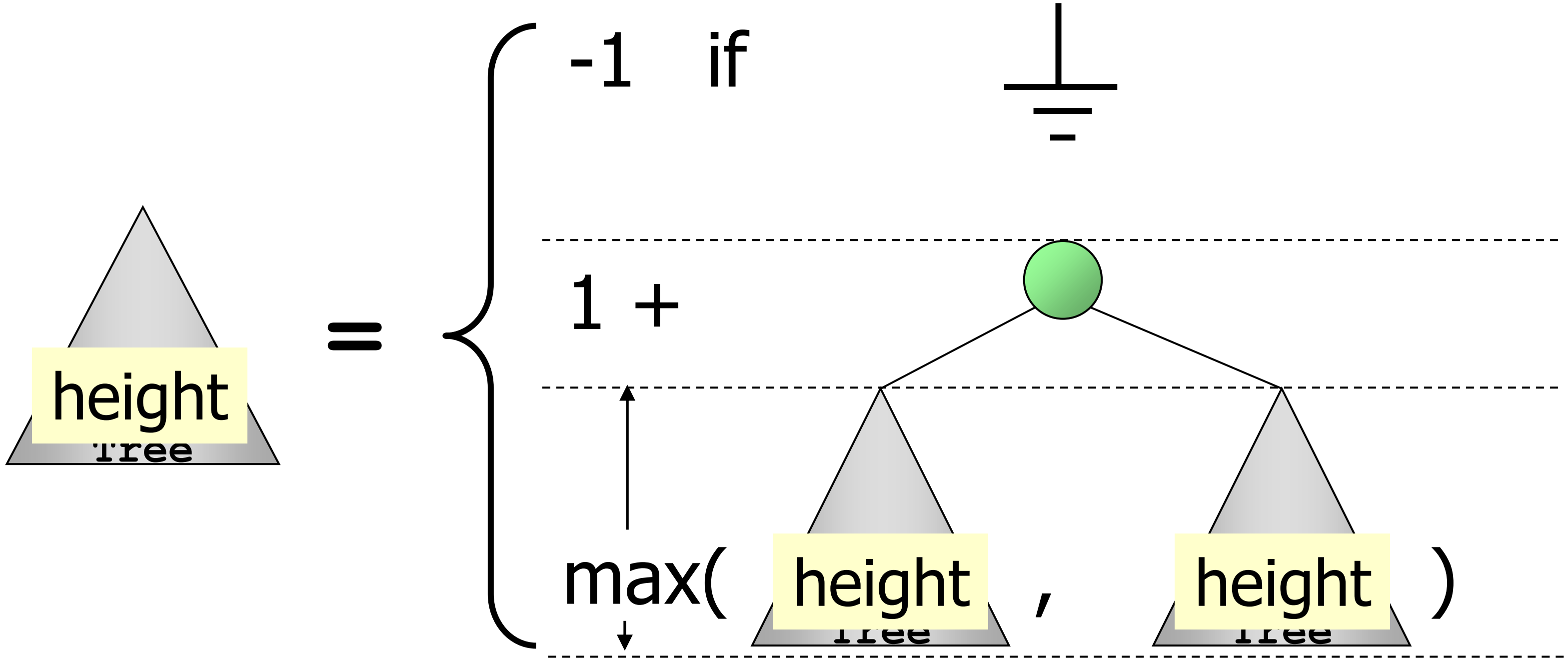


Counting number of nodes



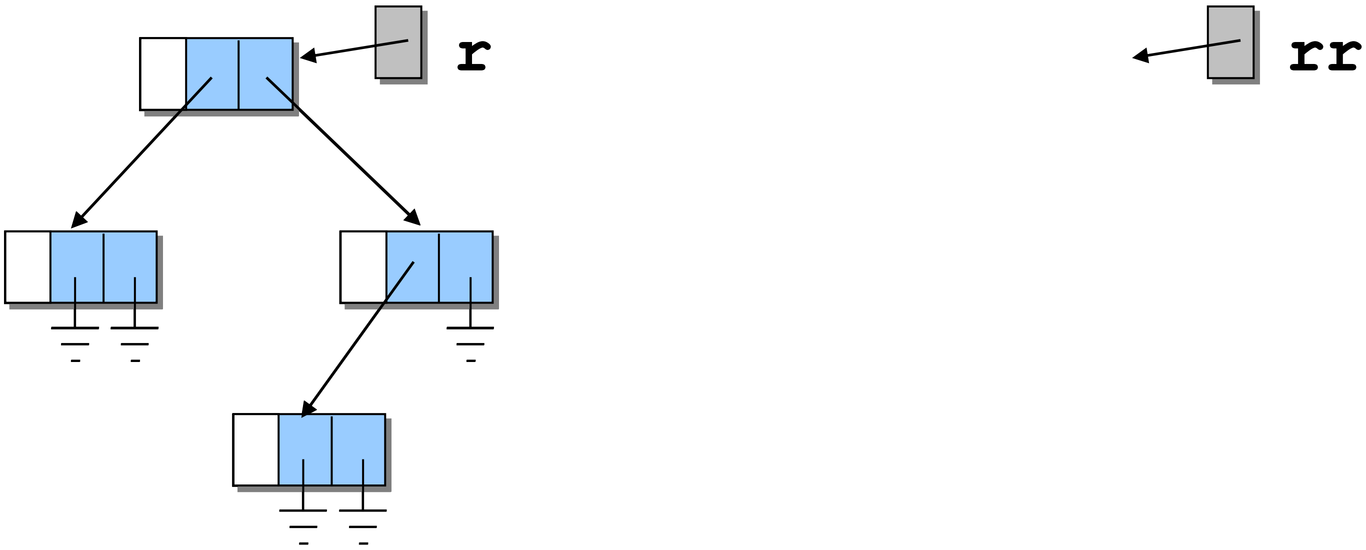

```
int numNodes(node *r) {
  if (r == NULL) return 0;
  return 1 + numNodes(r->left)
           + numNodes(r->right);
}
```

Finding height



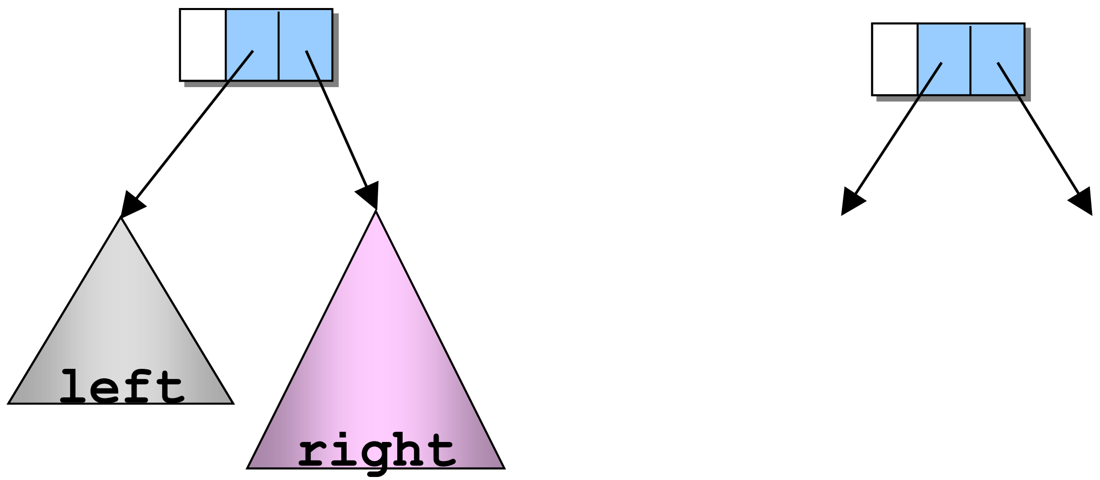

```
int height(node *r) {
  if (r == NULL) return -1;
  return 1 + max(height(r->left),
                 height(r->right));
}
```

Copying Tree




```
node *rr = copy(r);
```

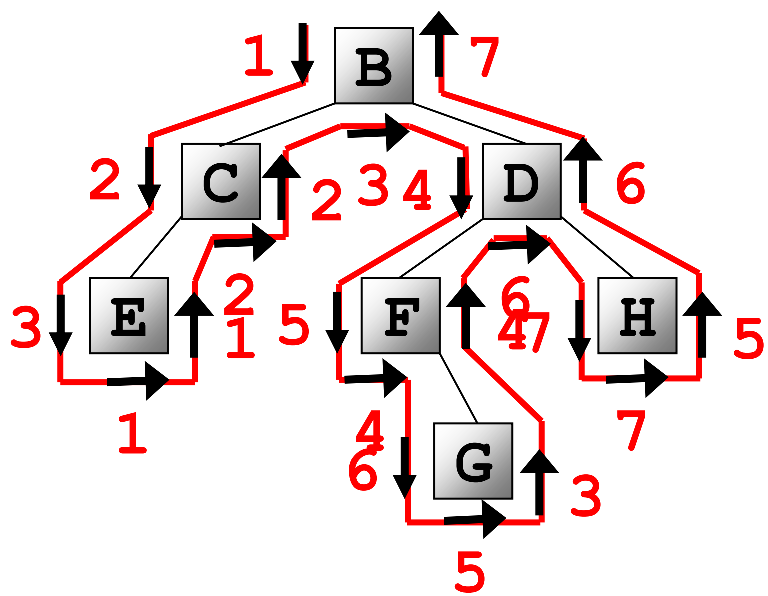
Copying tree (Recursive View)




```
node *copy(node *r) {
    if (r == NULL) return NULL;
    node *rL = copy(r->left);
    node *rR = copy(r->right);
    return new node(r->data, rL, rR);
}
```

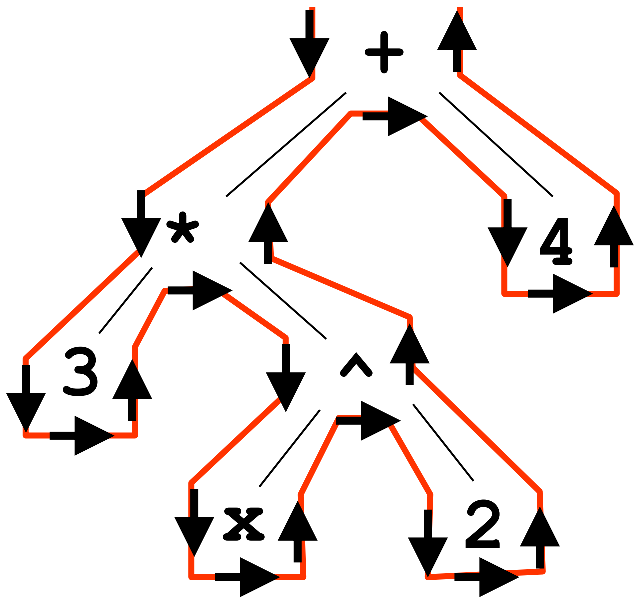
Tree traversal

- Tree traversal: mechanism to visit all nodes in the tree once per node
  - Preorder (แบบก่อนลำดับ) B, C, E, D, F, G, H
  - Inorder (แบบตามลำดับ) E, C, B, F, G, D, H
  - Postorder (แบบหลังลำดับ) E, C, G, F, H, D, B



Expression tree traversal

- Preorder traversal => Prefix expression
- Inorder traversal => Infix expression
- Postorder traversal => Postfix expression



+	*	3	^	x	2	4
3	*	x	^	2	+	4
3	x	2	^	*	4	+

Tree traversal with node x as root

- Preorder
  - visit x
  - traverse left
  - traverse right
- Inorder
  - traverse left
  - visit x
  - traverse right
- Postorder
  - traverse left
  - traverse right
  - visit x

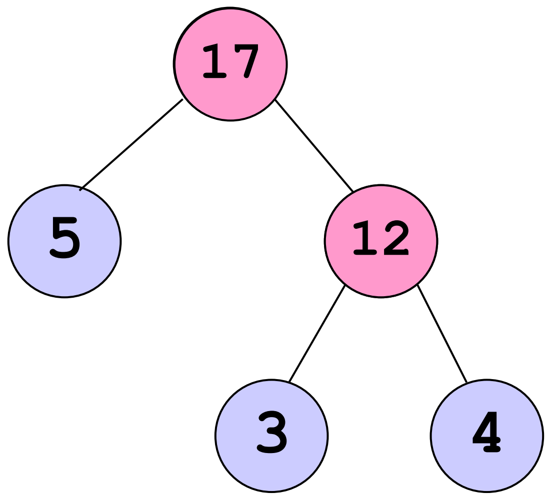
```
void preorder(node *x) {
   if (x == NULL) return;
   visit(x->data);
   preorder(x->left);
   preorder(x->right);
}
```

```
void inorder(node *x) {
   if (x == NULL) return;
   inorder(x->left);
   visit(x->data);
   inorder(x->right);
}
```

```
void postorder(node *x) {
   if (x == NULL) return;
   postorder(x->left);
   postorder(x->right);
   visit(x->data);
}
```

Expression tree evaluation

- Need to know the values of children first
- Similar to postorder traversal



Expression Tree Evaluation

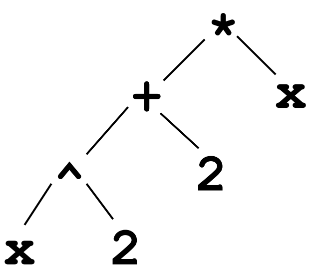
```
float eval(node *r) {
    if (r == NULL) return 0;
    if (r->isLeaf()) return (r->data - '0');
    float vLeft = eval(r->left);
    float vRight = eval(r->right);
    if (r->data == '+') return vLeft + vRight;
    if (r->data == '-') return vLeft - vRight;
    if (r->data == '*') return vLeft * vRight;
    if (r->data == '/') return vLeft / vRight;
    if (r->data == '^') return pow(vLeft, vRight);
    exit(-9); // Should never reach this line
}
```

Differentiation

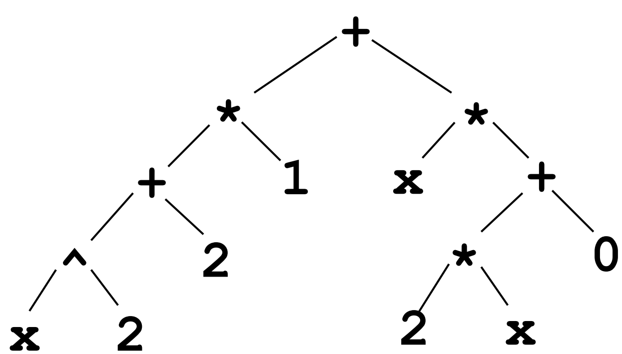
$$f(x) = (x^2 + 2)x$$

$$f'(x) = (x^2 + 2)\cdot 1 + x \cdot (2x + 0)$$

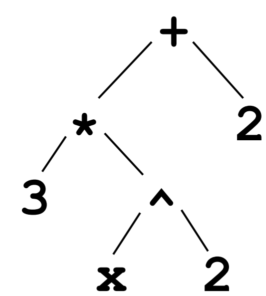
$$= 3x^2 + 2$$



`f`



`diff(f)`



`simplify(f)`

Rules

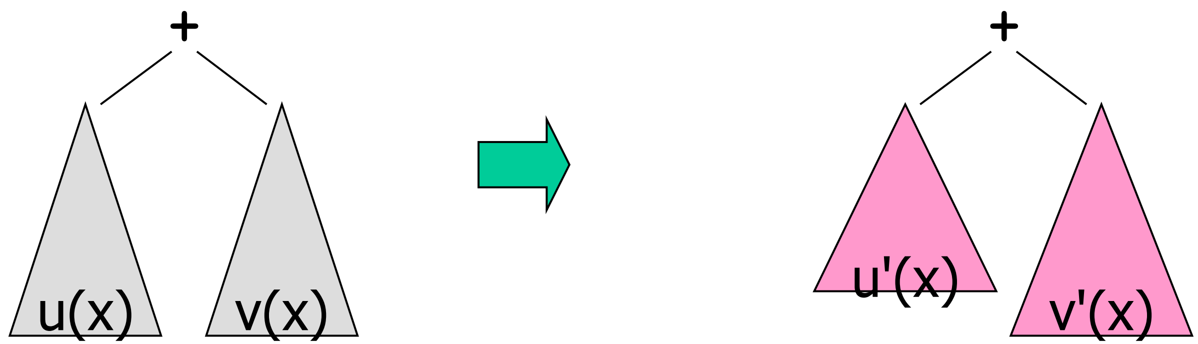
$$\left(u(x)+v(x)\right)' = u'(x)+v'(x)$$

$$\left(u(x)v(x)\right)' = v(x)u'(x)+u(x)v'(x)$$

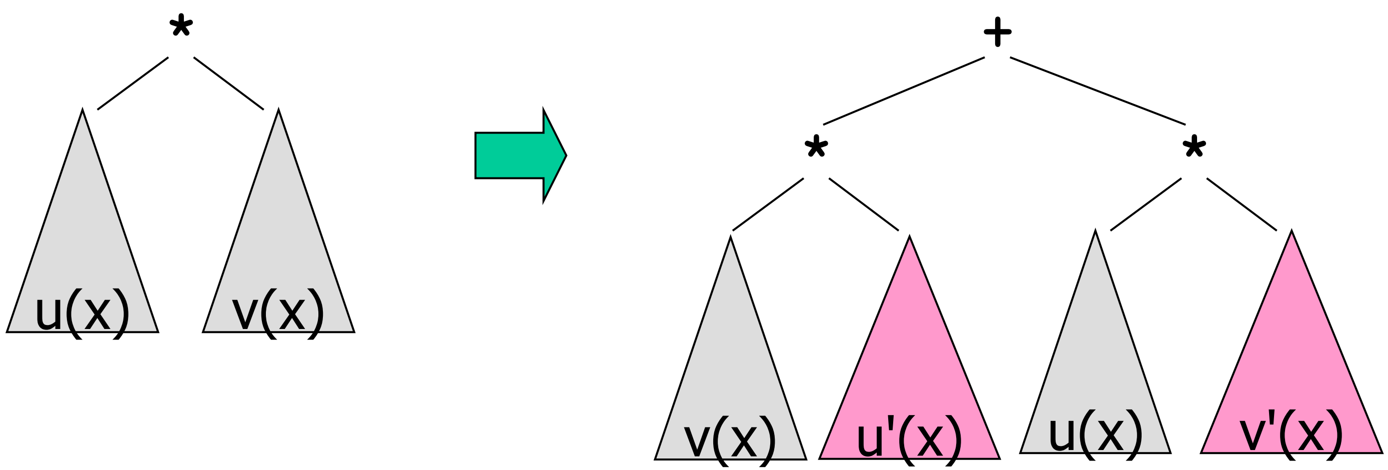
$$\left(\frac{u(x)}{v(x)}\right)' = \frac{v(x)u'(x)-u(x)v'(x)}{\left(v(x)\right)^2}$$

$$\left(\left(u(x)\right)^C\right)' = C\left(u(x)\right)^{C-1}u'(x)$$

Derivative of expression trees: +, *

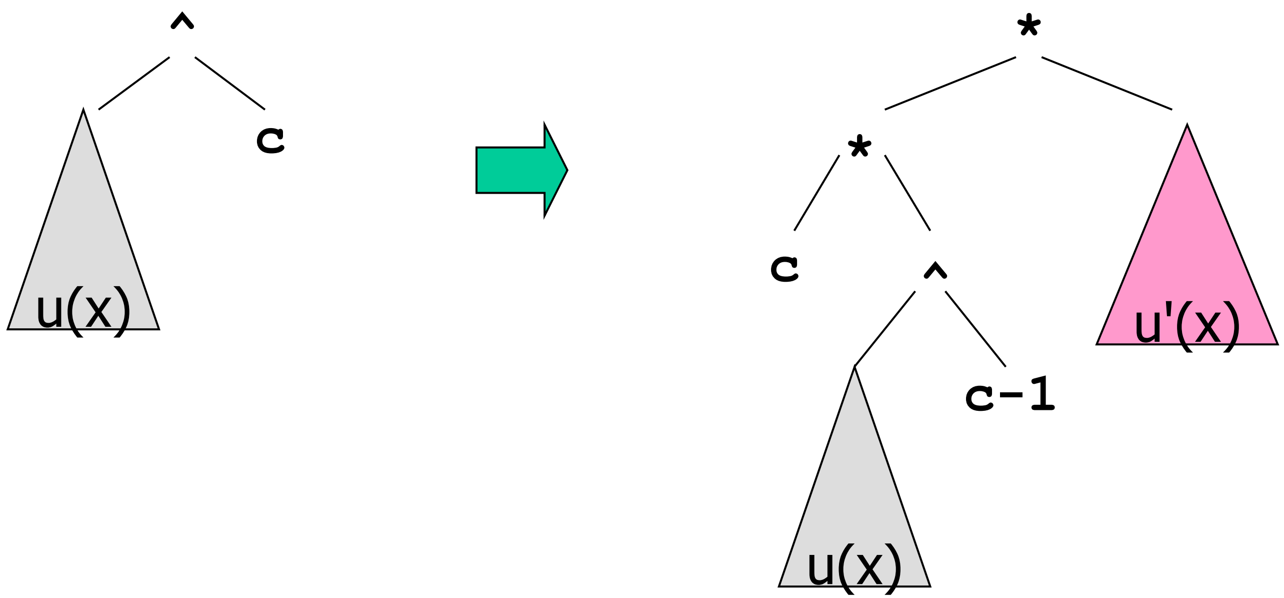


$$\left(u(x)+v(x)\right)' = u'(x)+v'(x)$$



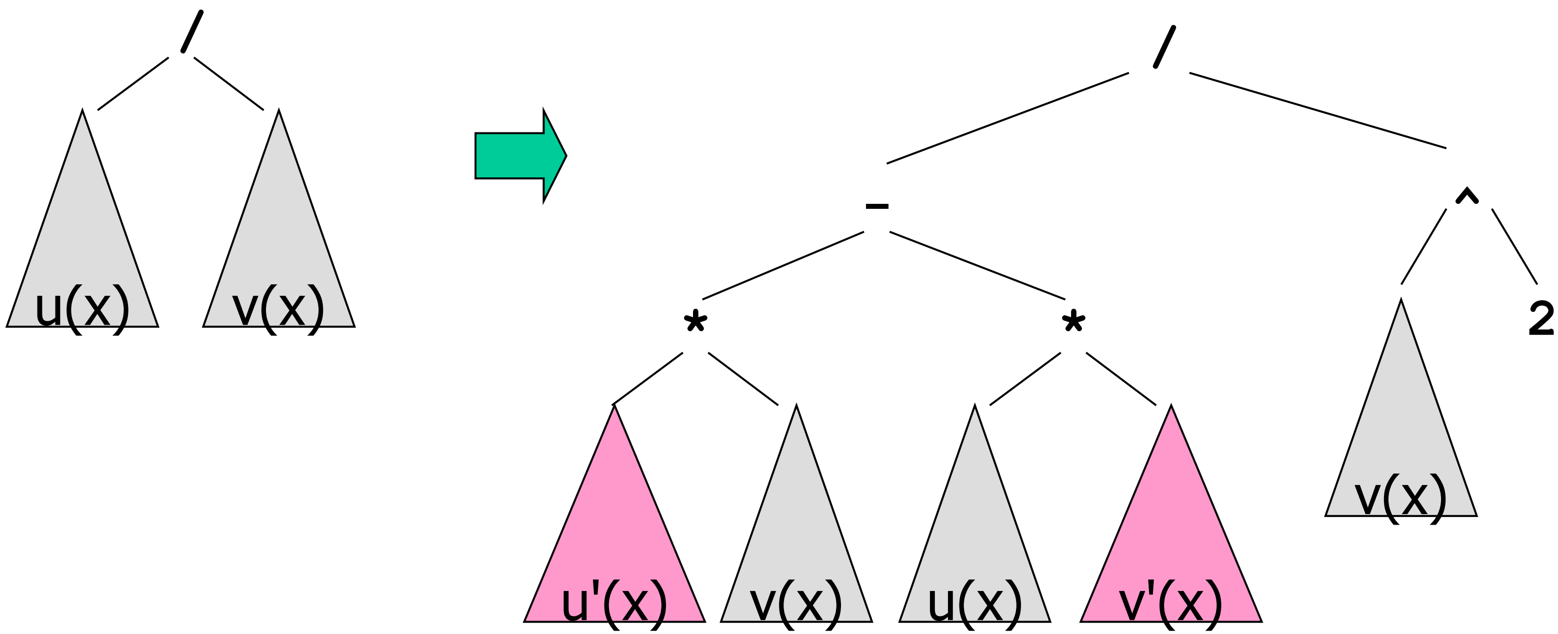
$$\left(u(x)v(x)\right)' = v(x)u'(x)+u(x)v'(x)$$

Derivative of expression tree : ^



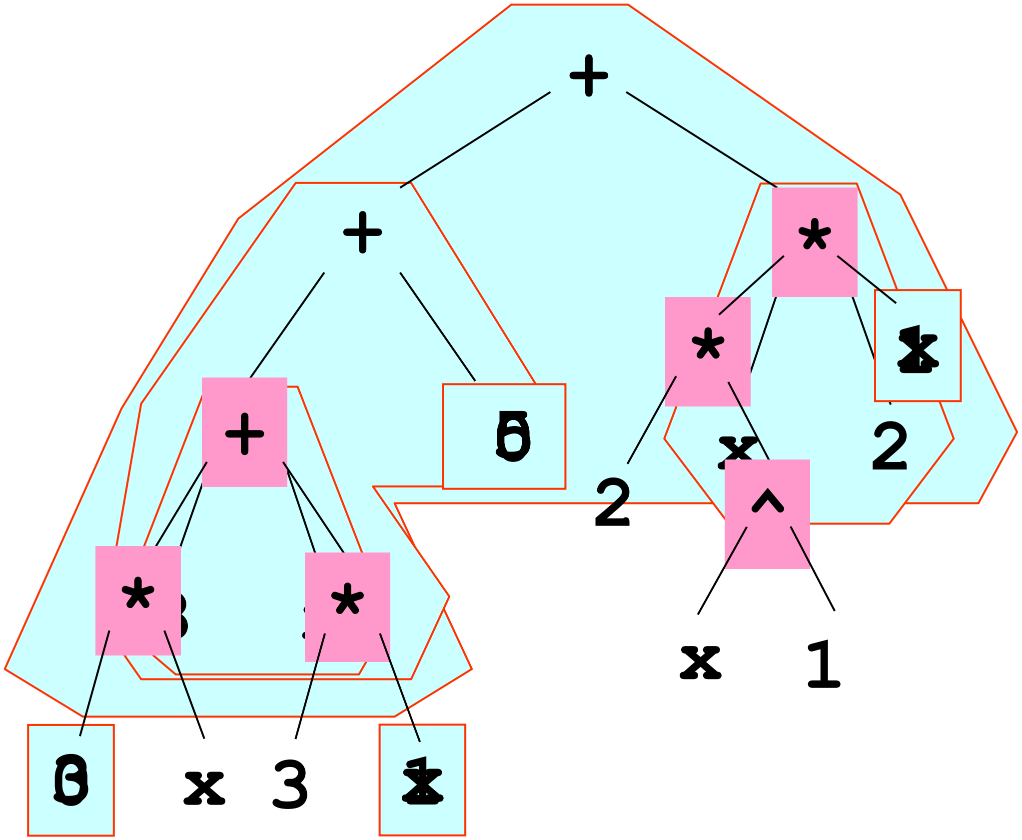
$$\left(\left(u(x)\right)^{C}\right)' = C\left(u(x)\right)^{C-1} u'(x)$$

Derivative of expression tree : /



$$\left(\frac{u(x)}{v(x)}\right)' = \frac{v(x)u'(x) - u(x)v'(x)}{\left(v(x)\right)^2}$$

Example of expression tree differentiation



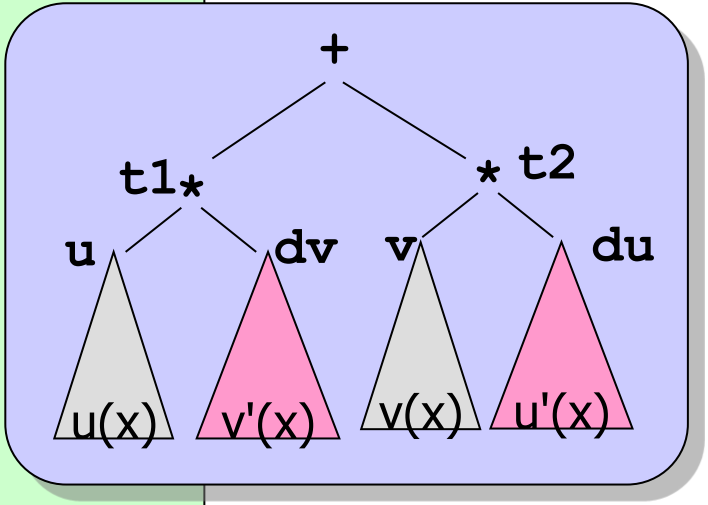
Differentiation

```
node *diff(node *r) {
    if (r == NULL) return NULL;
    char s = r->data;
    if ( r.isLeaf() ) {
      r->data = (s == 'x' ? '1' : '0');
    } else {
      if (s == '+')        r = diffSum(r);
      else if (s == '-') r = diffSum(r);
      else if (s == '^') r = diffPow(r);
      else if (s == '*') r = diffMult(r);
      else if (s == '/') r = diffDiv(r);
    }
    return r;
}
```

Differentiation : +, *

```
node *diffSum(node *r) {
    r->left = diff(r->left);
    r->right = diff(r->right);
    return r;
}

node *diffMult(node *r) {
    node *u = copy(r->left);
    node *v = copy(r->right);
    node *du = diff(r->left);
    node *dv = diff(r->right);
    node *t1 = new node('*', u, dv);
    node *t2 = new node('*', v, du);
    return new node('+', t1, t2);
}
```

$$\left(u(x)+v(x)\right)' = u'(x)+v'(x)$$

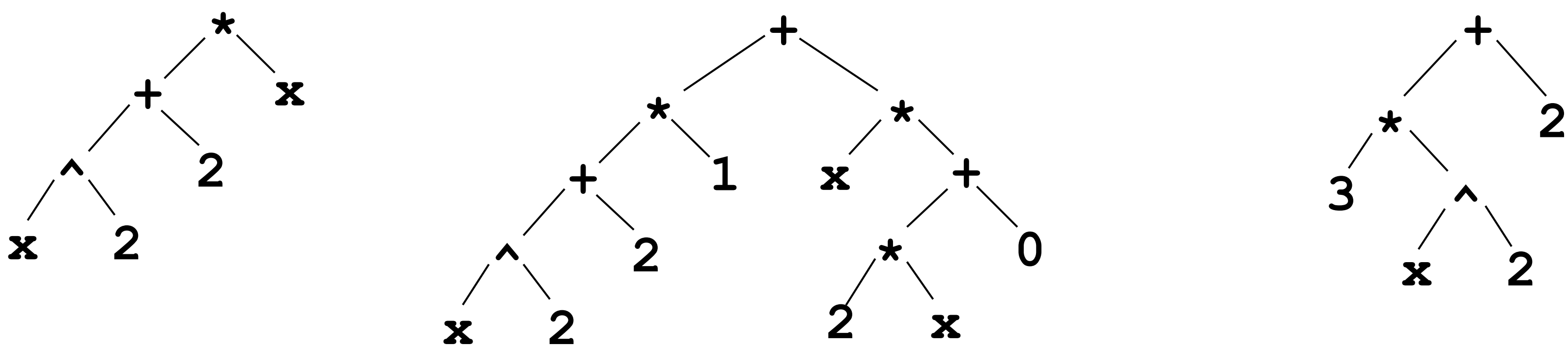


$$\left(u(x)v(x)\right)' = v(x)u'(x)+u(x)v'(x)$$

Put everything into code

```
char infix[100];
scanf("%s", infix);
node *r = newExpressionTree(infix);
printInorder(r); printPostorder(r);
r = diff(r);
r = simplify(r);
printInorder(r); printPostorder(r);
```

```
infix = "(x^2+2)*x"
```



r **diff(r)** **simplify(r)**

Summary

- Tree is a kind of data structure
  - Can be constructed by linking nodes
- Binary tree is a tree whose nodes has 2 children
  - Tree consists of small trees
  - Most operations can be naturally expressed with recursive functions
  - Tree operations can be done by preorder, inorder, or postorder traversal