# C Programming Basic – week 9

*Tree*

**Lecturers :**
**Cao Tuan Dung**
**Le Duc Trung**
**Dept of Software Engineering**
**Hanoi University of Technology**

1

# Topics of this week

- How to build programs using makefile utility
- Tree traversal
  - Depth first search
    - Preorder traversal
    - Inorder traversal
    - Postorder traversal
  - Breadth first search.
- Exercises

2

# Makefile - motivatio

- Small programs ⟶ single file
- "Not so small" programs :
  – Many lines of code
  – Multiple components
  – More than one programmer

- Problems:
  – Long files are harder to manage
    (for both programmers and machines)
  – Every change requires long compilation
  – Many programmers cannot modify the
    same file simultaneously

# Makefile - motivation

- Solution : divide project to multiple files
- Targets:
  – Good division to components
  – Minimum compilation when something is
    changed
  – Easy maintenance of project structure,
    dependencies and creation

# Project maintenance

- Done in Unix by the Makefile mechanism
- A makefile is a file (script) containing :
  - Project structure (files, dependencies)
  - Instructions for files creation
- The make command reads a makefile, understands the project structure and makes up the executable
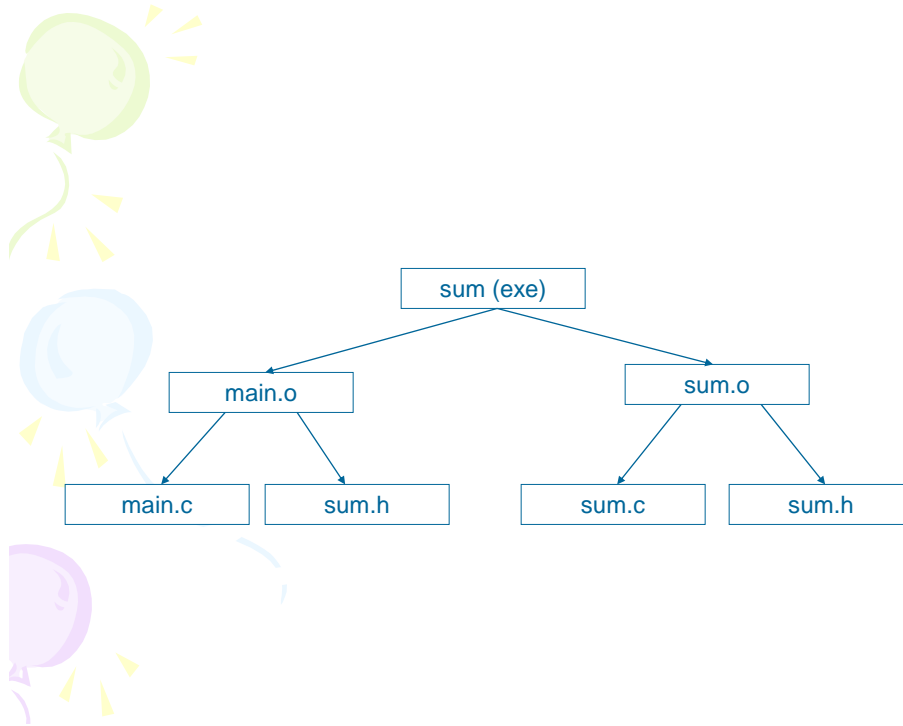- Note that the Makefile mechanism is not limited to C programs

# Project structure

- Project structure and dependencies can be represented as a DAG (= Directed Acyclic Graph)
- Example :
  - Program contains 3 files
  - main.c., sum.c, sum.h
  - sum.h included in both .c files
  - Executable should be the file sum

```
                    ┌─────────────┐
                    │  sum (exe)  │
                    └─────────────┘
                   ↙               ↘
        ┌──────────┐               ┌──────────┐
        │  main.o  │               │  sum.o   │
        └──────────┘               └──────────┘
        ↙          ↘               ↙          ↘
┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐
│  main.c  │ │  sum.h   │ │  sum.c   │ │  sum.h   │
└──────────┘ └──────────┘ └──────────┘ └──────────┘
```

## makefile

sum: main.o sum.o
   gcc –o sum main.o sum.o

main.o: main.c sum.h
   gcc –c main.c

sum.o: sum.c sum.h
   gcc –c sum.c

# Rule syntax

main.o: main.c sum.h
    gcc –c main.c

Rule

tab

dependency          action

9

# Equivalent makefiles

- .o depends (by default) on corresponding .c file. Therefore, equivalent makefile is:

```
sum: main.o sum.o
    gcc –o sum main.o sum.o

main.o: sum.h
    gcc –c main.c

sum.o: sum.h
    gcc –c sum.c
```
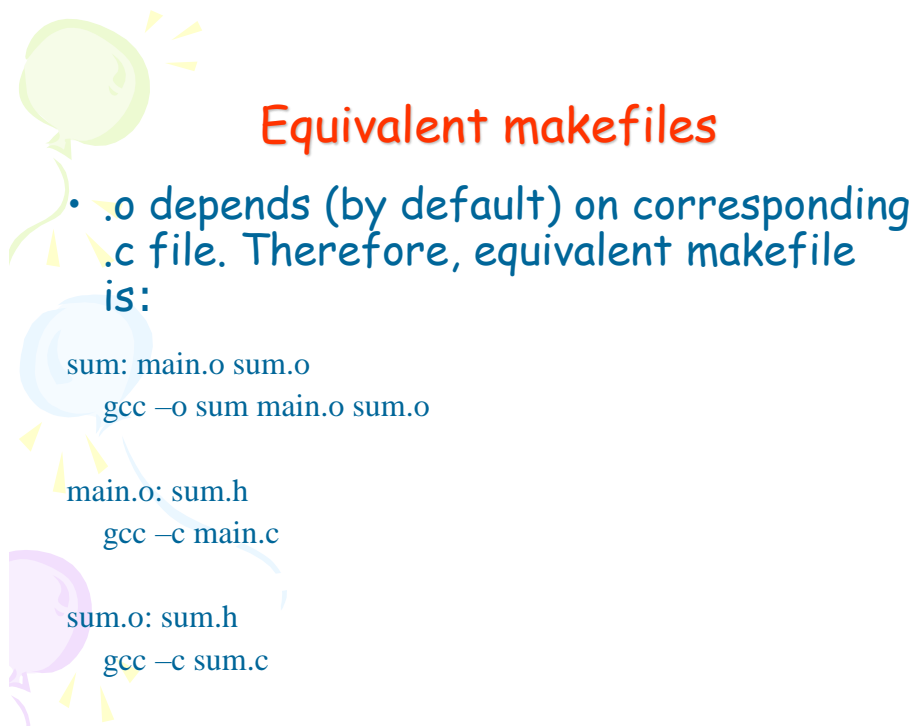
10

5
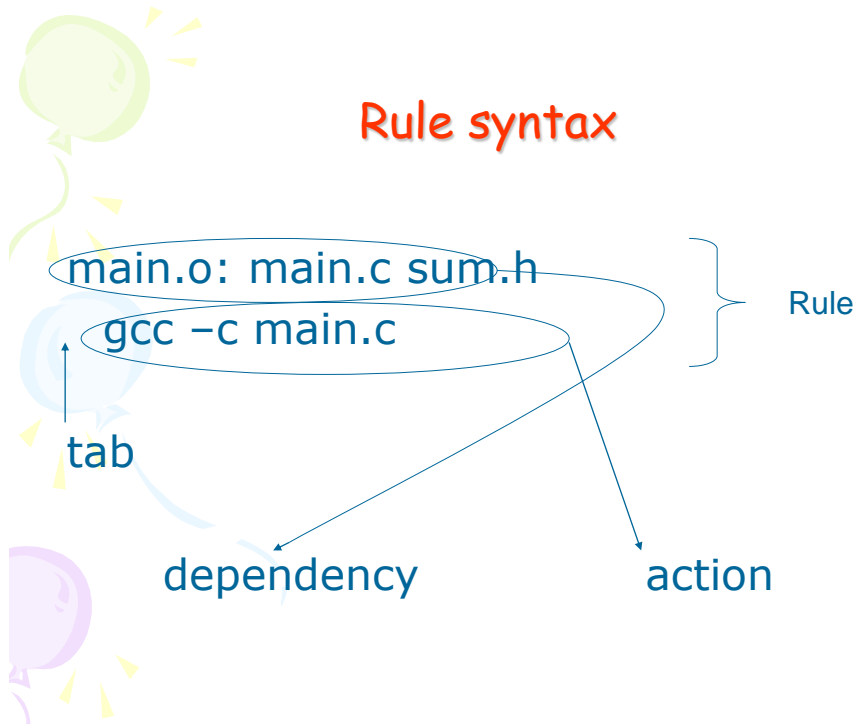
# Equivalent makefiles - continued

- We can compress identical dependencies and use built-in macros to get another (shorter) equivalent makefile :

sum: main.o sum.o

    gcc –o $@ main.o sum.o

main.o sum.o: sum.h

    gcc –c $*.c

11

# Binary Tree Traversal

- Many binary tree operations are done by performing a traversal of the binary tree
- In a traversal, each element of the binary tree is visited exactly once
- During the visit of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken

12

# Binary Tree Traversal
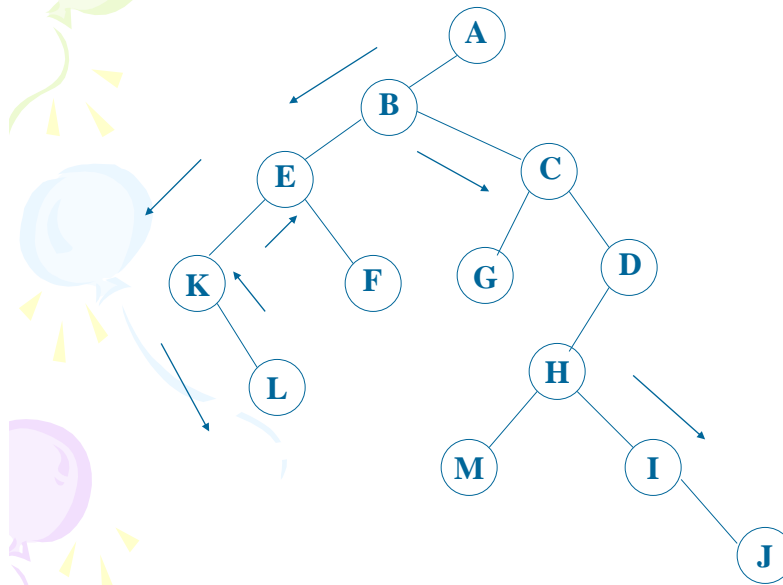
# DFS

- Depth-first search (traversal): This strategy consists of searching deeper in the tree whenever possible.
- Tree types:
  - Preorder
  - Inorder
  - Postorder

# Inorder Traversal

- Visit the nodes in the left subtree, then visit the root of the tree, then visit the nodes in the right subtree

**Tree**

A E H J M T Y

'J'

'E'

'T'

'A'

'H'

'M'

'Y'

**Print left subtree first**          **Print right subtree last**
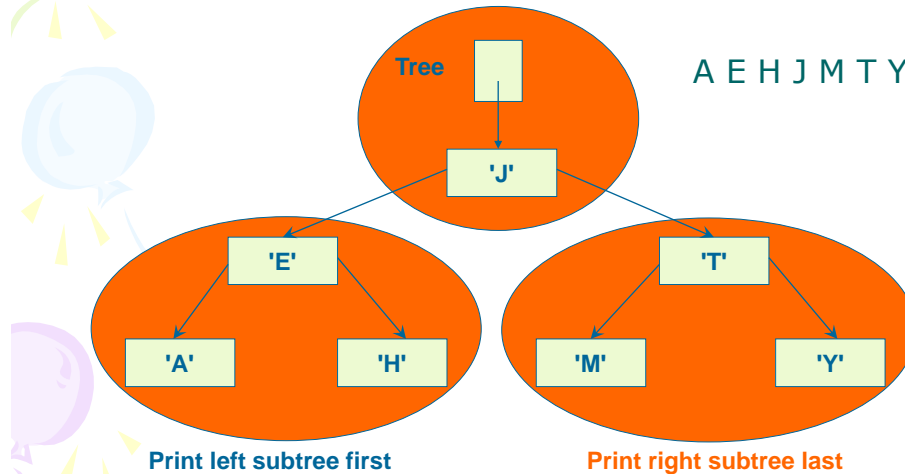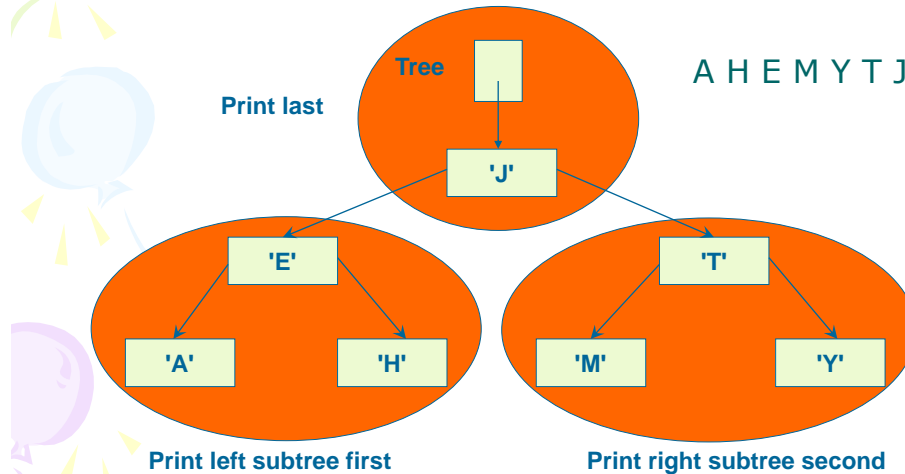
# Function inorderprint

```
void inorderprint(TreeType tree)
{
    if (tree!=NULL)
    {
        inorderprint(tree->left);
        printf("%4d\n",tree->Key);
        inorderprint(tree->right);
    }
}
```

# Postorder Traversal

- Visit the nodes in the left subtree, then visit the nodes in the right subtree, then visit the root of the tree

**Tree**

**Print last**

A H E M Y T J

'J'

'E'

'T'

'A'

'H'

'M'

'Y'

**Print left subtree first**

**Print right subtree second**
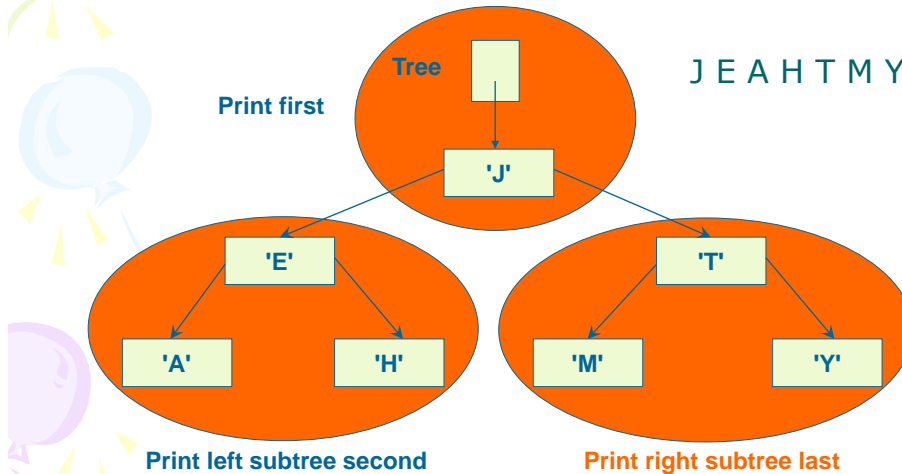
17

# Function postorderprint

```
void postorderprint(TreeType tree)
{
    if (tree!=NULL)
    {
        postorderprint(tree->left);
        postorderprint(tree->right);
        printf("%4d\n",tree->Key);
    }
}
```

18

# Preorder Traversal

- Visit the root of the tree first, then visit the nodes in the left subtree, then visit the nodes in the right subtree
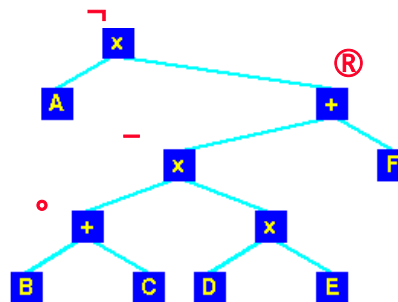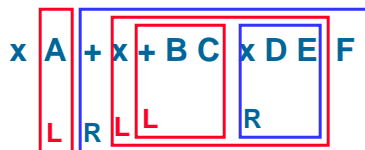
**Tree**

**Print first**

J E A H T M Y

'J'

'E'

'T'

'A'     'H'

'M'     'Y'

**Print left subtree second**          **Print right subtree last**

19

# Pre_order

☐ Pre-order
  – Root
  – Left sub-tree
  – Right sub-tree

x A + x + B C x D E F

L R L L    R

x

A

®

+

−

x

F

∘

+

x

B

C

D

E

20

# Function preorderprint

```
void preorderprint(TreeType tree)
{
   if (tree!=NULL)
   {
      printf("%4d\n",tree->Key);
      preorderprint(tree->left);
      preorderprint(tree->right);
   }
}
```
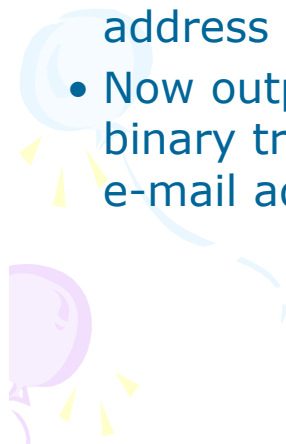
21

# Exercise

- Return to the exercise lastweek. We have already a tree for storing Phone address book.
- Now output all the data stored in the binary tree in ascending order for the e-mail address.

22

# Hint

- Just use the InOrderTraversal()

23

# Iterative Inorder Traversal

```
void iter_inorder(TreeType node)
{
  int top= -1; /* initialize stack */
  TreeType stack[MAX_STACK_SIZE];
  for (;;) {
   for (; node; node=node->left)
     add(&top, node);/* add to stack */
   node= delete(&top);/*delete from stack*/

   if (node==NULL) break;/* stack is empty */
   printf("%d", node->key);
   node = node->right;
  }
}
```
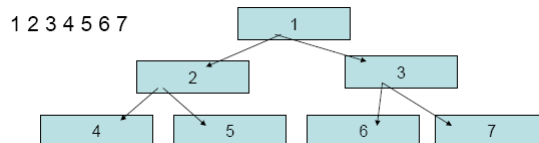
24

# Exercise

- Output all the data stored in the binary tree in ascending dictionnary order for the name in the Phone Book Tree:
  - to screen.
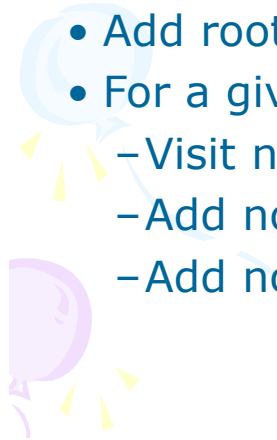  - to a file.

25

# Breadth First Search

- Instead of going down to children first, go across to siblings
- Visits all nodes on a given level in left-to-right order

1 2 3 4 5 6 7

```
            1
        2       3
    4     5   6     7
```
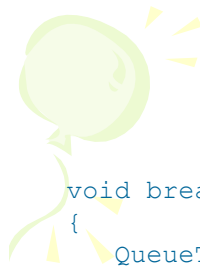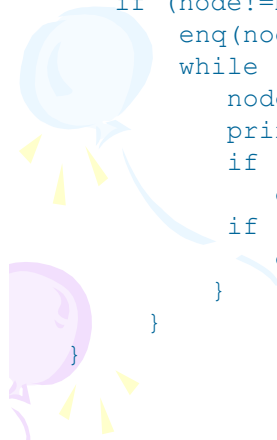
26

# Breadth First Search

- To handle breadth-first search, we need a queue in place of a stack
- Add root node to queue
- For a given node from the queue
  - Visit node
  - Add nodes left child to queue
  - Add nodes right child to queue

27

# Pseudo Algorithm

```
void breadth_first(TreeType node)
{
   QueueType queue; // queue of pointers
   if (node!=NULL) {
      enq(node,queue);
      while (!empty(queue)) {
         node=deq(queue);
         printf(node->key);
         if (node->left !=NULL)
            enq(node->left,queue);
         if (node->right !=NULL)
            enq(node->right,queue);
      }
   }
}
```

28

# Exercise

- Implement BFS algorithm in C language
- Add this function to the binary tree library
- Test it the Phone Book management program to print all the names in the tree.
- Output the results to a file

29

# Exercise

- Write a program to build a tournament: a binary tree where the item in every internal node is a copy of the larger of the items in its two children. So the root is a copy of largest item in the tournament. The items in the leaves constitute the data of interest.
- The input items are stored in an array.
- Hint: Uses a divide and conquer strategy



**48TH ANNUAL ACC TOURNAMENT**
MARCH 8-11, 2001 • THE GEORGIA DOME • ATLANTA, GA.
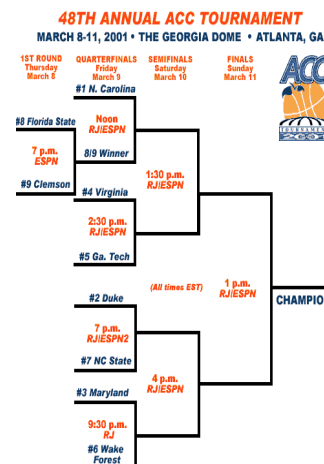
30

# Exercise: Calculate word frequencies

- Write to a program WordCount which reads a text file, then analyzes the word frequencies. The result is stored in a file. When user provide a word, program should return the number of occurrences of this word in the file.
- For example, suppose the input files has the following contents: *A black black cat saw a very small mouse and a very scared mouse.*
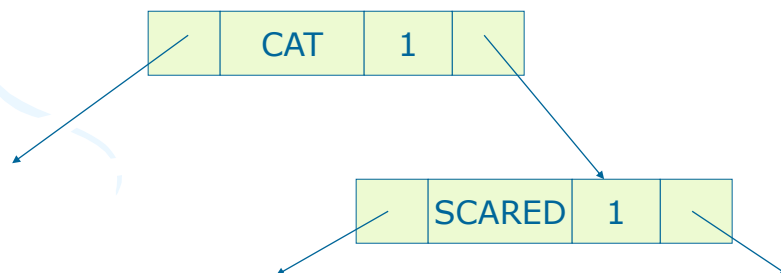- The word frequencies in this file are as follows:

| | |
|---|---|
| AND 1 | SMALL 1 |
| CAT 1 | BLACK 2 |
| SAW 1 | MOUSE 2 |
| SCARED 1 | VERY 2 |
| | A 3 |

# Hint

- Use a binary search tree (it's even better with AVL) to store data.
- A node in this tree should contain at least two fields:
  - word: string
  - count: int
- Words are stored in nodes in the dictionary order.