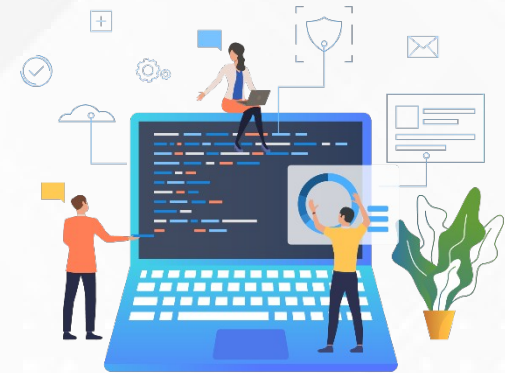




Unit – 8

Code Generation



Prof. Dixita B. Kagathara

Computer Engineering
Department
Darshan Institute of Engineering & Technology, Rajkot

✉ dixita.kagathara@darshan.ac.in
☎ +91 - 97277 47317 (CE
Department)





Topics to be covered

- Issues in code generation
- Target machine
- Basic block and flow graph
- Transformation on basic block
- Next use information
- Register allocation and assignment
- DAG representation of basic block
- Generation of code from DAG



Issues in Code Generation

Issues in Code Generation

► Issues in Code Generation are:

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Choice of evaluation
7. Approaches to code generation

Input to code generator

- ▶ Input to the code generator consists of the intermediate representation of the source program.
- ▶ Types of intermediate language are:
 1. Postfix notation
 2. Quadruples
 3. Syntax trees or DAGs
- ▶ The detection of semantic error should be done before submitting the input to the code generator.
- ▶ The code generation phase requires complete error free intermediate code as an input.

Target program

► The output may be in form of:

1. **Absolute machine language:** Absolute machine language program can be placed in a memory location and immediately execute.
2. **Relocatable machine language:** The subroutine can be compiled separately. A set of relocatable object modules can be linked together and loaded for execution.
3. **Assembly language:** Producing an assembly language program as output makes the process of code generation easier, then assembler is require to convert code in binary form.

Memory management

- ▶ **Mapping names** in the source program **to addresses of data objects** in run time memory is done cooperatively by the front end and the code generator.
- ▶ We assume that a name in a three-address statement refers to a symbol table entry for the name.
- ▶ From the symbol table information, a relative address can be determined for the name in a data area.

Instruction selection

- ▶ Example: the sequence of statements

$a := b + c$

$d := a + e$

- ▶ would be translated into

MOV b, R0

ADD c, R0

MOV R0, a

MOV a, R0

ADD e, R0

MOV R0, d

- ▶ Here the fourth statement is redundant, so we can eliminate that statement.

Register allocation

- ▶ The use of registers is often subdivided into two sub problems:
- ▶ During **register allocation**, we select the **set of variables** that will reside in registers at a point in the program.
- ▶ During a subsequent **register assignment** phase, we pick the **specific register** that a variable will reside in.
- ▶ Finding an optimal assignment of registers to variables is difficult, even with single register value.
- ▶ Mathematically the problem is **NP-complete**.

Choice of evaluation

- ▶ The **order in which computations are performed** can affect the efficiency of the target code.
- ▶ Some computation orders require fewer registers to hold intermediate results than others.
- ▶ Picking a best order is another difficult, **NP-complete problem**.

Approaches to code generation

- ▶ The most important criterion for a code generator is that it produces correct code.
- ▶ The design of code generator should be in such a way so it can be implemented, tested, and maintained easily.



Target Machine



Target machine

- ▶ We will assume our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps.
- ▶ The underlying computer is a byte-addressable machine with general-purpose registers,
- ▶ The two address instruction of the form: *op source, destination*
- ▶ It has following opcodes:
 - MOV (move source to destination)
 - ADD (add source to destination)
 - SUB (subtract source to destination)

Instruction Cost

- ▶ The address modes together with the assembly language forms and associated cost as follows:

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k + contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

- ▶ The instruction cost can be computed as one plus cost associated with the source and destination addressing modes given by “extra cost”.

Instruction Cost

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k + contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

- Calculate cost for following:

```
MOV B,R0
ADD C,R0
MOV R0,A
```

$$1 + 1 + 0 = 2$$

$$1 + 0 + 1 = 2$$

Total Cost=6

Instruction Cost

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k + contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

- Calculate cost for following:

MOV *R1 ,*R0 MOV *R1 ,*R0	MOV *R1 ,*R0 □ cost =
	Total Cost=2



Basic Block & Flow Graph

Basic Blocks

- ▶ A basic block is a **sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end** without halt or possibility of branching except at the end.
- ▶ The following sequence of three-address statements forms a basic block:

t1 := a*a

t2 := a*b

t3 := 2*t2

t4 := t1+t3

t5 := b*b

t6 := t4+t5

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements.

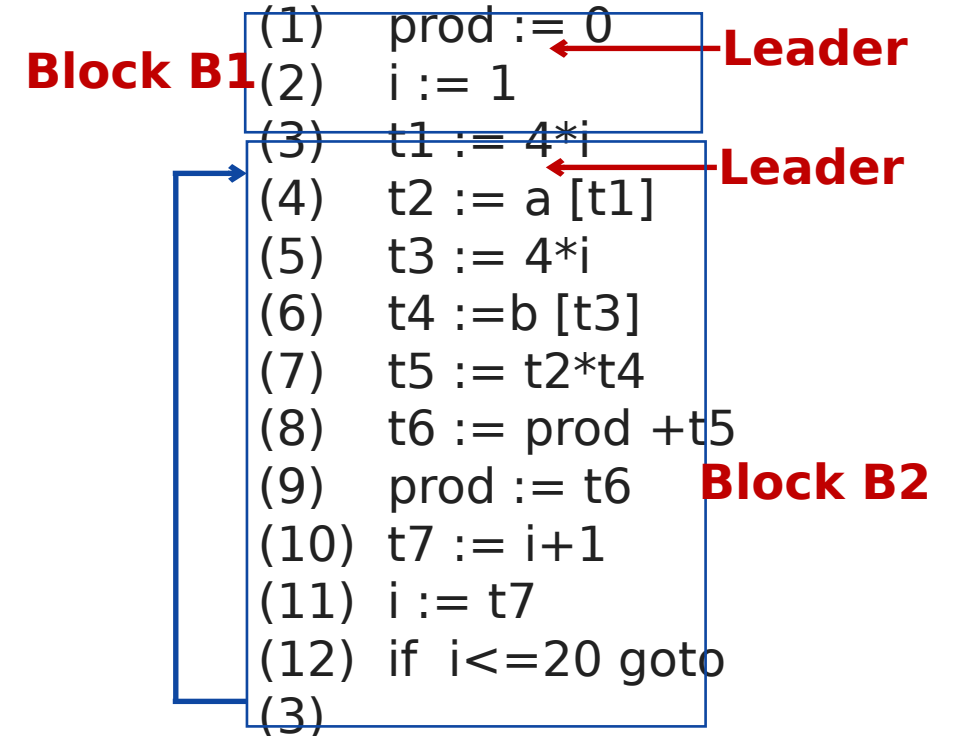
Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

1. We first determine the set of **leaders**, for that we use the following rules:
 - I. The first statement is a leader.
 - II. Any statement that is the target of a conditional or unconditional goto is a leader.
 - III. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example: Partition into basic blocks

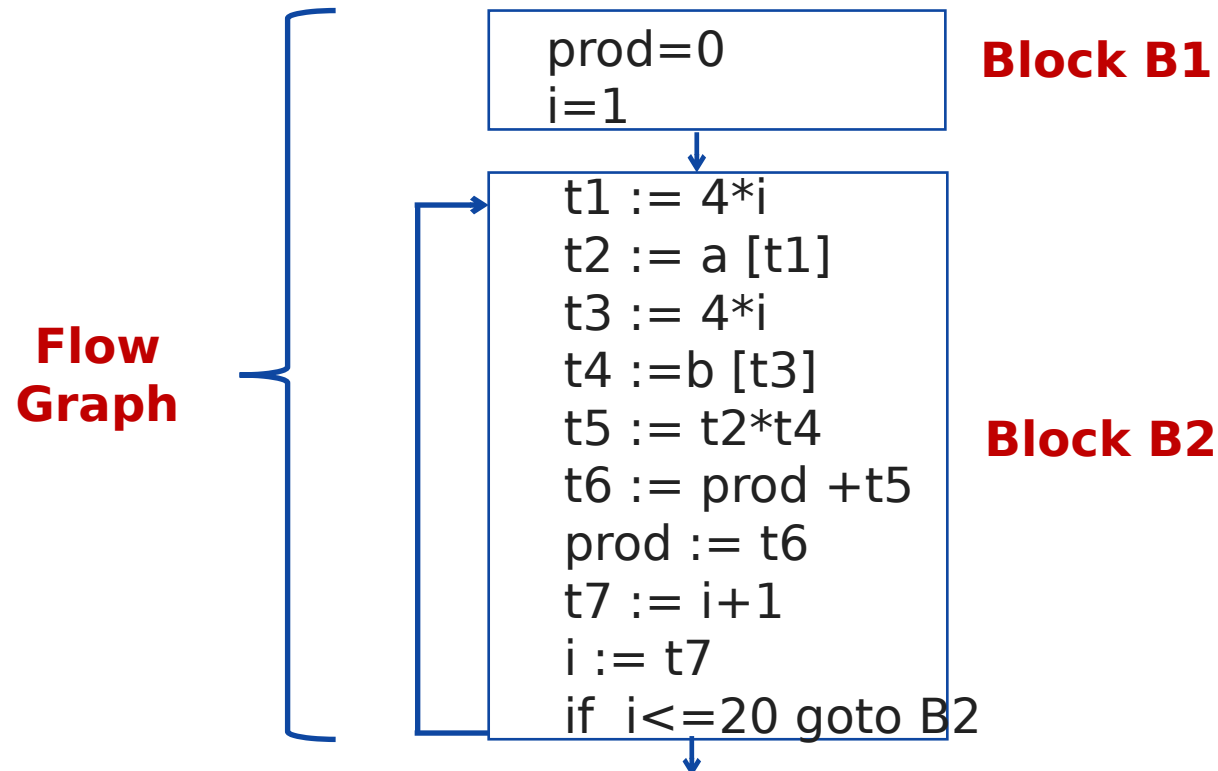
```
begin
    prod := 0;
        i := 1;
    do
        prod := prod + a[t1] *
b[t2];
        i := i+1;
    while i <= 20
end
```



**Three Address
Code**

Flow Graph

- ▶ We can add flow-of-control information to the set of basic blocks making up a program by constructing a direct graph called a **flow graph**.
- ▶ Nodes in the flow graph represent computations, and the edges represent the flow of control.
- ▶ Example of flow graph for following three address code:





Transformation on Basic Blocks

Transformation on Basic Blocks

- ▶ A number of transformations can be applied to a basic block without changing the set of expressions computed by the block.
- ▶ Many of these **transformations** are useful for **improving the quality of the code**.
- ▶ Types of transformations are:
 1. Structure preserving transformation
 2. Algebraic transformation

Structure Preserving Transformations

- ▶ Structure-preserving transformations on basic blocks are:
 1. Common sub-expression elimination
 2. Dead-code elimination
 3. Renaming of temporary variables
 4. Interchange of two independent adjacent statements

Common sub-expression elimination

- ▶ Consider the basic block,

a := b + c

b := a - d

c := b + c

d := a - d

- ▶ The second and fourth statements compute the same expression, hence this basic block may be transformed into the equivalent block:

a := b + c

b := a - d

c := b + c

d := b

Dead-code elimination

- ▶ Suppose s dead, that is, never subsequently used, at the point where the statement appears in a basic block.
- ▶ Above statement may be safely removed without changing the value of the basic block.

Renaming of temporary variables

- ▶ Suppose we have a statement
 $t := b + c$, where t is a temporary variable.
- ▶ If we change this statement to
 $u := b + c$, where u is a new temporary variable,
- ▶ Change all uses of this instance of t to u , then the value of the basic block is not changed.
- ▶ In fact, we can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary.
- ▶ We call such a basic block a *normal-form* block.

Interchange of two independent adjacent statements

- ▶ Suppose we have a block with the two adjacent statements,
 $t1 := b + c$
 $t2 := x + y$
- ▶ Then we can interchange the two statements without affecting the value of the block if and only if neither $t1$ is $t2$ and neither $t2$ is $t1$.
- ▶ A normal-form basic block permits all statement interchanges that are possible.

Algebraic Transformation

- ▶ Countless algebraic transformation can be used to change the set of expressions computed by the basic block into an algebraically equivalent set.
- ▶ The useful ones are those that **simplify expressions or replace expensive operations by cheaper one**.
- ▶ Example: **$x = x + 0$ or $x = x * 1$** can be eliminated.

Next Use Information


Computing Next Uses

- ▶ The next-use information is a collection of all the **names that are useful for next subsequent statement in a block.**
- ▶ The **use of a name** is defined as follows,
- ▶ Consider a statement,
$$x := i$$
$$j := x \text{ op } y$$
- ▶ That means the **statement j uses value of x.**
- ▶ The next-use information can be collected by making the backward scan of the programming code in that specific block.

Storage for Temporary Names

- ▶ For the distinct names each time a temporary is needed. And each time a space gets allocated for each temporary.
- ▶ To have optimization in the process of code generation we **pack two temporaries into the same location if they are not live simultaneously.**
- ▶ Consider three address code as,

```
t1=a*a  
t2=a*b  
t3=4*t2  
t4=t1+t3  
t5=b*b  
t6=t4+t5
```



Register and Address Descriptors

- ▶ The code generator algorithm uses descriptors to keep track of register contents and addresses for names.
- ▶ **Address descriptor** stores the location where the current value of the **name** can be found at run time. The information about locations can be stored in the symbol table and is used to access the variables.
- ▶ **Register descriptor** is used to keep track of **what is currently in each register**. The register descriptor shows that initially all the registers are empty. As the generation for the block progresses the registers will hold the values of computation.



Register Allocation & Assignment

Register Allocation & Assignment

- ▶ Efficient utilization of registers is important in generating good code.
- ▶ There are four strategies for deciding what values in a program should reside in a registers and which register each value should reside.
- ▶ Strategies are:
 1. Global Register Allocation
 2. Usage Count
 3. Register assignment for outer loop
 4. Register allocation for graph coloring

Global Register Allocation

- ▶ Global register allocation strategies are:
- ▶ The global register allocation has a strategy of **storing the most frequently used variables** in fixed registers throughout the **loop**.
- ▶ Another strategy is to assign some fixed number of global registers to hold the **most active values in each inner loop**.
- ▶ The registers not already allocated may be used to hold values local to one block.
- ▶ In certain languages like **C or Bliss** programmer can do the **register allocation by using register declaration** to keep certain values in register for the duration of the procedure.
- ▶ Example:

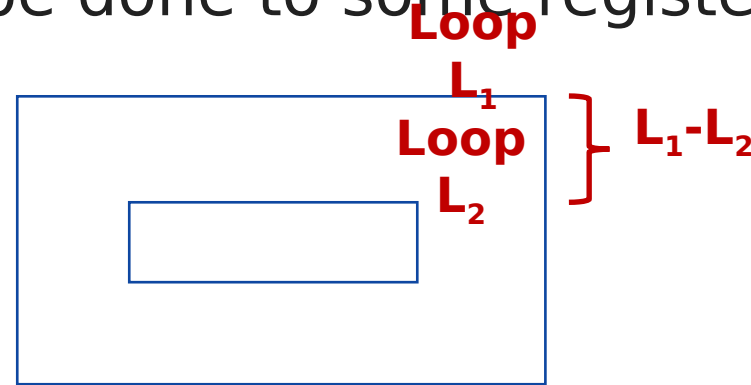
```
{  
    register int x;  
}
```

Usage count

- ▶ The usage count is the count for the use of some variable x in some register used in any basic block.
- ▶ The **usage count gives** the idea about **how many units of cost can be saved** by selecting a specific variable for global register allocation.
- ▶ The approximate formula for usage count for the Loop in some basic block can be given as,
- ▶ Where U_x is number of times x used in block prior to any definition of x
- ▶ if x is live on exit from block; otherwise 0.

Register assignment for outer loop

- Consider that there are two loops L_1 is outer loop and L_2 is an inner loop, and allocation of variable a is to be done to some register.



- Following criteria should be adopted for register assignment for outer loop,
- If a is allocated in L_2 then it should not be allocated in L_1 .
- If a is allocated in L_1 and it is not allocated in L_2 then store a on entrance to L_2 and load a while leaving L_2 .
- If a is allocated in L_1 and not in L_2 then load a on entrance of L_1 and store a on exit from L_1 .

Register allocation for graph coloring

- ▶ The graph coloring works in two passes. The working is as given below:
- ▶ In the first pass the specific machine instruction is selected for register allocation. For each variable a symbolic register is allocated.
- ▶ In the second pass the register inference graph is prepared.
- ▶ In register inference graph each node is a symbolic registers and an edge connects two nodes where one is live at a point where other is defined.
- ▶ Then a graph coloring technique is applied for this register inference graph using k-color.
- ▶ The k-colors can be assumed to be number of assignable registers.
- ▶ In graph coloring technique no two adjacent nodes can have same color. Hence in register inference graph using such graph coloring principle each node is assigned the symbolic registers so that no two symbolic registers can interfere with each other with assigned physical registers.



DAG Representation of Basic Block

Algorithm: DAG Construction

We assume the three address statement could of following types:

Case (i) $x := y \text{ op } z$

Case (ii) $x := \text{op } y$

Case (iii) $x := y$

With the help of following steps the DAG can be constructed.

► **Step 1:** If y is undefined then create $\text{node}(y)$. Similarly if z is undefined create a node (z)

► **Step 2:**

Case(i) create a $\text{node}(\text{op})$ whose left child is $\text{node}(y)$ and $\text{node}(z)$ will be the right child. Also check for any common sub expressions.

Case (ii) determine whether is a node labeled op , such node will have a child $\text{node}(y)$.

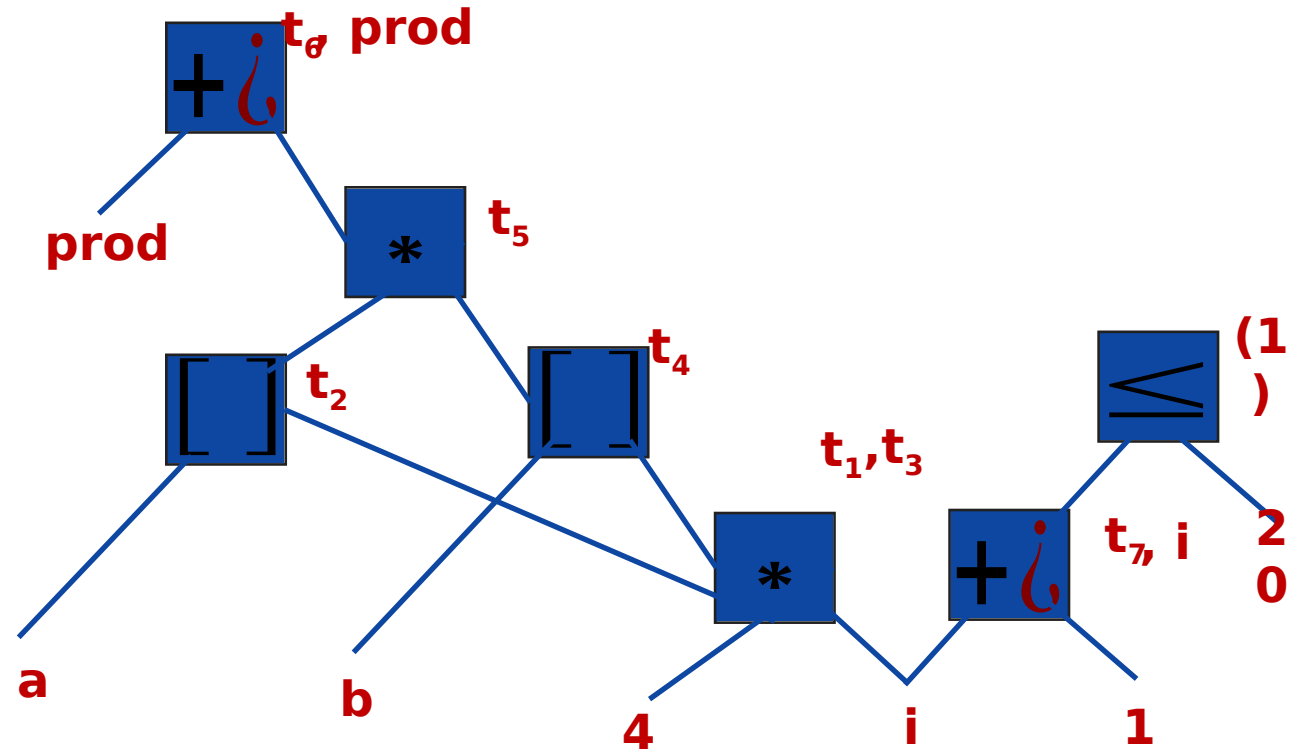
Case (iii) node n will be $\text{node}(y)$.

► **Step 3:** Delete x from list of identifiers for $\text{node}(x)$. Append x to the list of attached identifiers for node n found in 2.

DAG Representation of Basic Block

Example:

```
(1) t1 := 4*i
(2) t2 := a[t1]
(3) t3 := 4*i
(4) t4 := b[t3]
(5) t5 := t2*t4
(6) t6 := prod
    +t5
(7) prod := t6
(8) t7 := i+1
(9) i := t7
(10) if i<=20
goto (1)
```



Applications of DAG

- ▶ The DAGs are used in following:
 1. Determining the **common sub-expressions**.
 2. Determining which **names** are used inside the block and computed **outside the block**.
 3. Determining which statements of the **block** could have their computed **value outside the block**.
 4. Simplifying the list of quadruples by **eliminating the common sub-expressions and not performing the assignment of the form $x:=y$ unless and until it is a must**.



Generation of Code from DAGs

Generation of Code from DAGs

- Methods generating code from DAGs are:
 1. Rearranging Order
 2. Heuristic ordering
 3. Labeling algorithm

Rearranging Order

- ▶ The order of three address code affects the cost of the object code being generated.
- ▶ By changing the order in which computations are done we can obtain the object code with minimum cost.
- ▶ Example:

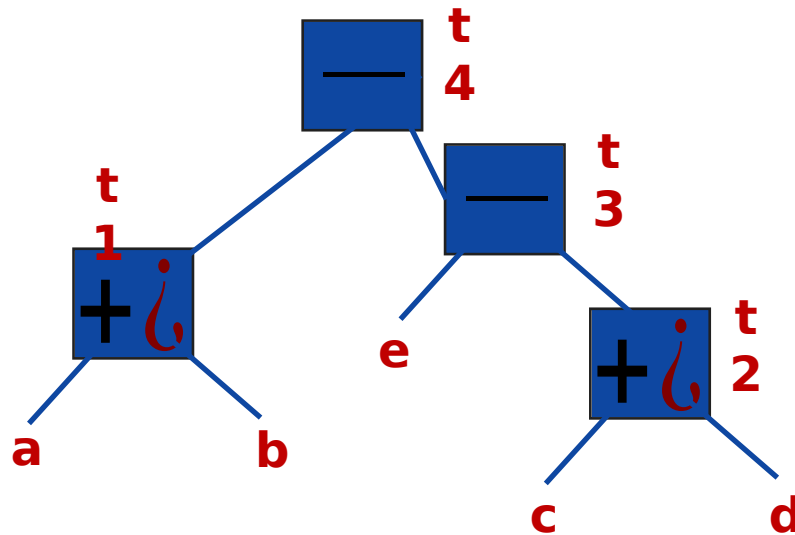
$t1 := a + b$

$t2 := c + d$

$t3 := e - t2$

$t4 := t1 - t3$

Three Address
Code



Example: Rearranging Order

t1:=a+b

t2:=c+d

t3:=e-t2

t4:=t1-t3

**Three Address
Code**

Re-arrange

t2:=c+d

t3:=e-t2

t1:=a+b

t4:=t1-t3

**Three Address
Code**

MOV a, R0

ADD b, R0

MOV c, R1

ADD d, R1

MOV R0, t1

MOV e, R0

SUB R1, R0

MOV t1, R1

SUB R0, R1

MOV R1, t4

Assembly Code

MOV c, R0

ADD d, R0

MOV e, R1

SUB R0, R1

MOV a, R0

ADD b, R0

SUB R1, R0

MOV R0, t4

Assembly Code



Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

Algorithm: Heuristic Ordering

Obtain all the interior nodes. Consider these interior nodes as unlisted nodes.

while(unlisted interior nodes remain)

{

pick up an unlisted node n , whose parents have been listed

list n ;

while(the leftmost child m of n has no unlisted parent AND is not leaf)

{

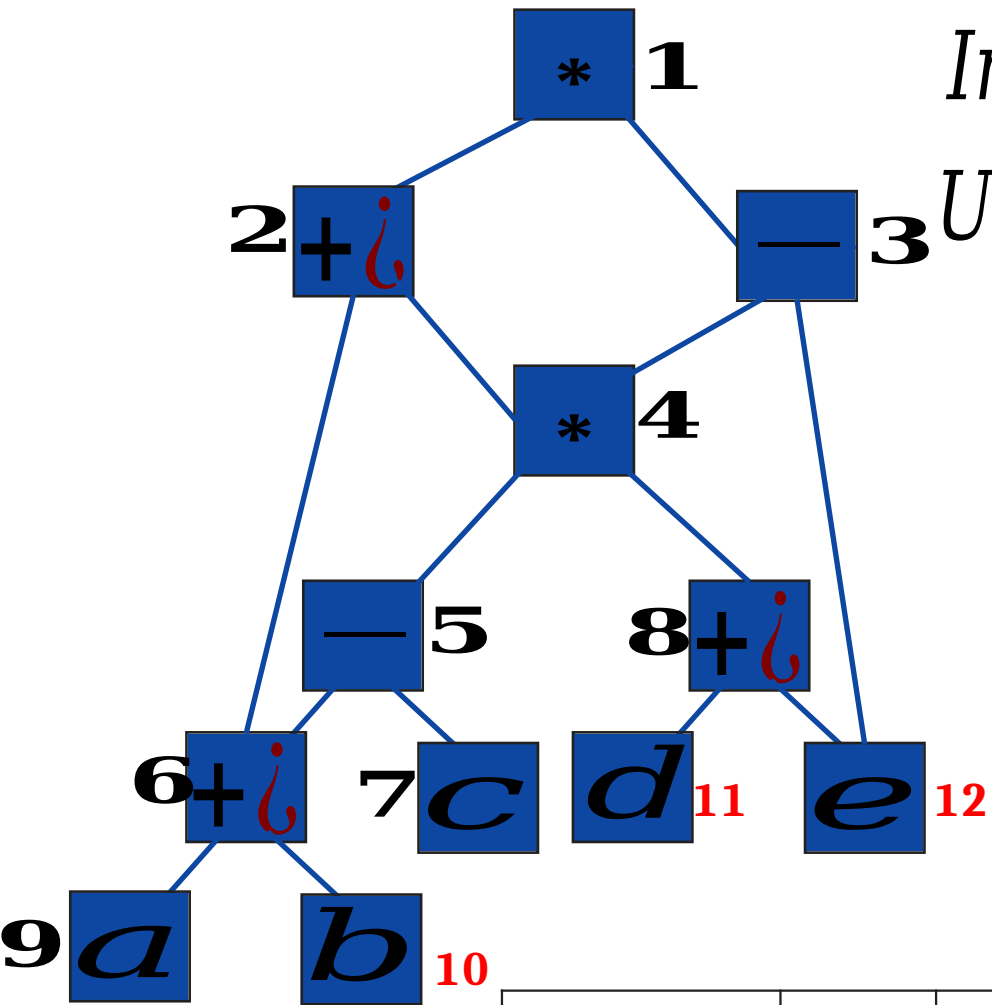
List m ;

$n=m$;

}

}

Example: Heuristic Ordering



Interior nodes=1234568

Unlisted nodes=~~1~~234568

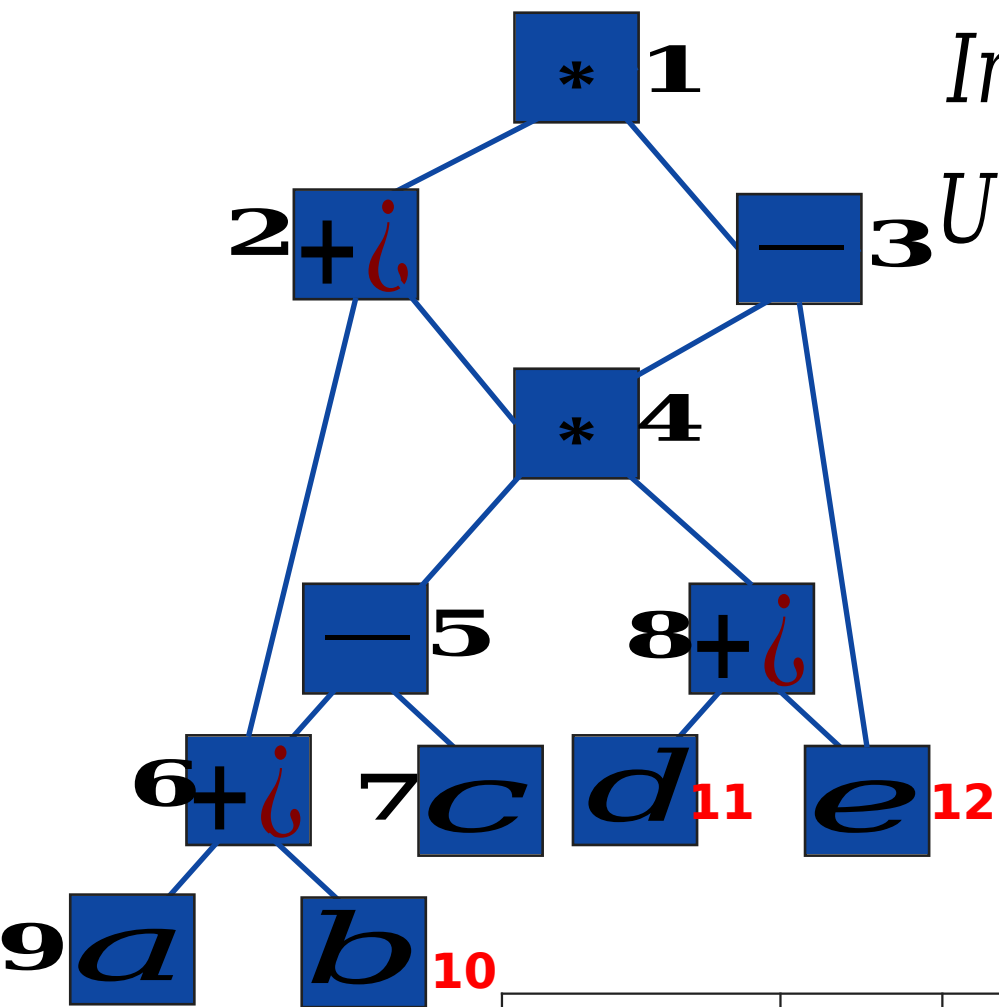
Pick up an unlisted node,
whose parents have been
listed

1

Left child of 2
Parents not listed so
can't list

Listed Node							
-------------	--	--	--	--	--	--	--

Example: Heuristic Ordering



Interior nodes=1234568

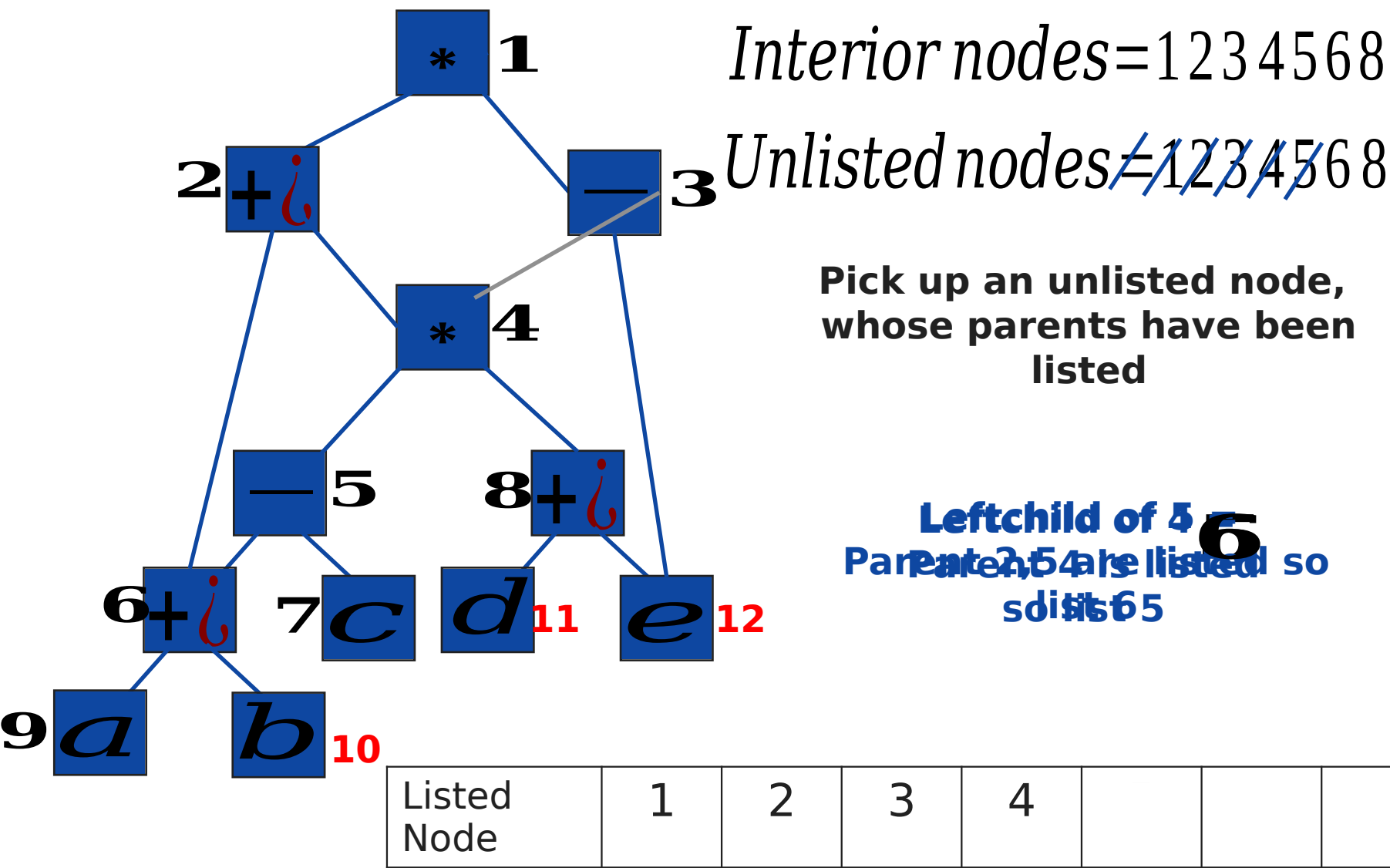
Unlisted nodes=~~1~~~~2~~~~3~~4568

Pick up an unlisted node,
whose parents have been
listed

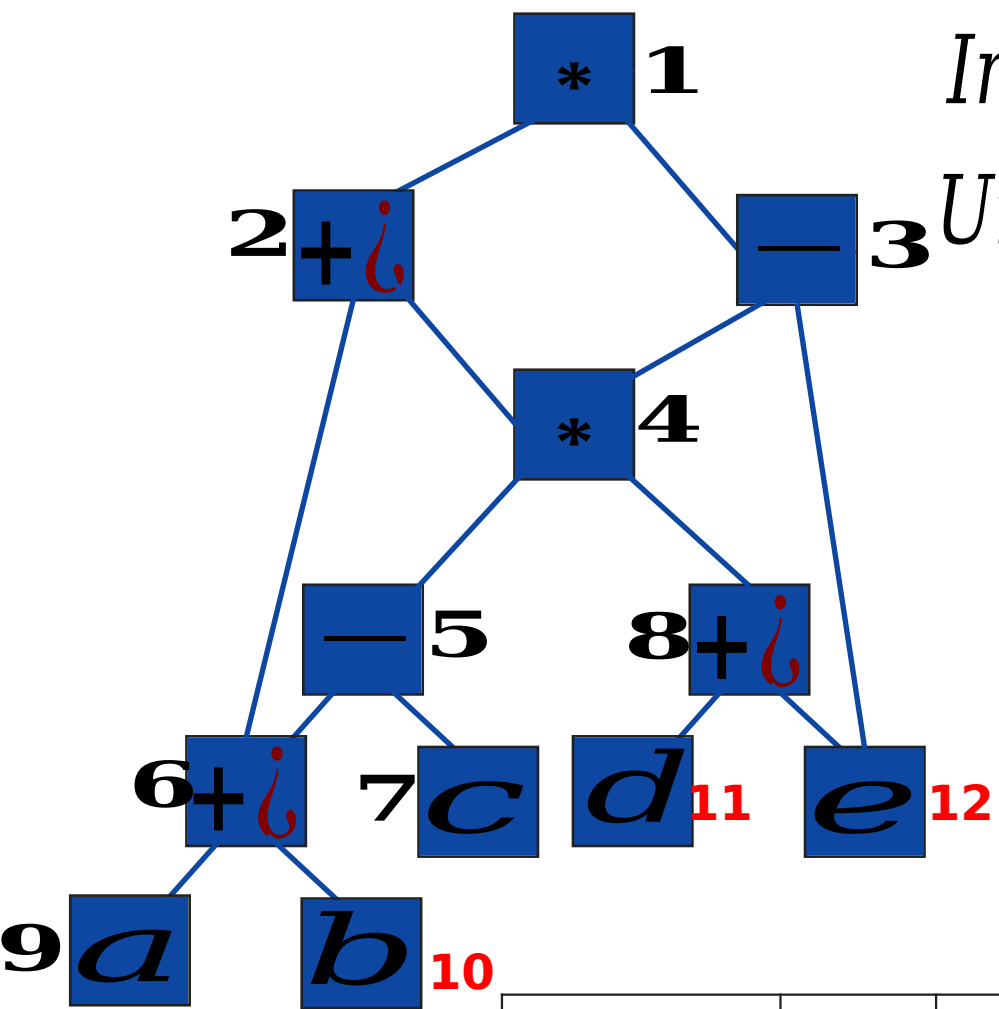
Rightchild of 3 is listed
Parent 2,3 is listed
so list 4

Listed Node	1	2					
-------------	---	---	--	--	--	--	--

Example: Heuristic Ordering



Example: Heuristic Ordering



Interior nodes=1234568

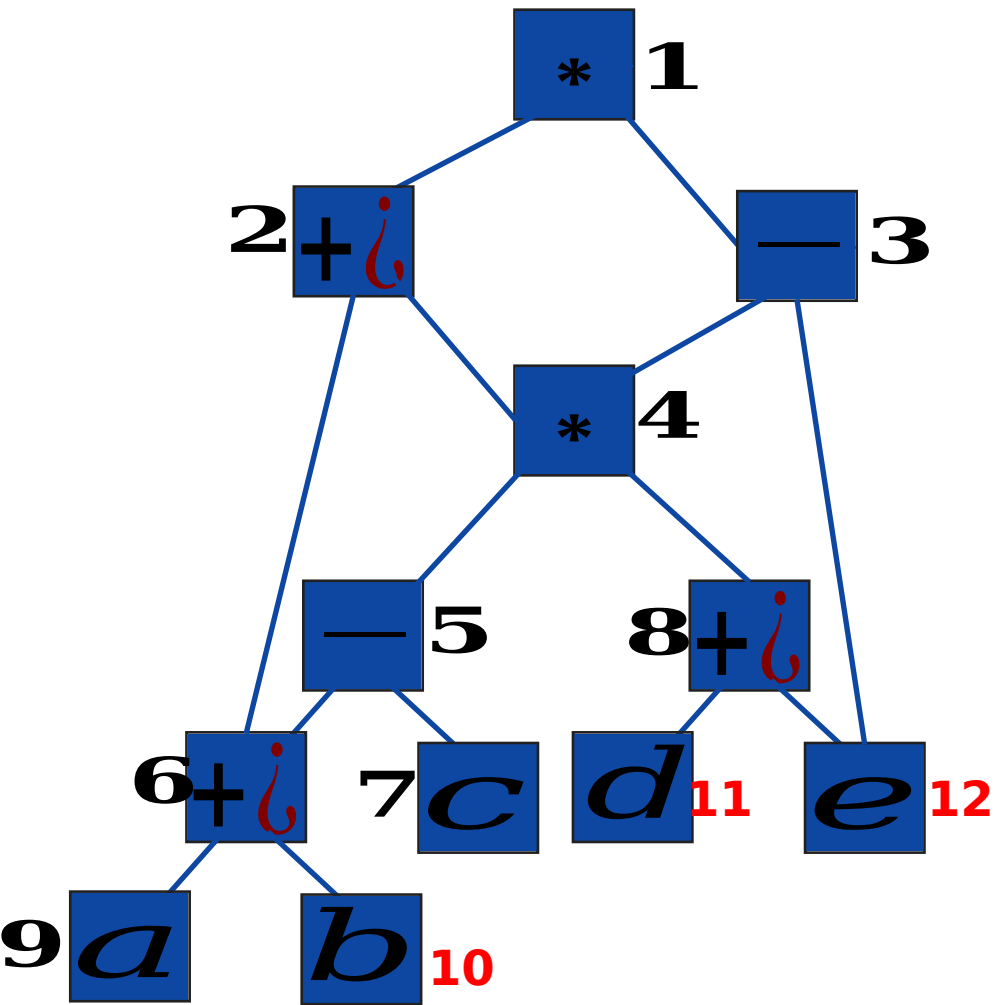
Unlisted nodes=~~1~~~~2~~~~3~~~~4~~~~5~~~~6~~~~8~~

**Pick up an unlisted node,
whose parents have been
listed**

**Rightchild of 4 is 8
Parent 4 is listed
so list 8**

Listed Node	1	2	3	4	5	6	
-------------	---	---	---	---	---	---	--

Example: Heuristic Ordering



Listed Node	1	2	3	4	5	6	8
-------------	---	---	---	---	---	---	---

Reverse Order for three address code = 8 6
5 4 3 2 1

t8=d+e
t6=a+b
t5=t6-c
t4=t5*t8
t3=t4-e
t2=t6+t4
t1=t2*t3

Optim
alThre
e
Addre
ss
code

Labeling Algorithm

- ▶ The labeling algorithm **generates the optimal code for given expression** in which minimum registers are required.
- ▶ Using labeling algorithm the labeling can be done to tree by visiting nodes in bottom up order.
- ▶ For computing the label at node n with the label $L1$ to left child and label $L2$ to right child as,
- ▶ We start in bottom-up fashion and label **left leaf as 1** and **right leaf as 0**.

Example: Labeling Algorithm

t1:=a+b

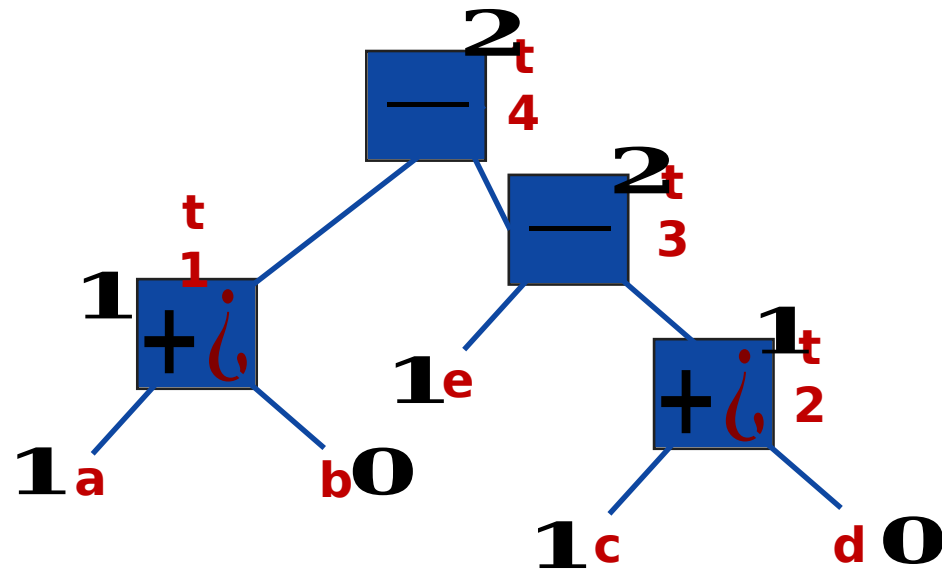
t2:=c+d

t3:=e-t2

t4:=t1-t3

Three Address
Code

$$Label(n) = i \left\{ \begin{array}{l} \text{if } n \text{ is a leaf node} \\ \text{if } n \text{ is an internal node} \end{array} \right.$$



postorder traversal = a b t1 e c d t2 t3 t4



Thank You

