# Code Generation
# Last Phase of Compiler Construction

# Code Generation

- The code generation is the task of mapping intermediate code to machine code.

- **Requirements:**
  - Correctness
  - Must preserve semantic meaning of source program
  - Make effective use of available resources
  - Must run efficiently

# Input to the Code Generator

- We assume, front end has
  - Scanned, parsed and translate the source program into a reasonably detailed intermediate representations
  - Type checking, type conversion and obvious semantic errors have already been detected
  - Symbol table is able to provide run-time address of the data objects
- Intermediate representations may be
  - Postfix notations
  - Three address representations
  - Syntax tree
  - DAG

# TARGET PROGRAMS

- The output of the code generator is the target program.
- Target architecture:
  - Must be **well** understood
  - Significantly influences the difficulty of code generation
  - *Eg*. RISC, CISC
- Target program may be
  - Absolute machine language
    - It can be placed in a fixed location of memory and immediately executed
  - Re-locatable machine language
    - Subprograms to be compiled separately
    - A set of re-locatable object modules can be linked together and loaded for execution by a linker

# Issues in Design of Code Generator

- Instruction Selection
- Register Allocation
- Evaluation Order


5

# Instruction Selection

- There may be a *large number of 'candidate'* machine instructions for a given IR instruction
- Level of IR
  - High: Each IR translates into many machine instructions
  - Low: Reflects many low-level details of machine
- Nature of the instruction set
  - Uniformity and completeness
- Each has own cost and constraints
  - Accurate cost information is difficult to obtain
  - Cost may be influenced by surrounding context


6

# Instruction Selection

- For each type of three-address statement, *a code skeleton* can be designed that outlines the target code to be generated for that construct.

Say,

**x := y + z**

Mov y, R0

Add z, R0

Mov R0, x

**Statement by statement generation often produces poor code**

# Instruction Selection

a := b + c

d := a + e

MOV b, R0

ADD c, R0

MOV R0, a

MOV a, R0

ADD e, R0

MOV R0, d

If a is subsequently used

# Instruction Selection

*IR Code:* `x := x + 5`

*Target Code:*

```
mov   x,r0
add   5,r0
mov   r0,x
```

---

*IR Code:* `x := x + 1`

*Target Code:*

```
mov   x,r0
add   1,r0
mov   r0,x
```

*Target Code:*

```
mov   x,r0
inc   r0
mov   r0,x
```

*Target Code:*

```
inc   x
```

# Register Allocation

- How to best use the bounded number of registers.
- Use of registers
  - Register allocation
    - We select a set of variables that will reside in registers at each point in the program
  - Register assignment
    - We pick the specific register that a variable will reside in.
- Complications:
  - special purpose registers
  - operators requiring multiple registers.
- Optimal assignment is NP-complete


10

# Register Allocation

Multiply Instruction

```
mul    y,r4
```

Must specify an even numbered register

r4 × y → [r4,r5]

Multiply Instruction

```
div    y,r4
```

Must specify an even numbered register

[r4,r5] ÷ y ⇒ [r4,r5]

SRDA: Shift Right Double Arithmetic

```
srda   32,r6
```

| 1010110010 | xxxxxxxxxx | → | ssssssssss | 1010110010 |
|---|---|---|---|---|
| r6 | r7 | | r6 | r7 |

# Register Allocation

*IR Code:*

```
t := a + b
t := t * c
t := t / d
```

*Target Code:*

```
mov   a,r1
add   b,r1
mul   c,r0
div   d,r0
mov   r1,t
```

*IR Code:*

```
t := a + b
t := t + c
t := t / d
```

*Target Code:*

```
mov   a,r0
add   b,r0
add   c,r0
srda  32,r0
div   d,r0
mov   r1,t
```

**Conclusion:**

*Where you put the result of t:=a+b (either r0 or r1) depends on how it will be used later!!!*

[A "chicken-and-egg" problem]

# EVALUATION ORDER

- Choosing the order of instructions to best utilize resources
- Picking the optimal order is NP-complete problem

- Simplest Approach
  - Don't mess with re-ordering.
  - Target code will perform all operations in the same order as the IR code
- Trickier Approach
  - Consider re-ordering operations
  - May produce better code
  - ... Get operands into registers just before they are need
  - ... May use registers more efficiently

# Moving Results Back to Memory

- When to move results from registers back into memory?
- After an operation, the result will be in a register.
- **Immediately**
  - Move data back to memory just after it is computed.
  - May make more registers available for use elsewhere.
- ***Wait as long as possible* before moving it back**
  - Only move data back to memory "at the end"
  - or "when absolutely necessary"
  - May be able to avoid re-loading it later!

# CODE GENERATION ALGORITHM #1

Simple code generation algorithm:
Define a target code sequence to each intermediate code statement type.

## Example Target Machine

**2-Address Architecture**

source destination

```
mov     x,r0
add     y,r0
mov     r0,z
```

r0 := r0 + y

store back into memory

# CODE GENERATION ALGORITHM #1

Statement-by-statement generation
Code for each IR instruction is
generated independently of all other IR instructions.

*IR Code:*

```
a := b + c
d := a + e
```

*Target Code:*

```
...
mov   b,r0
add   c,r0        a := b + c
mov   r0,a
mov   a,r0
add   e,r0        d := a + e
mov   r0,d
...
```

*ALSO: Registers are not used effectively.*

*This instruction is totally unnecessary!!!*

# EXAMPLE TARGET MACHINE

**A 2-address Architecture**

```
op source,destination
```

2 operands, at most

**Op-Codes:**

```
mov
add
sub
mul
...
```

**Address Modes:**

*Absolute Memory Address*

```
mov    x,y
sub    x,y
```

x → y
y–x → y

*Register*

```
mov    r0,r1
sub    r2,r3
```

r3–r2 → r3

*Literal*

```
mov    39,r1
sub    47,r2
```

Data is included in the instruction directly

*Indirect Register*

```
mov    r0,[r1]
```

Register contains an address. Moves data in to word pointed to by r1

*Indirect plus Index*

```
mov    r0,[r1+48]
```

*Double Indirect*

```
mov    r0,[[r1+48]]
```

Use r1+48 as an address. Go to memory and fetch a second address, "p". "p" points to the word.

# EVALUATING A POTENTIAL CODE SEQUENCE

- Each instruction has a "*cost*"

  Cost = Execution Time
- Execution Time is difficult to predict.
  - Pipelining, Branches, Delay Slots, etc.
- **Goal:** Approximate the real cost

A "***Cost Model***"

**Simplest Cost Model:**

**Code Length ≈ Execution Time**

**Just count the instructions!**

# A Better Cost model

**Look at each instruction.**
**Compute a cost (in "units").**
**Count the number of memory accesses.**

**Cost = 1 + Cost-of-operand-1 + Cost-of-operand-2 + Cost-of-result**

| | example | cost |
|---|---|---|
| Absolute Memory Address | `x` | 1 |
| Register | `r0` | 0 |
| Literal | `39` | 0 |
| Indirect Register | `[r1]` | 1 |
| Indirect plus Index | `[r1+48]` | 1 |
| Double Indirect | `[[r1+48]]` | 2 |

<u>Example:</u> `sub 97,r5` r5 – 97 → r5
Cost = 1 + 0 + 0 + 0 = 1

<u>Example:</u> `sub 97,[r5]` [r5] – 97 → [r5]
Cost = 1 + 1 + 0 + 1 = 3

<u>Example:</u> `sub [r1],[[r5+48]]` [[r5+48]] – [r1] → [[r5+48]]
Cost = 1 + 2 + 1 + 2 = 6

# Cost Generation Example

**IR Code:** `x := y + z`

**Translation #1:**

```
mov   y,x   3
add   z,x   4
```

} Cost = 7

**Translation #2:**

```
mov   y,r1  2
add   z,r1  2
mov   r1,x  2
```

} Cost = 6

***Lesson #1:***
*Use Registers*

**Translation #3:**

*Assume "y" is in r1 and "z" is in r2*
*Assume "y" will not be needed again*

```
add   r2,r1  1
mov   r1,x   2
```

} Cost = 3

***Lesson #2:***
*Keep variables in registers*

***Lesson #3:***
*Avoid or delay storing into memory.*

**Translation #4:**

*Assume "y" is in r1 and "z" is in r2*
*Assume "y" will not be needed again.*
*Assume we can keep "x" in a register.*

```
add   r2,r1  1
```

} Cost = 1

***Lesson #4:*** *(not illustrated)*
*Use different addressing modes effectively.*

# Basic Blocks

**Break IR code into blocks such that...**

**The block contains NO transfer-of-control instructions**

**... except as the last instruction**

- A sequence of consecutive statements.
- Control enters only at the beginning.
- Control leaves only at the end.

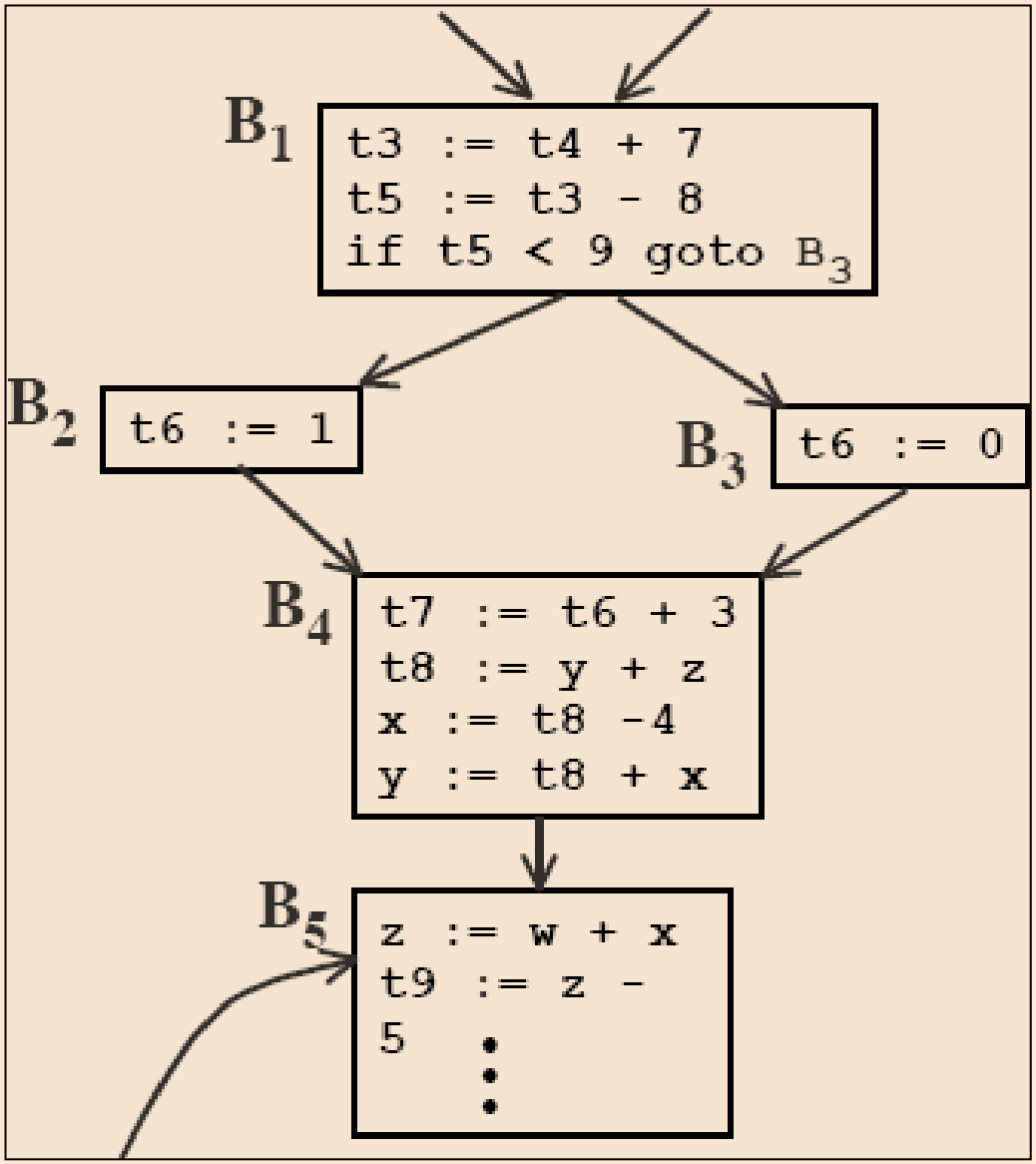
# Basic Blocks

```
                  ⋮
Label_43:     t3 := t4 + 7
              t5 := t3 - 8
              if t5 < 9 goto Label_44
              t6 := 1
              goto Label_45
Label_44:     t6 := 0
Label_45:     t7 := t6 + 3
              t8 := y + z
              x := t8 -4
              y := t8 + x
Label_46:     z := w + x
              t9 := z - 5
                  ⋮
```

# Basic Blocks

```
                 ⋮
Label_43:    t3 := t4 + 7
             t5 := t3 - 8                    B1
             if t5 < 9 goto Label_44
             t6 := 1                         B2
             goto Label_45
Label_44:    t6 := 0                         B3
Label_45:    t7 := t6 + 3
             t8 := y + z                     B4
             x := t8 -4
             y := t8 + x
Label_46:    z := w + x                      B5
             t9 := z - 5
                 ⋮
```

# CONTROL FLOW GRAPH



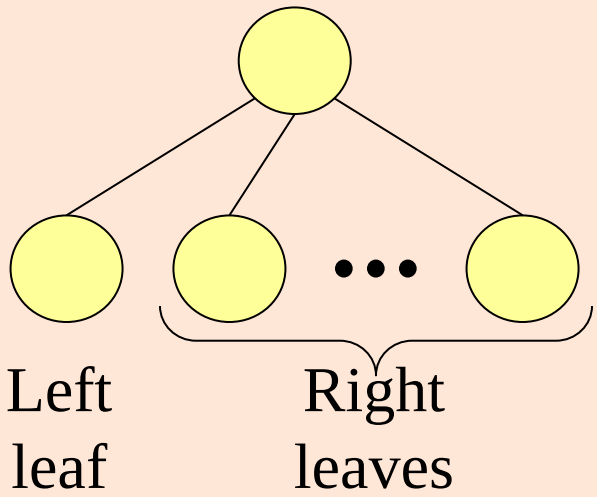
# Sethi-Ullman Algorithm
## Identifying no. of registers required

**Intuition:**

1. Label each node according to the number of registers that are required to generate code for the node.

2. Generate code from top down always generating code first for the child that requires the most registers.

# SETHI-ULLMAN ALGORITHM (INTUITION)




Bottom-Up Labeling: visit a node after all its children are labeled.

# LABELING ALGORITHM

(1) **if** $n$ is a leaf **then**

(2) **if** $n$ is the leftmost child of its parent **then**

(3) $label(n) := 1$

(4) **else** $label(n) := 0$

**else begin** /* $n$ is an interior node */

(5) let $c_1, c_2, \cdots, c_k$ be the children of $n$ ordered by *label*

so that $label(c_1) \geq label(c_2) \geq \cdots \geq label(c_k)$

(6) $label(n) := \max_{1 \leq i \leq k} (label(c_i) + i - 1)$

**end**

# LABELING ALGORITHM

$$label(c_1) \geq label(c_2) \geq \cdots \geq label(c_k)$$

If $k = 1$ (a node with two children), then the following relation

$$label(n_1) := \max_{1 \leq i \leq k} (label(c_i) + i - 1)$$

becomes:

$$label(n) = \begin{cases} \max[label(c_1), label(c_2)] & \text{if } label(c_1) \neq label(c_2) \\ label(c_1) + 1 & \text{if } label(c_1) = label(c_2) \end{cases}$$



28

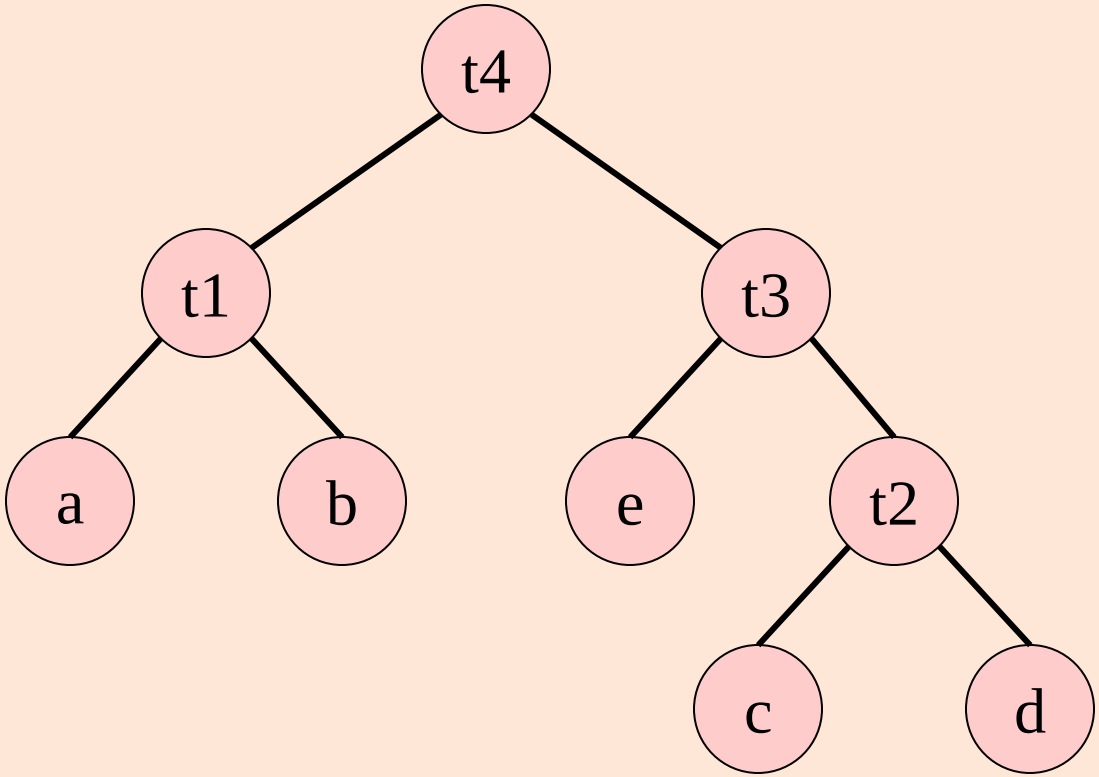
# EXAMPLE

- Let's assume we have following set of instructions:

$$t_1=a+b$$
$$t_2=c+d$$
$$t_3=e+t_2$$
$$t_4=t_1+t_3$$

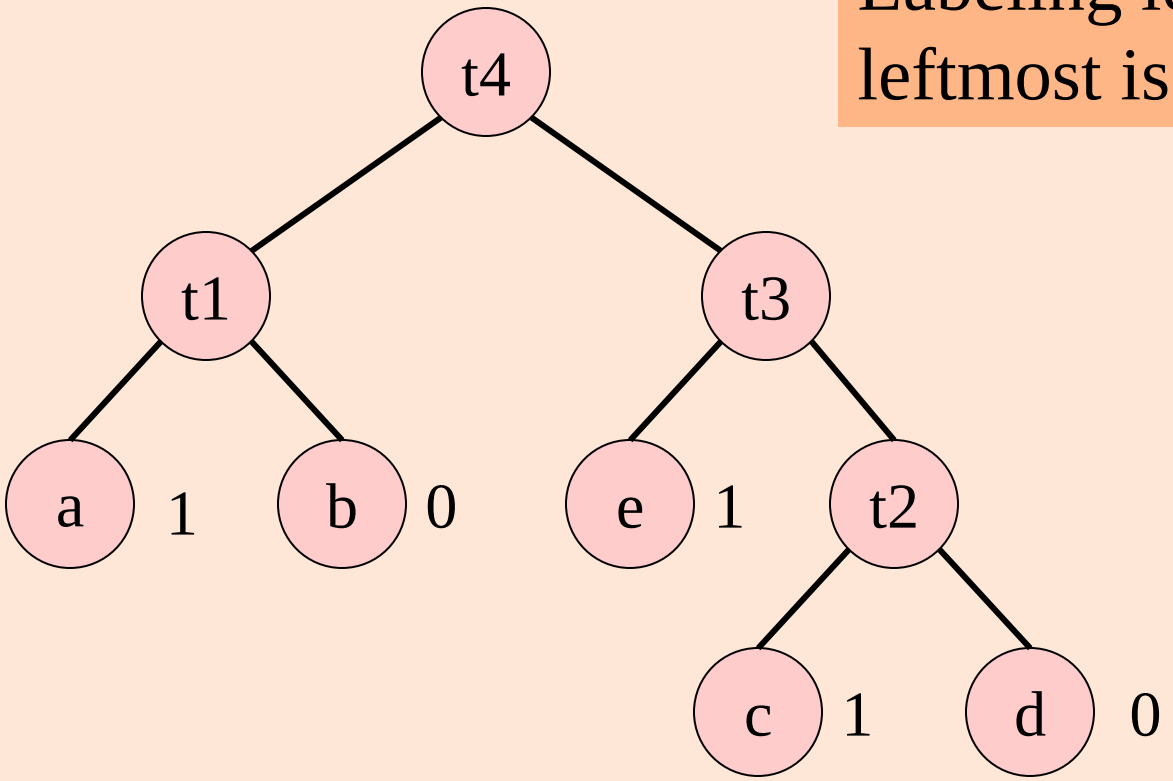
- Draw the tree corresponding to the given instructions

# EXAMPLE

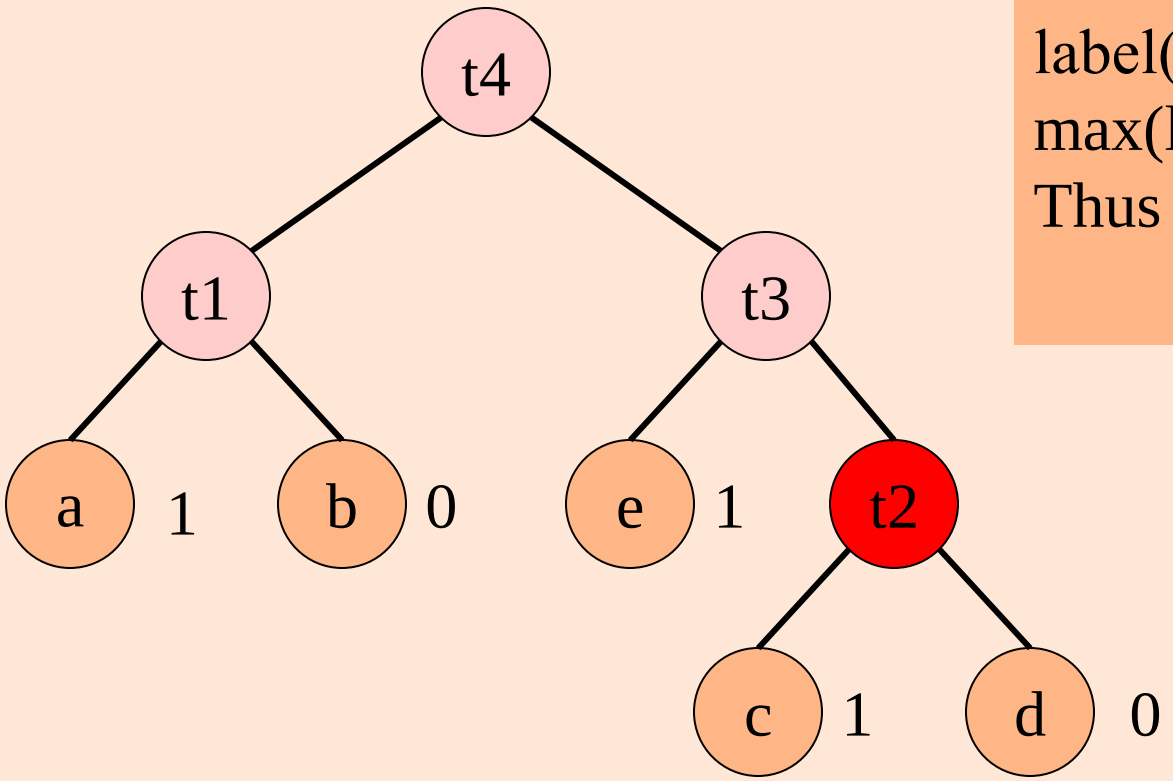


# EXAMPLE

Labeling leaves:
leftmost is 1, others are 0



# EXAMPLE



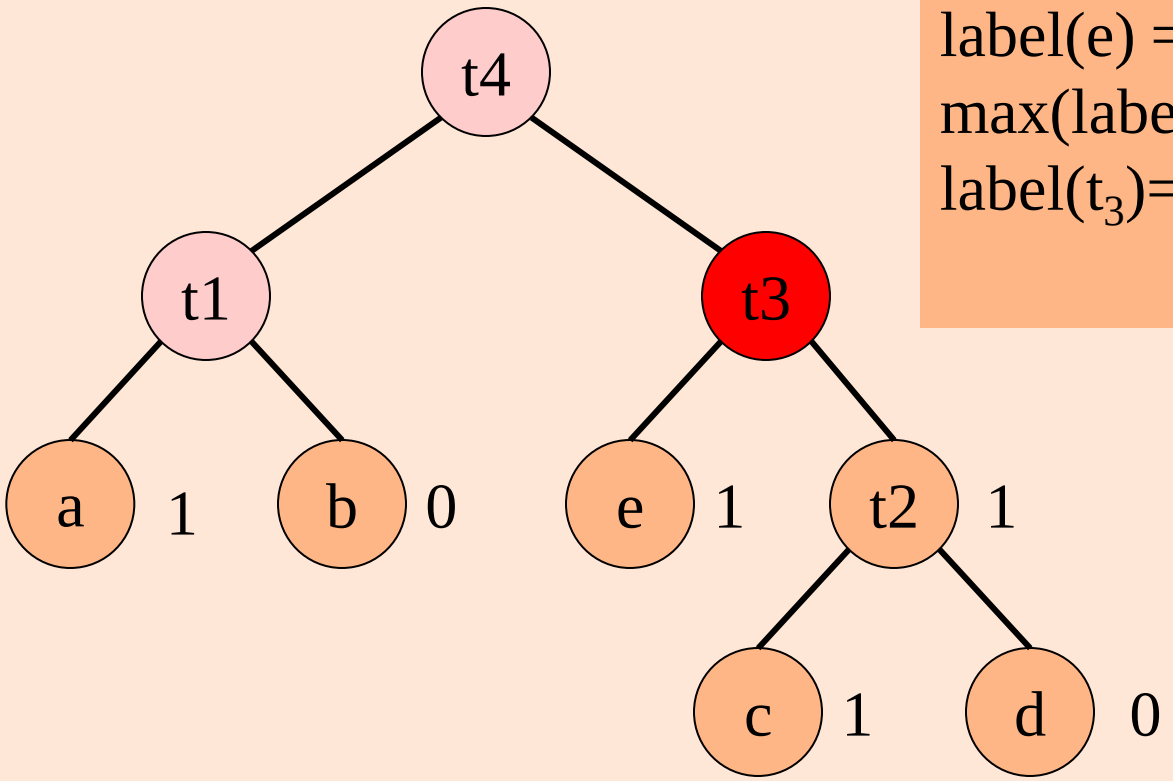

Labeling $t_2$:

label(c) ≠ label (d)

max(label(1),label(2))

Thus label($t_2$)=1

# Example



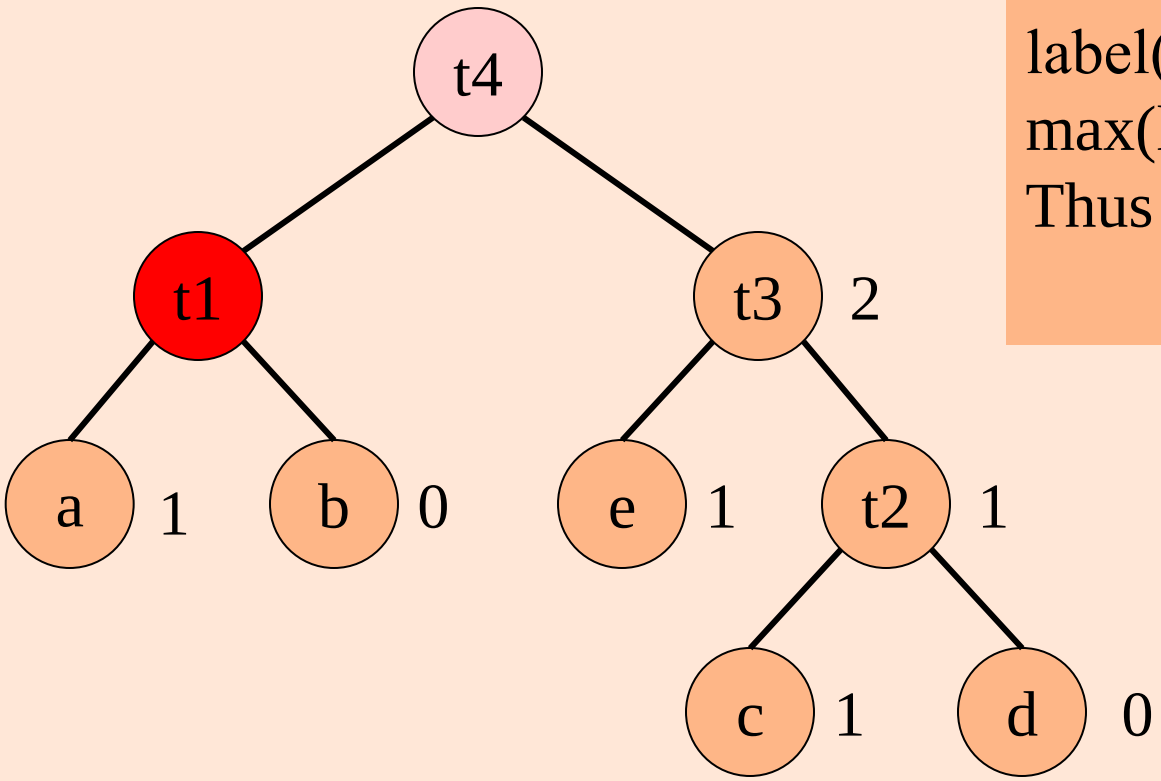

Labeling $t_3$:

label(e) = label ($t_2$)

max(label(1),label(2))

label($t_3$)=label(1)+1 =2

# Example



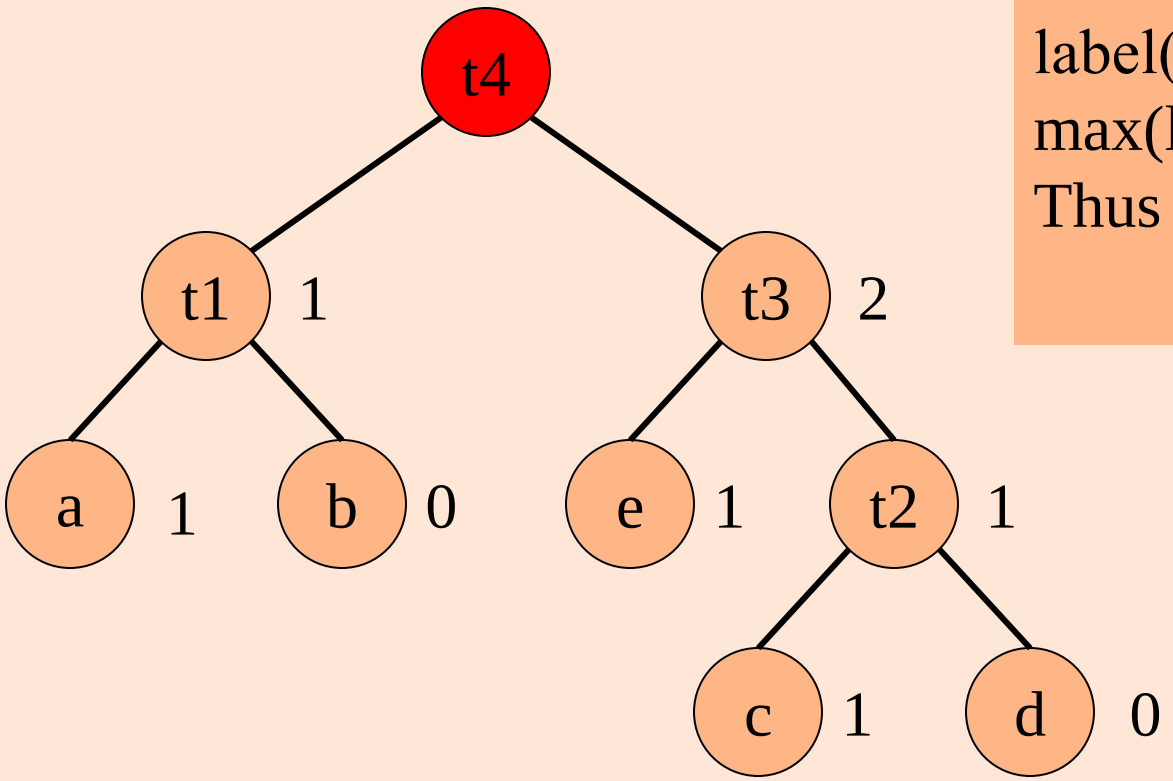

Labeling $t_1$:

label(a) ≠ label (b)

max(label(1),label(2))

Thus label($t_1$)=1

# Example



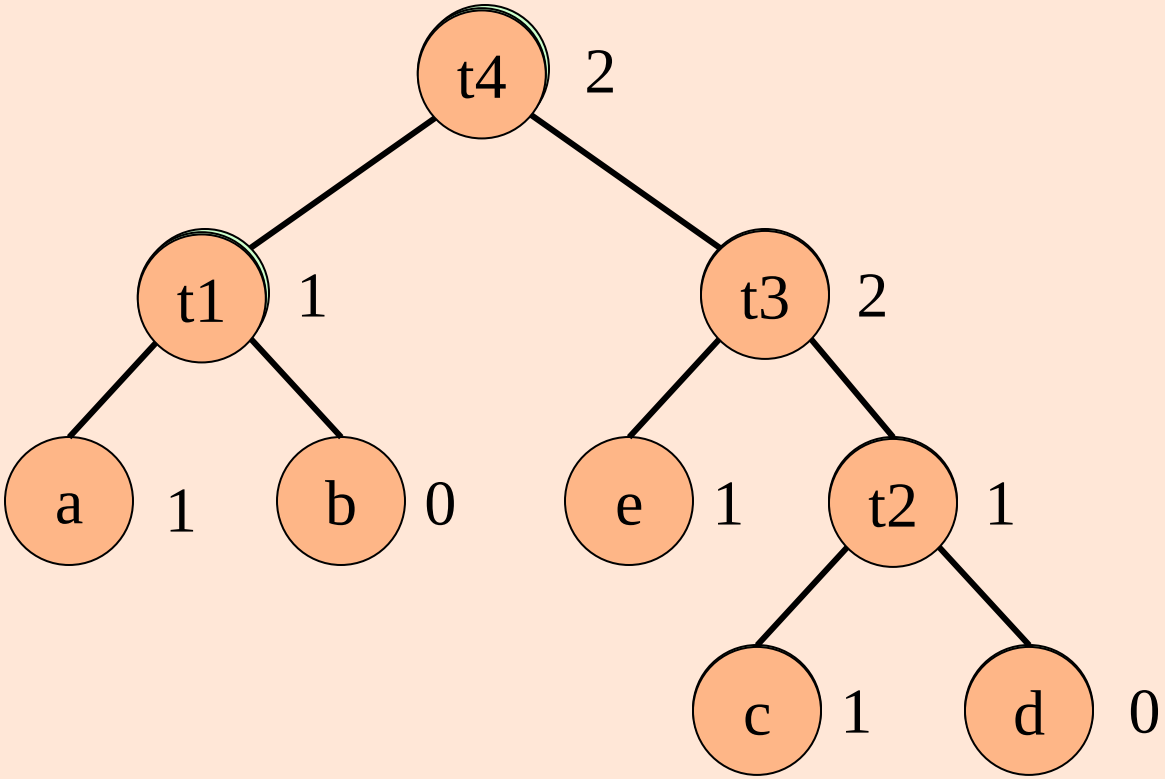

Labeling $t_4$:

label(a) ≠ label (b)

max(label(1),label(2))

Thus label($t_4$)=2

# Example



# Thanks