

DISCRETE MATHEMATICS

(离散数学)

云南大学数学系
李 建 平

Chapter 7 Trees

7.1 Trees

7.2 Labeled Trees

7.3 Tree Searching

7.4 Undirected Trees

7.5 Minimum Spanning Trees

7.1 TREES

Let A be a set, and let T be a relation on A . We say that T is an **arborescence** or a **tree** (树) if there is a vertex v_0 in A with the property that there exists a unique path in T from v_0 to every other vertex in A , but no path from v_0 to v_0 .

The vertex v_0 is unique. It is often called the **root** (根) of the tree T , and T is then referred to as a **rooted tree** (有根树). We write (T, v_0) to denote a rooted tree T with root v_0 .

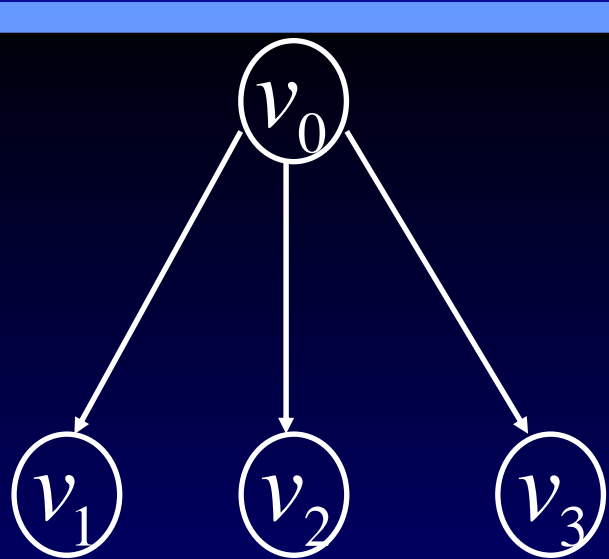
Theorem 1

Let (T, v_0) be a rooted tree. Then

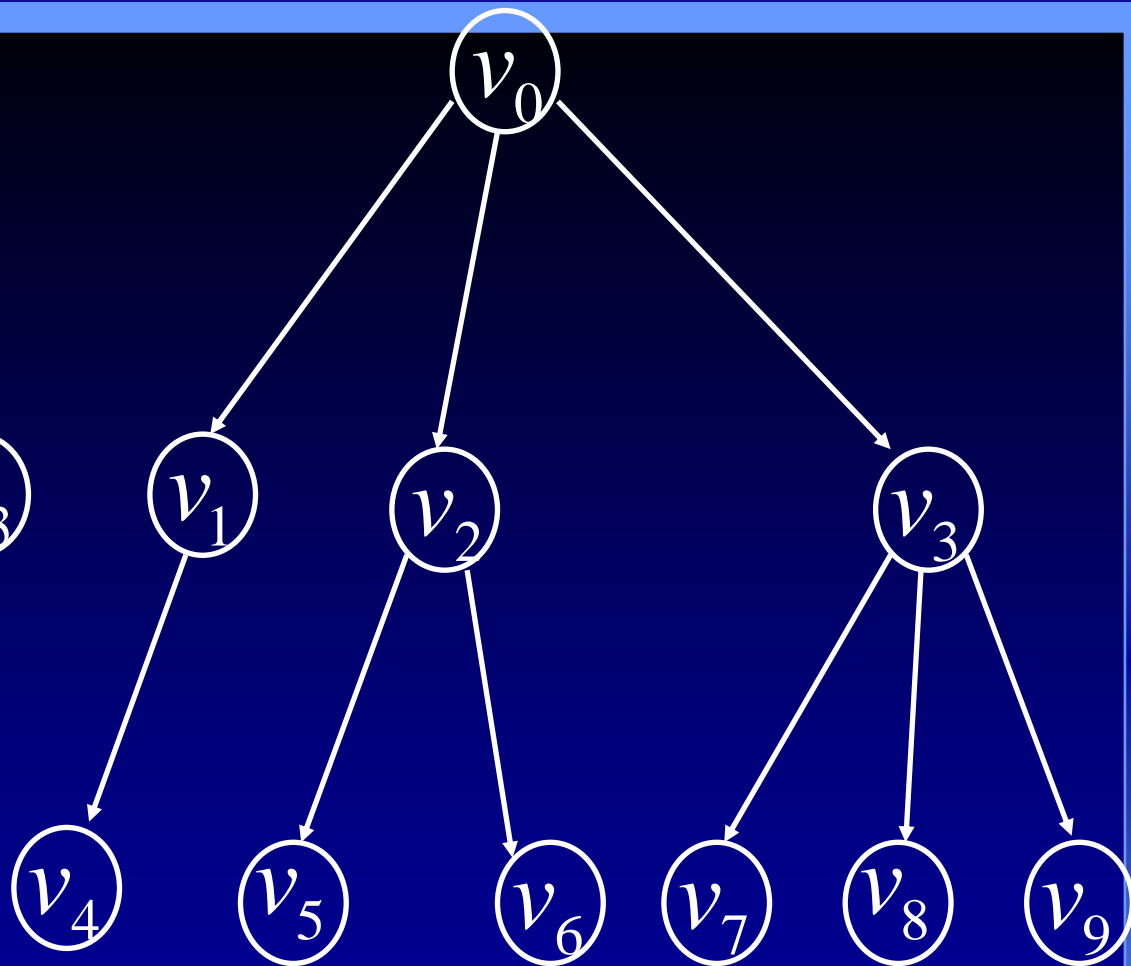
- (a) There are no cycles in T .
- (b) v_0 is the only root of T .
- (c) Each vertex in T , other than v_0 , has in-degree one, and v_0 has in-degree zero.

Let us draw the root v_0 . The terminal vertices of the arcs beginning at v_0 will be called the **level 1** vertices, while v_0 will be said to be at **level 0**. v_0 is sometimes called the **parent** of these level 1 vertices, and the level 1 vertices are called the

offspring of v_0 . This is shown in Figure 7.1(a). Each vertex at level 1 has no other arcs entering it, by part (c) of Theorem 1, but each of these vertices may have arcs leaving the vertex. The arcs leaving a vertex of level 1 are drawn downward and terminate at various vertices, which are said to be at **level 2**.



(a)



(b)

Figure 7.1

The largest level number of a tree is called the **height** (高度) of the tree.

The vertices of the tree that have no offspring are called the **leaves** (树叶) of the tree.

Whenever we draw the digraph of a tree, we automatically assume some ordering at each level by arranging offspring from left to right. Such tree will be called an **ordered tree** (有序树).

Theorem 2

Let (T, v_0) be a rooted tree on a set A . Then

(a) T is irreflexive (反自反的).

(b) T is asymmetric (斜对称的).

(c) If $(a,b) \in T$ and $(b,c) \in T$, then $(a,c) \notin T$, for all a , and b in A .

EXAMPLE 2

Let $A = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$ and
 $T = \{(v_2, v_3), (v_2, v_1), (v_4, v_5), (v_4, v_6), (v_5, v_8), (v_6, v_7), (v_4, v_2), (v_7, v_9), (v_7, v_{10})\}$.

Show that T is a rooted tree and identify the root (shown in Figure 7.2).

If n is a positive integer we say that a tree is an **n-tree** if every vertex has at most n offspring. If all vertices of T , other than the leaves, have exactly

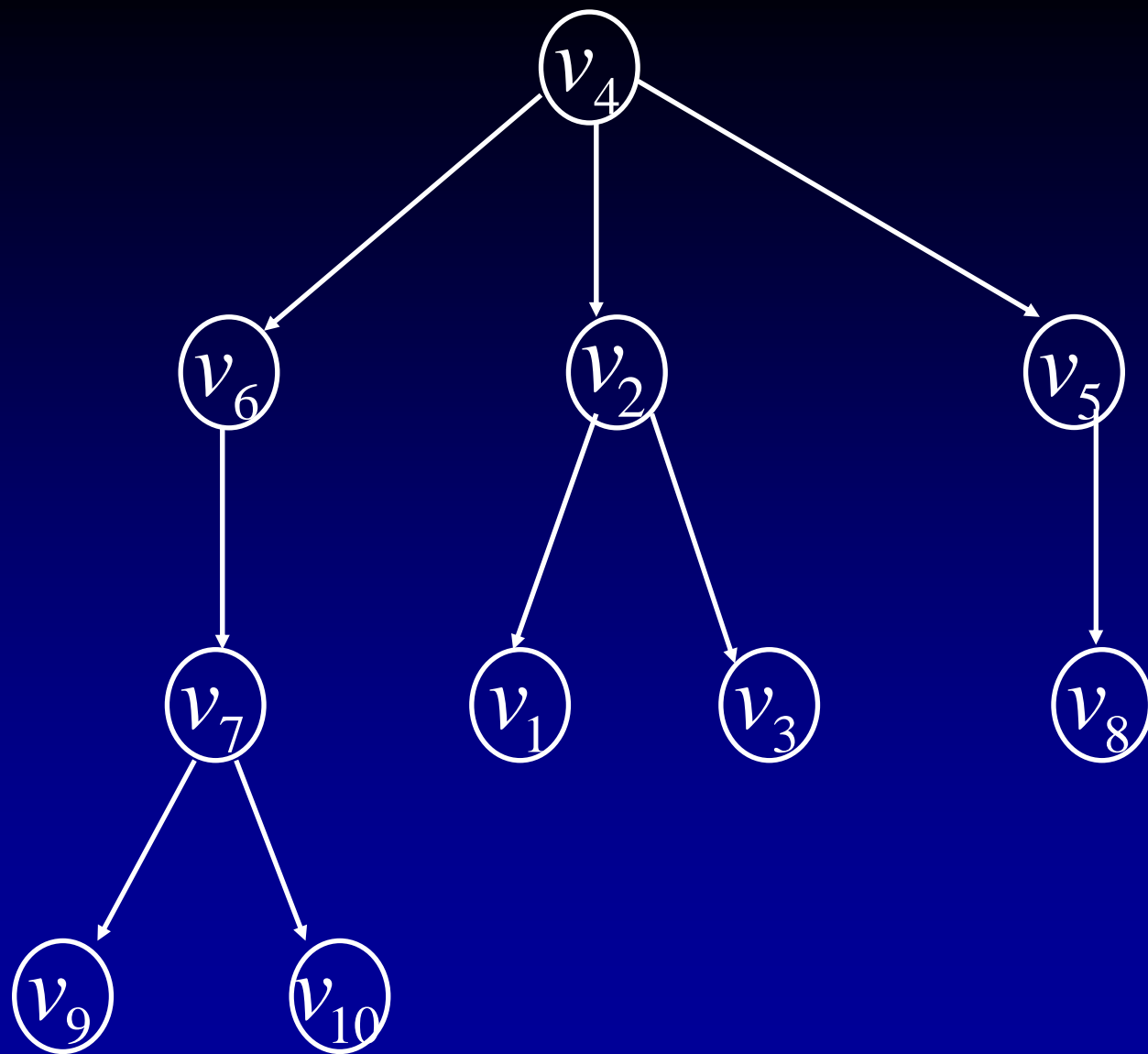


Figure 7.2

n offspring, we say that T is a **complete n -tree**. A 2-tree is often called a **binary tree**, and a complete 2-tree is often called a **complete binary tree**.

$T(v)$ is the tree that results from T in the following way. Deleted all vertices that are not descendants of v and all edges that do not begin or end at any such vertices.

Theorem 3

If (T, v_0) is a rooted tree and $v \in T$, then $T(v)$ is also a rooted tree with root v . We will say that $T(v)$

is the **subtree** (子树) of T beginning at v .

7.2 LABELED TREE (标号树)

Consider the fully parenthesized, algebraic expression

$$(3 - (2 * x)) + ((x - 2) - (3 + x)).$$

Each such expression has a **central operator**, corresponding to the last computation that can be performed. Thus $+$ is central to the main expression above, $-$ is central to $(3 - (2 * x))$ and so on. An important graphical representation of such an expression is as a labeled binary tree. (shown in

Figure 7.6).

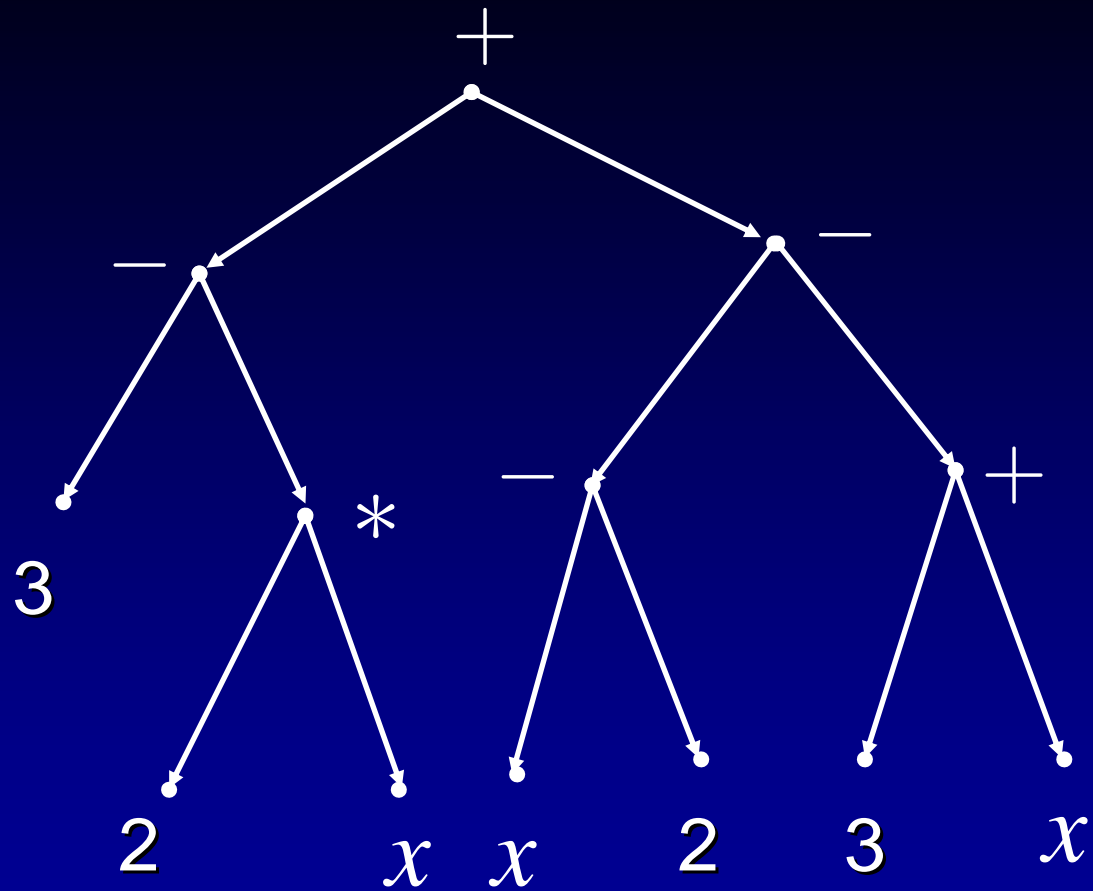


Figure 7.6

We start with an n -tree (T, v_0) . Each vertex in T has at most n offspring. The offspring of any vertex are labeled with distinct numbers from the set $\{1, 2, \dots, n\}$.

Such a labeled digraph is sometimes called **positional tree** (位置树).

Figure 7.8 shows the digraph of a positional 3-tree. If offspring 1 of any vertex v actually exists, the arc from v to that offspring is drawn sloping to the left. Offspring 2 of any vertex v is drawn vertically downward from v , whenever it occurs.

Similarly, offspring labeled 3 will be drawn to the right. Naturally, the root is not labeled, since it is not an offspring.

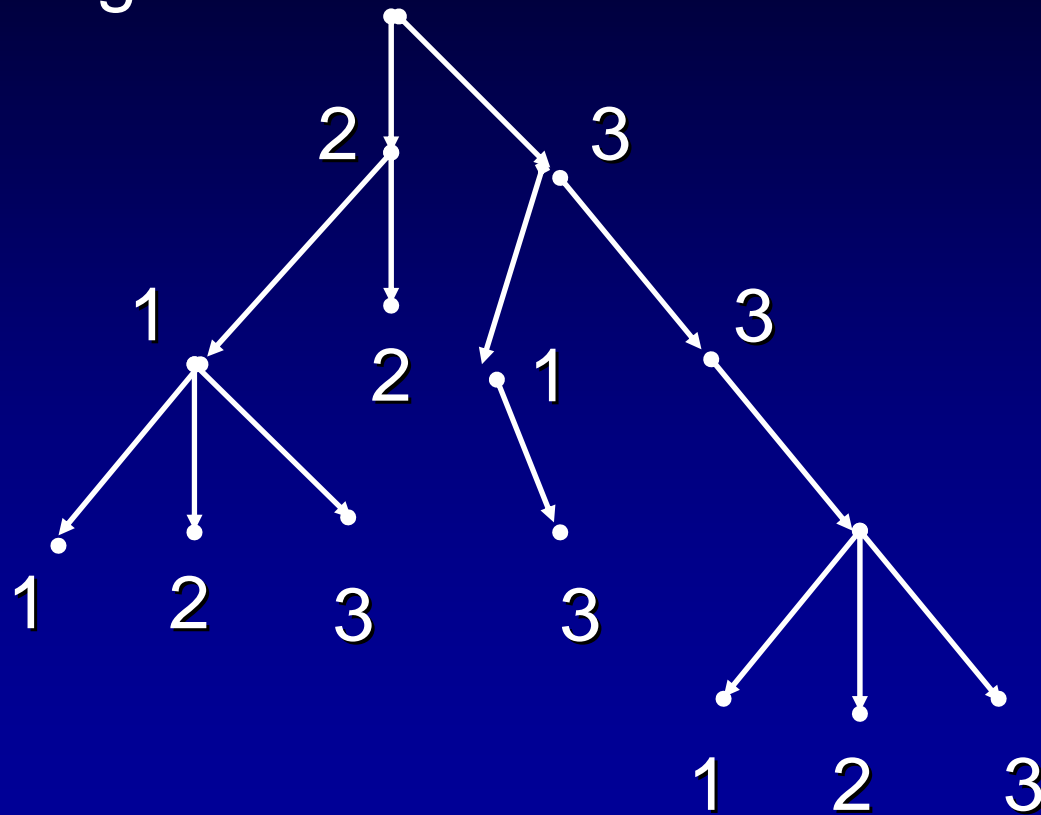


Figure 7.8

The **positional binary tree** is of special importance. The positions for potential offspring are often labeled left and right, instead of 1 and 2. (Figure 7.9)

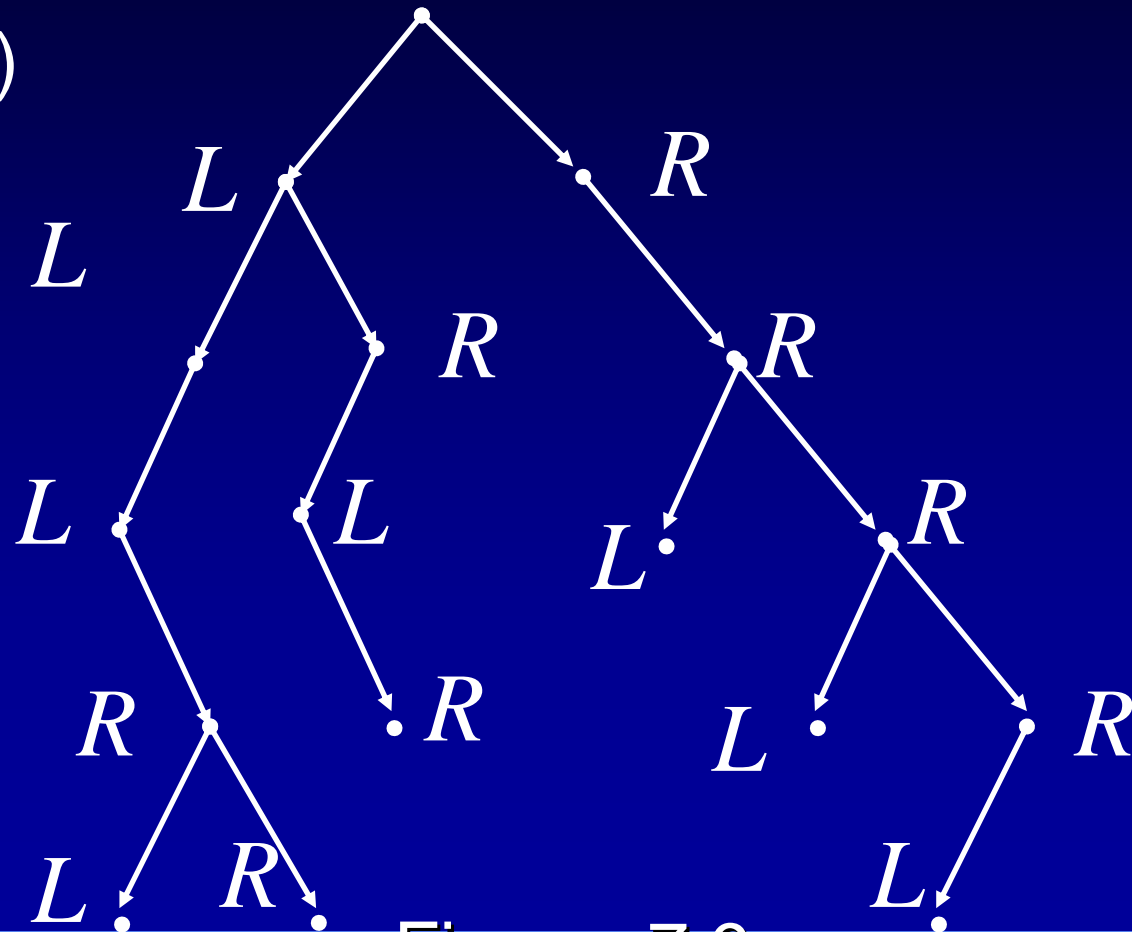





Figure 7.9

Computer Representation of Binary Positional Trees

In the section 4.6, we discussed an idealized information storage unit called a **cell**. A cell contains two items. One is data of some sort and the other is a pointer to the next cell, that is, an address where the next cell is located.

We need here an extended version of this concept, called a **doubly linked list**, in which each cell contains two pointers and a data item. We use the pictorial symbol  to represent

these new cells. The center space represents data storage and the two pointers, called the **left pointer** and the **right pointer**, are represented as before by dots and arrow, we use the symbol  for a pointer signifying no additional data. If either offspring fails to exist, the corresponding pointer will be .

We implement this representation by using three arrays: LEFT holds pointers to the left offspring, RIGHT holds the pointers to the right offspring, and DATA holds information or labels related to

each vertex, or pointers to such information. The value 0, used as a pointer, will signify that the corresponding offspring does not exist. To the linked list and the arrays we add a starting entry that points to the root of the tree.

EXAMPLE 1

In Figure 7.10(a), we represent this tree as a doubly linked list, in symbolic form. In Figure 7.10(b), we show the implementations of this list as a sequence of three arrays.

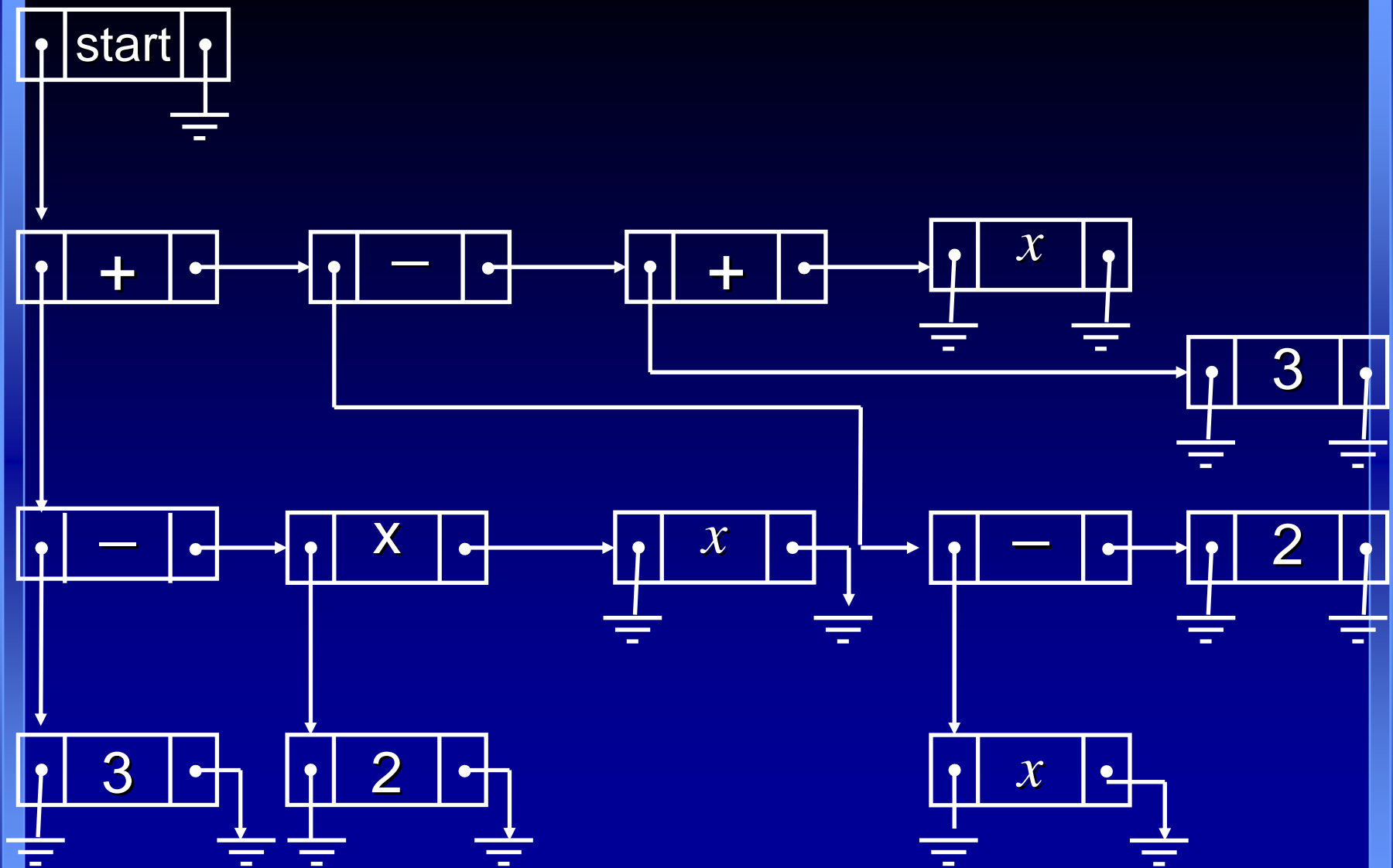


Figure 7.10 (a)

INDEX

1
2
3
4
5
6
7
8
9
10
11
12
13
14

LEFT

2
3
4
0
6
0
0
9
10
0
0
13
0
0

DATA

×
+
−
3
×
2
x
−
−
x
2
+
3
x

RIGHT

0
8
5
0
7
0
0
12
11
0
0
14
0
0

(b)

7.3 TREE SEARCHING (树的搜索)

If T is the tree of an algebraic expression, then at each vertex we may want to perform the computation indicated by the operator that labels that vertex. Performing appropriate tasks at a vertex will be called **visiting** (访问) the vertex.

The process of visiting each vertex of a tree in some specific order will be called **searching** (搜索) the tree or performing a **tree search**. This process is called **walking** or **traversing** (遍历) the tree.

Let us consider tree searched on binary positional trees. We denoted these potential offspring by v_L (the left offspring) and v_R (the right offspring), and either or both may be missing.

Let T be a binary positional tree with root v . Then, if v_L exists, the subtree $T(v_L)$ (see Section 7.1) will be called the **left subtree** of T , and if v_R exists, the subtree $T(v_R)$ will be called the **right subtree** of T .

We first describe a method of searching called a **preorder search**. Consider the following algorithm for searching a positional binary tree T with root v .

ALGORITHM PREORDER

Step 1 Visit v .

Step 2 If v_L exists, then apply this algorithm to $(T(v_L), v_L)$.

Step 3 If v_R exists, then apply this algorithm to $(T(v_R), v_R)$.

End of Algorithm

EXAMPLE 1

Let T be the labeled, positional binary tree whose digraph is shown in Figure 7.14(a). The root of this tree is the vertex labeled A .

According to PREORDER, applied to T , we will visit the root and print A , then search subtree 1, and then subtree 7.

The result of the complete search of T is to print the string ABCDEFGHIJKL.

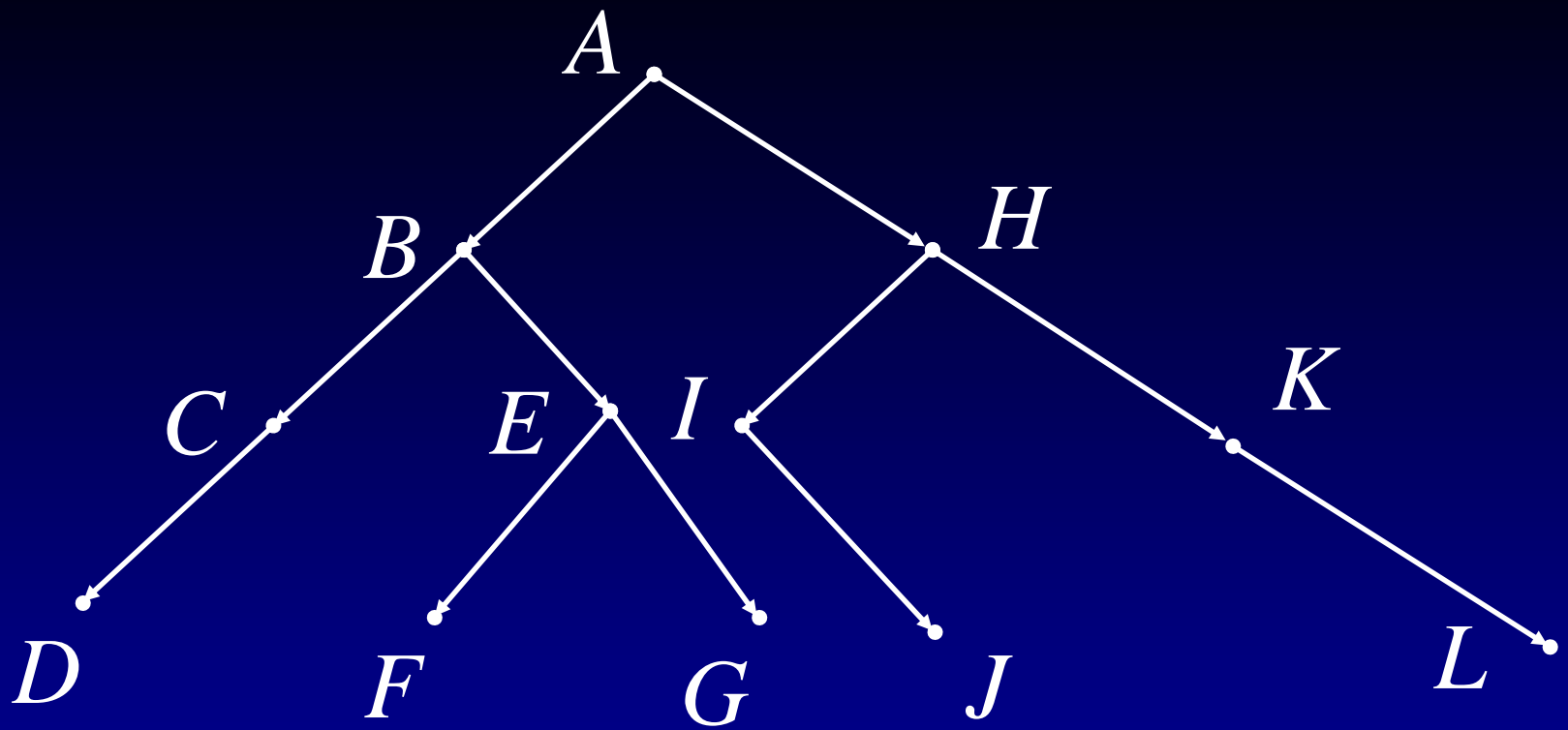


Figure 7.14 (a)

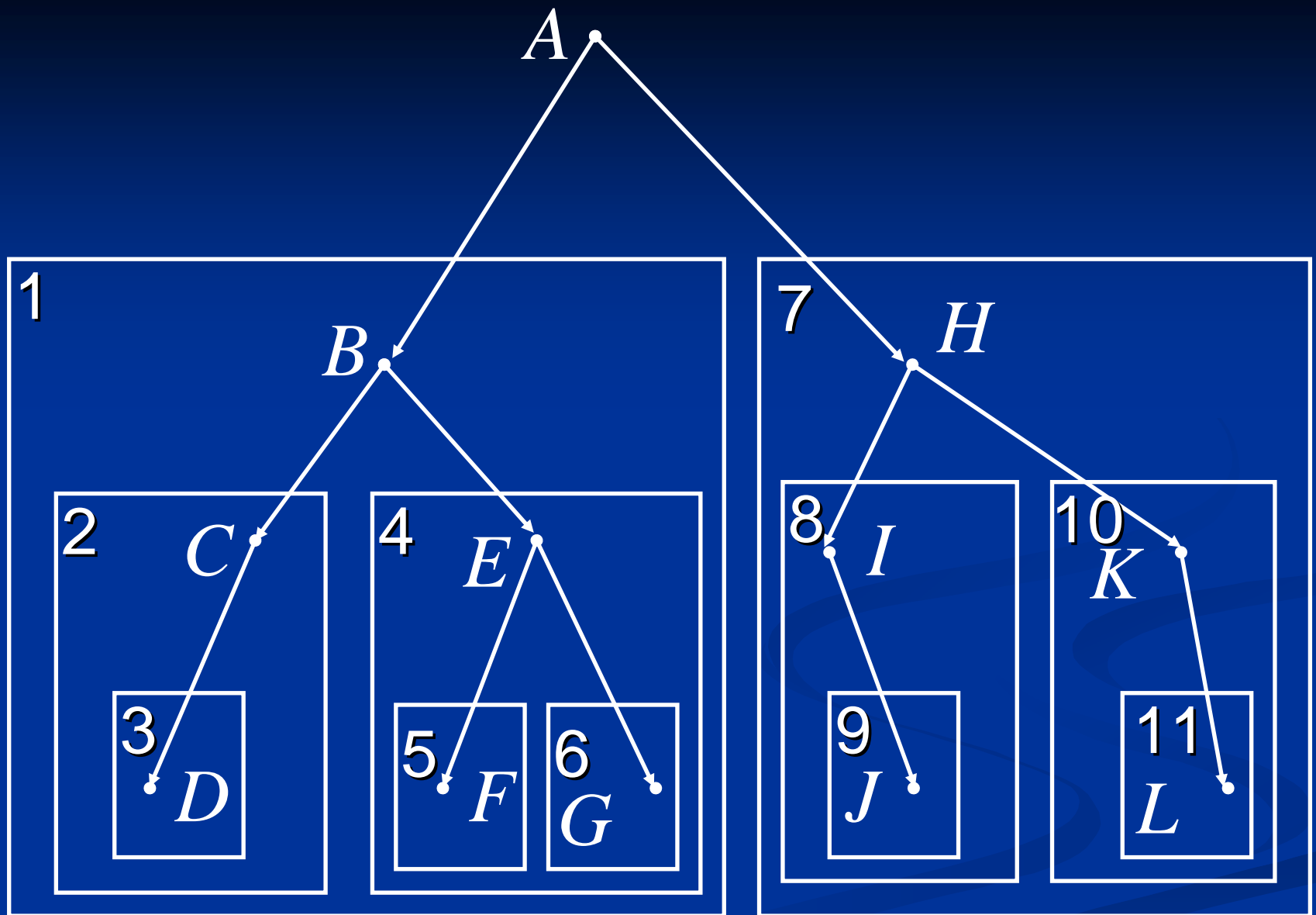


Figure 7.14 (b)

EXAMPLE 2

Consider the completely parenthesized $(a-b)\times(c+(d\div e))$. Figure 7.15(a) shows the digraph of the labeled, positional binary tree representation of this expression.

Proceeding as in Example 1 and supposing again that visiting v simply prints out the label of v , we see that the string $\times - ab + c \div de$ is the result of the search. This is the **prefix** (前綴形式) or **Polish form** (波兰形式) of the given algebraic expression.

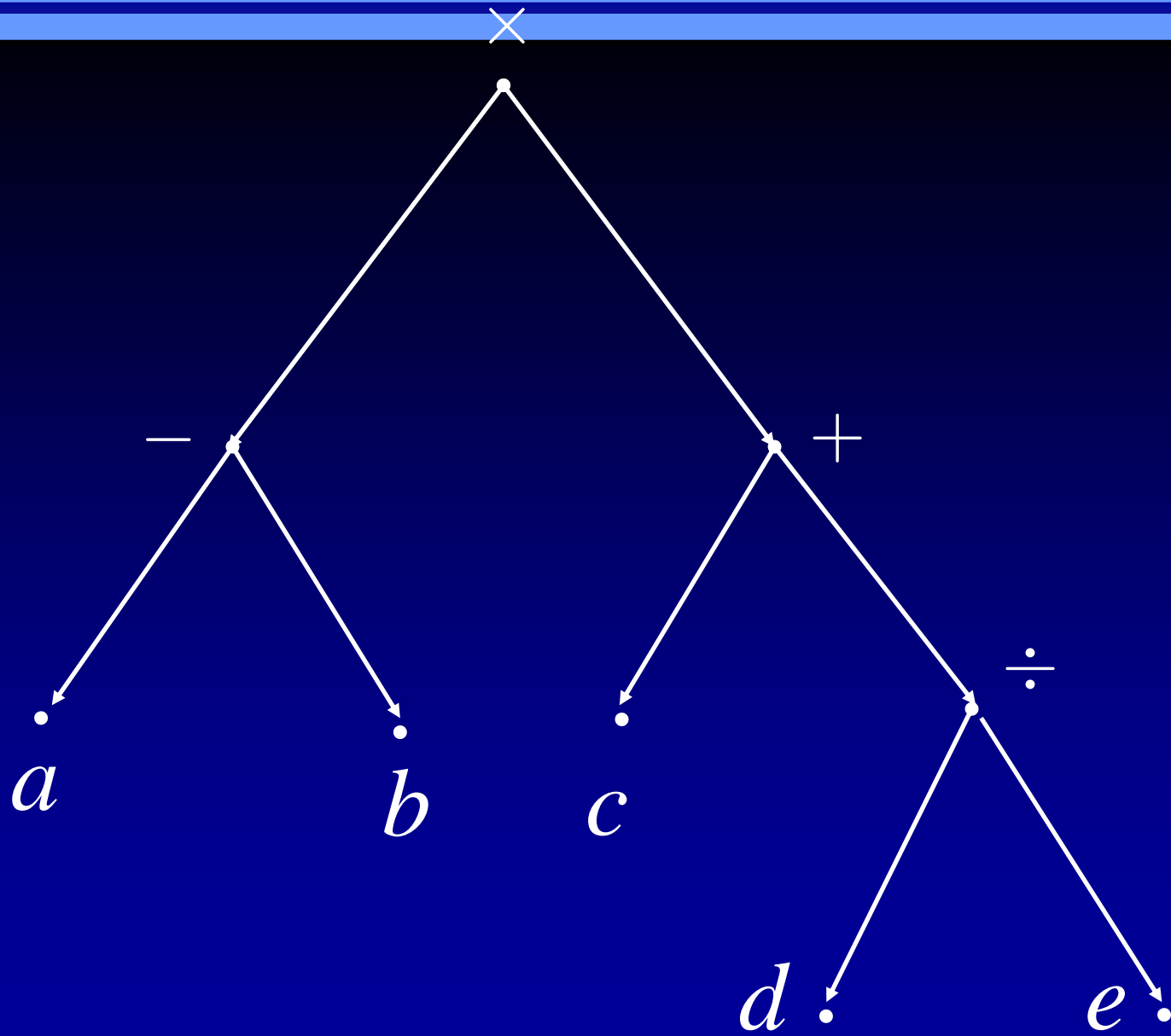


Figure 7.15 (a)

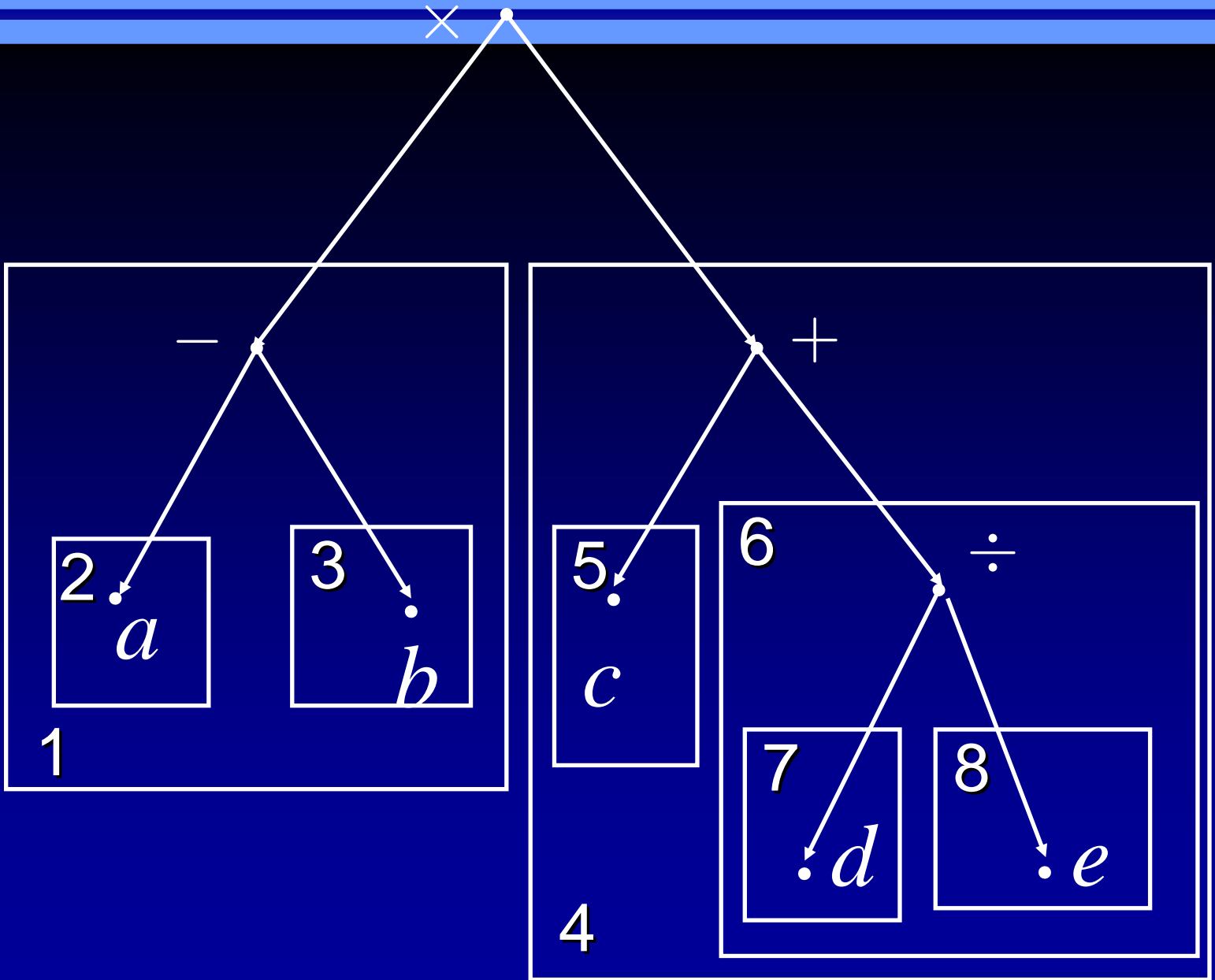


Figure 7.15 (b)

Move from left to right until we find a string of the form Fxy , where F is the symbol for a binary operation ($+$, $-$, \times , and so on), and x and y are numbers. Evaluate xFy and substitute the answer for the string Fxy . Continue this procedure until only one number remains.

Consider now the following informal descriptions of two other procedures for searching a positional binary tree T with root v .

ALGORITHM INORDER

Step 1 Search the left subtree $(T(v_L), v_L)$, if it exists.

Step 2 Visit the root, v .

Step 3 Search the right subtree $(T(v_R), v_R)$, if it exists.

End Algorithm

This search is called the **inorder** (中序) search.

ALGORITHM POSTORDER

Step 1 Search the left subtree $(T(v_L), v_L)$, if it exists.

Step 2 Search the right subtree $(T(v_R), v_R)$, if it exists.

Step 3 Visit the root, v .

End of Algorithm

This search is called **postorder** (后序) search.

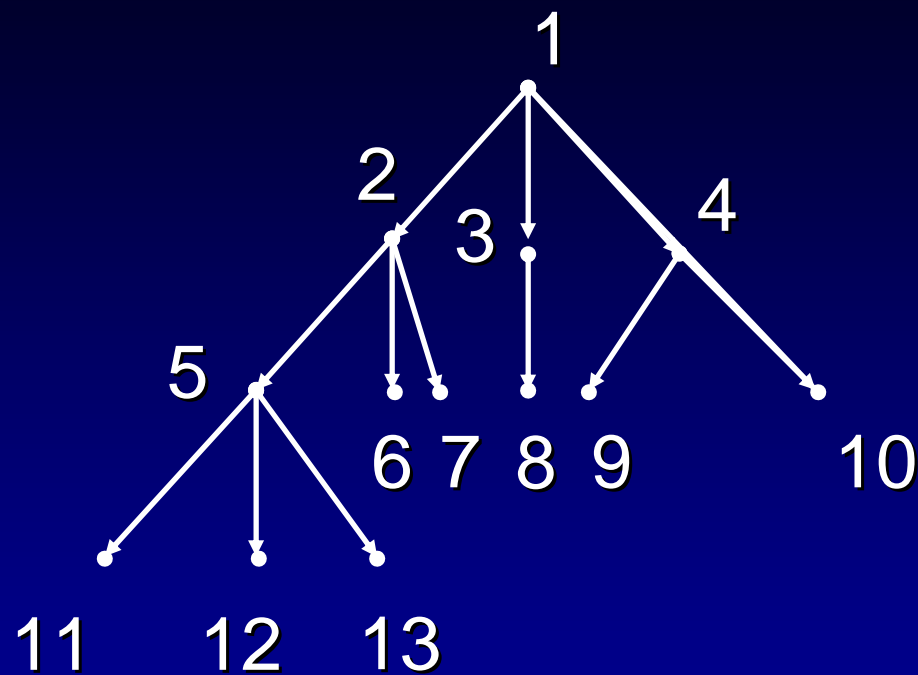
This is the **postfix** (后缀形式) or **reverse Polish** form (逆波兰形式) of the expression.

◉ Searching General Trees

We now show that any ordered tree T (see the section 7.1) may be represented as a binary positional tree that, although different from T , captures all the structure of T and can be used to recreate T .

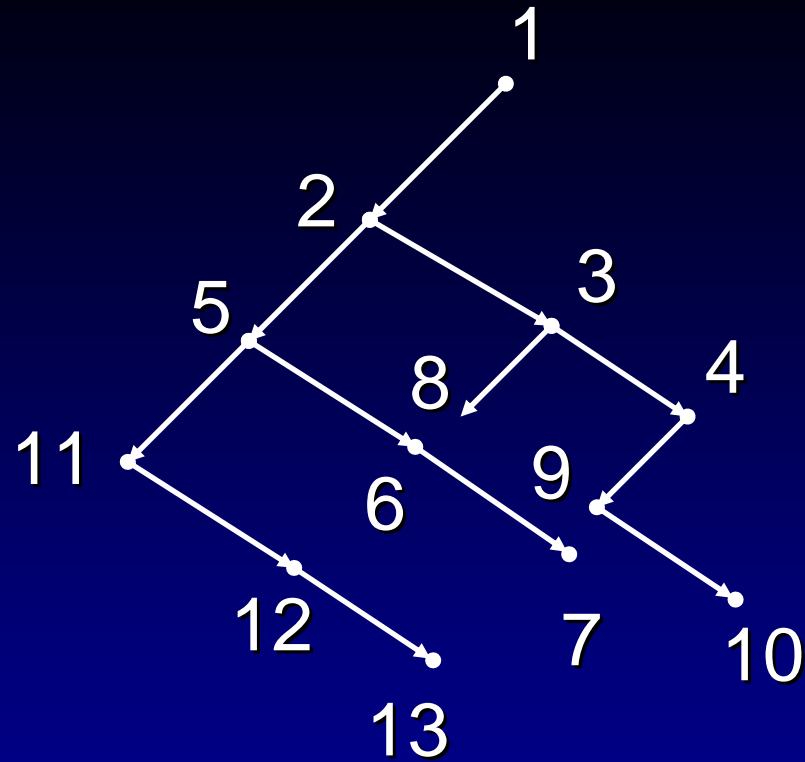
Let T be any ordered tree and A the set of vertices of T . Define a binary positional tree $B(T)$ on the set of vertices A as follows. If $v \in A$, then the left offspring v_L of v in $B(T)$ is the first offspring of v in T (in the given order of siblings (兄弟) in T), if it exists. The right offspring v_R of v in $B(T)$ is the next sibling (兄弟) of v in T (in the given order of sibling in T), if it exists.

A doubly-linked-list representation of $B(T)$ is sometimes simply referred to as a **linked-list representation of T** .



T

Figure 7.16 (a)



$B(T)$

Figure 7.16 (b)

Pseudocode Versions

SUBROUTINE PREORDER (T, v)

1. CALL VISIT(v)
2. IF(v_L exists) THEN
 - a. CALL PREORDER($T(v_L), v_L$)
3. IF(v_R exists) THEN
 - a. CALL PREORDER($T(v_R), v_R$)
4. RETURN

END OF SUBROUTINE PREORDER

SUBROUTINE INORDER (T, v)

1. IF(v_L exists) THEN

a. CALL INORDER ($T(v_L), v_L$)

2. CALL VISIT(v)

3. IF(v_R exists) THEN

a. CALL INORDER ($T(v_R), v_R$)

4. RETURN

END OF SUBROUTINE INORDER

SUBROUTINE POSTORDER

1. IF(v_L exists) THEN

a. CALL POSTORDER ($T(v_L), v_L$)

2. IF(v_R exists) THEN

a. CALL POSTORDER ($T(v_R), v_R$)

3. CALL VISIT(v)

4. RETURN

END OF SUBROUTINE POSTORDER

7.4 UNDIRECTED TREES (无向树)

An **undirected tree** is simply the **symmetric closure** of a tree. The graph of an undirected tree T will have a single line without arrows connecting vertices a and b whenever (a,b) and (b,a) belong to T . The set $\{a,b\}$, is called an (undirected) **edge** of T . In this case, the vertices a and b are called **adjacent vertices**.

Let R be a symmetric relation, and let $P: v_1, v_2, \dots, v_n$ be a path in R . We will say that P is **simple** if no

two edges of P corresponding to the same undirected edge. In addition, if v_1 equals v_n (so that P is a cycle), we will call P a **simple cycle**.

We will say that a symmetric relation R is **acyclic** if it contains no simple cycles.

Theorem 1

Let R be a symmetric relation on a set A . Then the following statements are equivalent.

- (a) R is an undirected tree.
- (b) R is connected and acyclic.

Theorem 2

Let R be a symmetric relation on a set A . Then R is an undirected tree if and only if either of the following statements is true.

- (a) R is acyclic, and if any undirected edge is added to R , the new relation will not be acyclic.
- (b) R is connected, and if any undirected edge is removed from R , the new relation will not be connected.

Theorem 3

A tree with n vertices has $n-1$ edges.

Spanning Trees of Connected Relations

If R is a symmetric, connected relation on a set A , a tree T on A is called as a **spanning tree** (支撑树) for R if T is a tree with exactly the same vertices as R .

There is interest in an **undirected spanning tree** for a symmetric connected relation R . This is just the **symmetric closure** of a spanning tree.

Let R be a relation on a set A , and $a, b \in A$. Let $A_0 = A - \{a, b\}$, and $A' = A_0 \cup \{a'\}$, where a' is some new element not in A . Define a relation R' on A' as follows. Suppose $u, v \in A'$, $u \neq a', v \neq a'$. Let $(a', u) \in R'$ if and only if $(a, u) \in R$ or $(b, u) \in R$. Let $(u, a') \in R'$ if and

only if $(u, a) \in R$ or $(u, b) \in R$. Finally, let $(u, v) \in R'$ if and only if $(u, v) \in R$. We say that R' is a result of merging (收缩、合并) the vertices a and b .

The algebraic form of this merging process is also very important.

Suppose now that vertices a and b of a relation R are merged into a new vertex a' that replaces a and b to obtain the relation R' . To determine the matrix of R' . We proceed as the following algorithm:

Step 1 Let row i represent vertex a and row j represent vertex b . Replace row i by the **join** of

rows i and j . The join of two n -tuples of 0's and 1's has a 1 in some position exactly when either of those two n -tuples has a 1 in that position.

Step 2 Replace column i by the join of columns i and j .

Step 3 Restore the main diagonal to its original values in R .

Step 4 Delete row j and column j .

We can now give an algorithm for finding a spanning tree for a symmetric, connected relation R on the set $A = \{v_1, v_2, \dots, v_n\}$. The method is a

special case of an algorithm called **Prim's algorithm**. The steps are as follows:

Step 1 Choose a vertex v_1 of R , and arrange the matrix of R so that the first row corresponds to v_1 .

Step 2 Choose a vertex v_2 of R such that $(v_1, v_2) \in R$, merge v_1 and v_2 into a new vertex v_1' , representing $\{v_1, v_2\}$, and replace v_1 by v_1' . Compute the matrix of the resulting relation R' . Call the vertex v_1' a merge vertex.

Step 3 Repeat steps 1 and 2 on R' and on all subsequent relation until a relation with a single

vertex is obtained. At each step, keep a record of the set of original vertices that is represented by each merged vertex.

Step 4 Construct the spanning as follows. At each stage, when merging vertices a and b , select an edge in R from one of the original vertices represented by a to one of the original vertices represented by b .

7.5 MINIMUM SPANNING TREES

A **weighted graph** (赋权图) is a graph for which each edge is labeled with a numerical value called its **weight** (权重).

Find an undirected spanning tree for which the total weight of the edges in the tree is as small as possible. Such a spanning tree is called a **minimum spanning tree** (最小支撑树).

◉ PRIM'S ALGORITHM

Let R be a symmetric, connected relation with n

vertices.

Step 1 Choose a vertex v_1 of R . Let $V = \{v_1\}$ and $E = \phi$.

Step 2 Choose a nearest neighbor v_i of V that is adjacent to $v_j, v_j \in V$, and for which the edge (v_i, v_j) does not form a cycle with members of E . Add v_i to V and add (v_i, v_j) to E .

Step 3 Repeat Step 2 until $|E| = n - 1$. Then V contains all n vertices of R , and E contains the edges of a minimum spanning tree for R .

End of Algorithm

This is an example of a **greedy** algorithm.

Theorem 1

Prim's algorithm, as given, produces a minimum spanning tree for the relation.

If a symmetric connected relation R has n vertices, then Prim's algorithm has running time $\Theta(n^2)$.

Kruskal's algorithm is another example of a greedy algorithm that produces an optimal solution.

KRUSKAL'S ALGORITHM

Let R be a symmetric, connected relation with n vertices and let $S = \{e_1, e_2, \dots, e_k\}$ be the set of

weighted edges of R .

Step 1 Choose an edge e_1 in S of least weight.
Let $E = \{e_1\}$ Replace S with $S - \{e_1\}$.

Step 2 Select an edge e_i in S of least weight that will not make a cycle with members of E . Replace E with $E \cup \{e_i\}$ and S with $S - \{e_i\}$.

Step 3 Repeat step 2 until $|E| = n - 1$.

End of Algorithm

THE END