Chapter 5 Binary Trees 二叉树

1

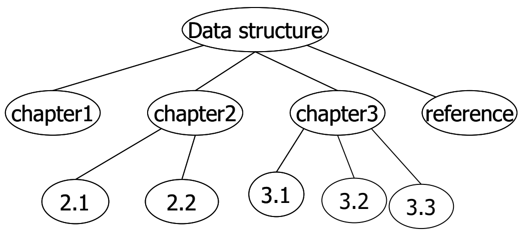
1

Content

2

2

5.1 Basic Concepts of Tree and representation



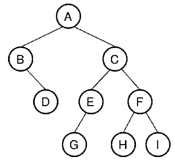
Data structure
- chapter1
- Chapter2
  - 2.1
  - 2.2
- chapter3
  - 3.1
  - 3.2
  - 3.3
- reference

Compare to list, Tree structures permit both search and insert efficient to large collections of data

3

3

Terminology 术语(1)



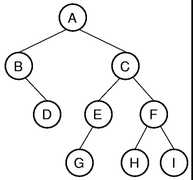
- A tree consists of:
  - Nodes(结点): finite set of elements
  - Edges/branches(边): directed lines connecting the nodes
- For a node:
  - Degree(度): number of branches away from the node
- For a tree:
  - Root(根): node with indegree 0
  - nodes different from the root must have indegree 1

4

4

Terminology(2)

- Leaf(叶子) : node with degree 0
- Internal node (内部结点) : node that not a leaf
- Parent(双亲):
- Child(孩子):
- Siblings(兄弟): nodes with the same parent
- Path(路径): if $n_1, n_2, \ldots, n_k$ is a sequence of nodes in the tree such that n_i is the parent of n_{i+1} for $0 < i < k$, then this sequence is called a path from n_1 to n_k , the length of the path is $k-1$

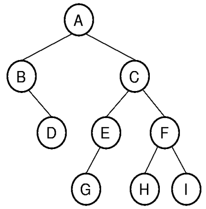


5

5

Terminology(3)

- Ancestor(前辈):
- Descendent(后代):
- Depth(深度) of a node M: the length of the path from root to M
- Level (层) : all nodes of depth d are at level d in the tree
- Height (高度) of a tree: the depth of deepest node in tree plus 1
- Sub-tree(子树): connected structure below the root



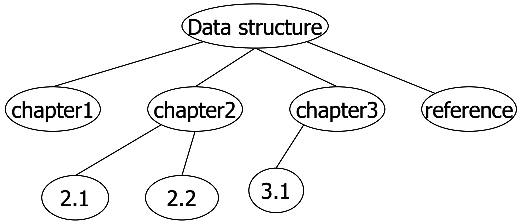
6

6

Tree Representation(1)

General tree

通用树



7

7

Tree Representation(2)

Indented list

缩进式

Data structure
  chapter1
  Chapter2
    2.1
    2.2
  chapter3
    3.1
    3.2
    3.2
  reference

8

8

Tree Representation(3)

Parenthetical list 括号式

Data Structure (chapter1 chapter2 (2.1 2.2)
chapter3(3.1 3.2 3.3) reference)

9

9

5.2 Definition and properties of Binary tree

5.2.1 Definition of Binary tree 二叉树的定义

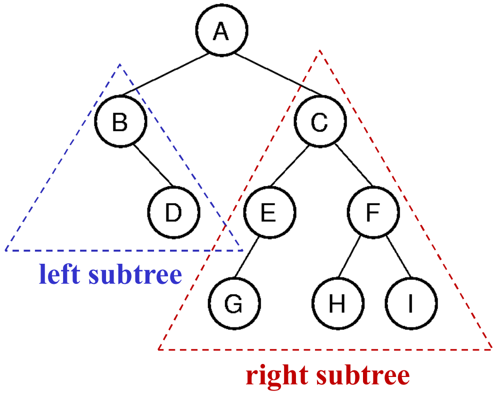
5.2.2 Properties of Binary tree 二叉树的性质

10

10

5.2.1 Definition of Binary tree

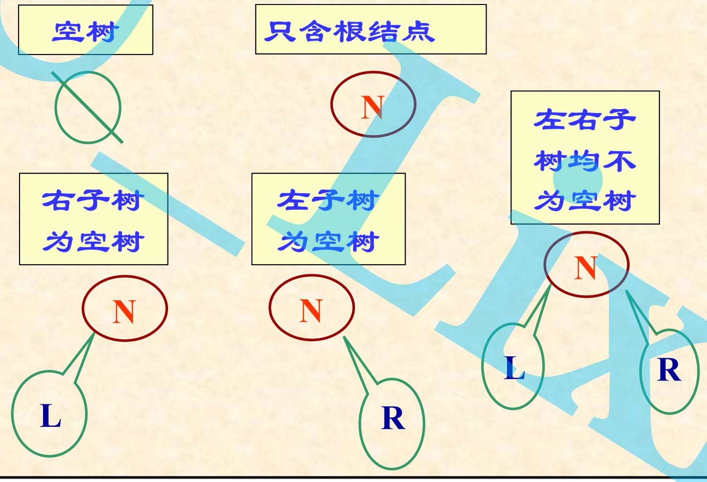
For binary tree(二叉树), any node cannot have more than two sub-trees(left and right)



11

11

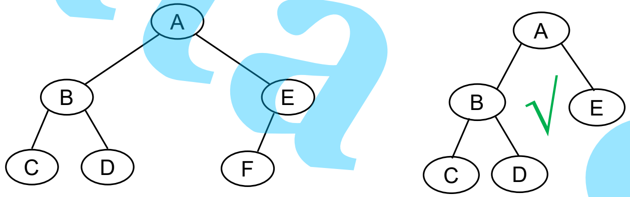
二叉树的五种基本形态



12

Full Binary Trees(满二叉树)

Each node in a full binary tree is either a leaf (degree is 0) or internal node with exactly two non-empty children(degree is 2).



13

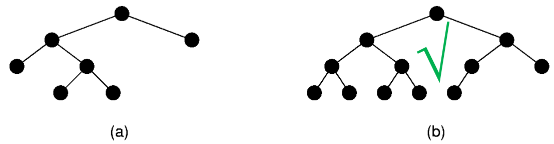
13

Complete Binary Trees(完全二叉树)

In the complete binary tree of height d,

1) all levels except level d-1 must be completely full

2) The d-1 level has all of its nodes filled from the left side.



14

14

5.2.2 Properties(性质) of Binary tree

性质1： 在二叉树的第i (i≥0)层上至多有2^i个结点

证明（用归纳法）

1) $i = 0$ 时，只有1个根结点： $2^i = 2^0 = 1$

2) 假设i-1时命题成立，即i-1层最多有2^{i-1}个结点。

3）因为二叉树上每个结点至多有两棵子树，则第 i 层的结点数最多为 $2^{i-1} \times 2 = 2^i$ √

15

15

性质2: 高度为k(k≥1)的二叉树上至多含2^k-1 个结点

证明:

根据性质1，高度为k的二叉树上的结点数至多为 $2^0+2^1+\cdots\cdots+2^{k-1} = 2^k-1$ √

高度为k(k≥1)的二叉树上至少含多少个结点呢？？

16

16

性质3： 对任何一棵二叉树，若它含有n_0个叶子结点、n_2个度为2的内部结点，则必存在关系式：

$n_0 = n_2+1$

证明：

二叉树上结点总数 $n = n_0 + n_1 + n_2$

二叉树上分支总数 $b = n_1+2n_2$ （后继）

分支总数还可表示为 $b = n-1$ （前驱）

由此， $n0 = n2 + 1$ √

17

17

性质4 具有n个结点的完全(complete)二叉树的高度为$\lfloor \log_2 n \rfloor+1$

证明：

1) 设完全二叉树的高度为 k
2) 则根据性质2 得 $2^{k-1} -1< n <= 2^k-1$

   即 $k-1 < \log_2 (n+1) <= k$
3) 因k只能是整数，因此，$k=\lfloor \log_2 n \rfloor+1$ √

推论：具有n个结点的二叉树的最大高度$H_{max} = n$，最小高度$H_{min} = \lfloor \log_2 n \rfloor + 1$

18

18

性质5: The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.

Proof: (性质3的特例)

for a non-empty full binary tree

因为 $N_{inte\_nod} = n_2$;

所以 $N_{leave} = n_0 = n_2+1 = N_{inte\_nod} +1$

19

19

性质6: The number of empty subtrees in a non-empty Binary tree is one more than the number of nodes in the tree. $n_1+2n_0 = n+1$

Proof: left $=2*n_0 +n_1$

$=n_0 +n_0 + n_1$

$=n_2 +1 + n_0 + n_1$

$=n +1$

思考：非空二叉树中，非空子树的个数是多少呢？

20

20

5.3 Binary Tree Node implementation

5.3.1 Pointer-based node implementation

5.3.2 Space Requirements and Overhead analysis

5.3.3 Array-based implementation for CBT

21

21

二叉树的链式存储表示：链式二叉树

用2个类表达链式二叉树

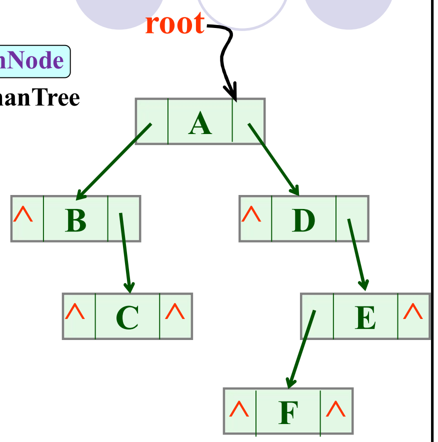
- 结点类：BSTNode/ VarBinNode
- 链式二叉树类：BST/HuffmanTree

二叉结点结构

left	data	right

1个根指针几乎就可描述一棵链式二叉树

- ✓ 根指针root指向根结点，
- ✓ 然后由根结点的左右孩子指针又分别指向其左右孩子，依此往下，就描述了整棵树



22

二叉结点所涉及的基本操作有：

1.	返回结点值	element( )
2.	设置结点值	setElement(const E&)
3.	返回左孩子	left()
4.	设置左孩子	setLeft(BinNode*)
5.	返回右孩子	right()
6.	设置右孩子	setRight(BinNode*)
7.	是否叶子：	isLeaf()

二叉结点结构

left	data	right

23

23

5.3.1 Pointer-based node implementation

二叉结点结构

left	data	right

```
// simple binary tree node imeplemetation
template <class E>
class BSTNode {
private:
  E it;          // The node's data value
  BSTNode* lc;   // Pointer to left child
  BSTNode* rc;   // Pointer to right child

public:
  BSTNode ( )   { lc = rc = NULL; }
  BSTNode (E  e, BSTNode*   l=NULL,  BSTNode*  r=NULL)
  {
     it = e;   lc = l;   rc = r;
  }
```

24

24

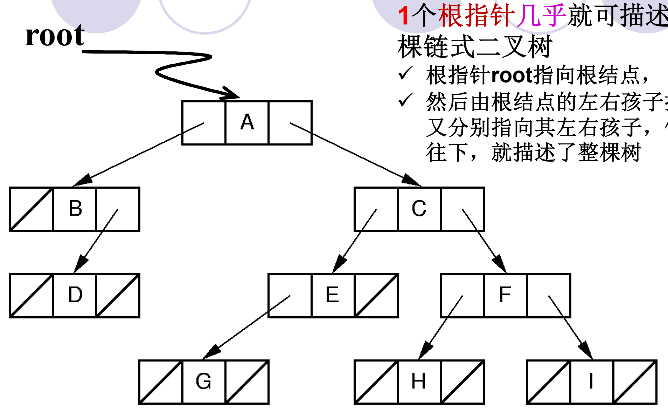
Simple Binary tree node class(continue)

```
~BSTNode() { }
E& element() { return it; }  // 返回结点的元素值
void setElement (const E& e) { it = e; }  // 设置结点的元素值
BSTNode* left() const { return lc; } //返回结点的左孩子
void setLeft(BSTNode* b) { lc = b; }  // 设置结点的左孩子
BSTNode* right() const { return rc; } //返回结点的右孩子
void setRight(BSTNode* b) { rc = b; } //设置结点的右孩子
bool isLeaf()     //判断该结点是否为叶子
{ return (lc == NULL) && (rc == NULL); }
};
```

25

25

A typical Pointer-based binary tree



1个根指针几乎就可描述一棵链式二叉树

- ✓ 根指针root指向根结点，
- ✓ 然后由根结点的左右孩子指针又分别指向其左右孩子，依此往下，就描述了整棵树

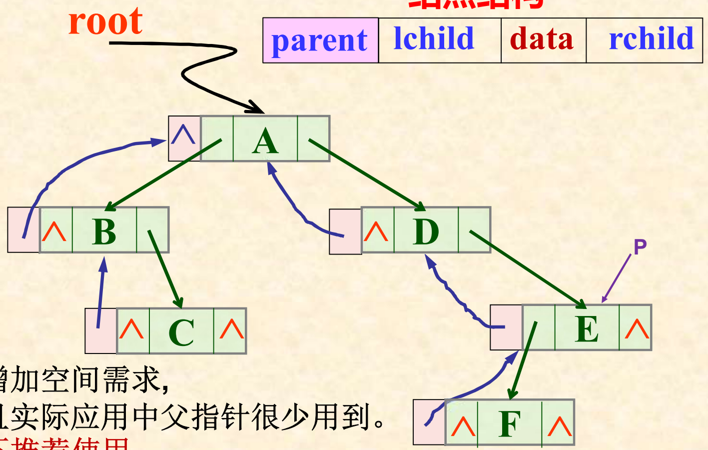
26

26

Add a upward pointer to the node's parent

结点结构

parent	lchild	data	rchild



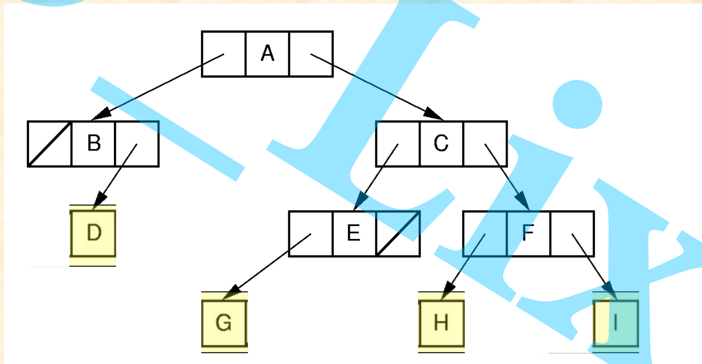
增加空间需求，
且实际应用中父指针很少用到。
不推荐使用

27

可变结构结点 VarBinNode

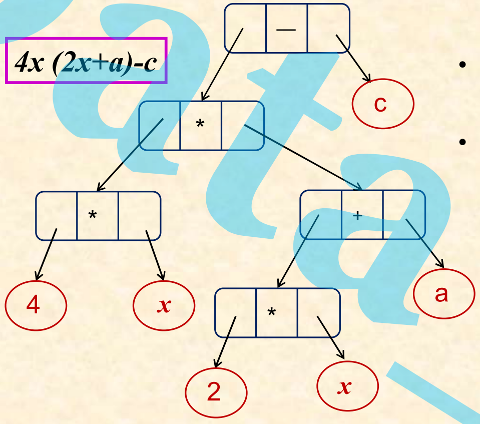
The structures of leaf and internal nodes are different:

leaf: data   internal nodes: lchild | data | rchild



28

可变结构结点 VarBinNode的一个典型应用 ---expression tree/表达式树



- Each leaf is an operand 操作数
- The internal nodes are operators 操作符
- Sub-trees are sub-expressions 子表达式

1. 节省空间
2. 适合应用

29

可变结构结点 VarBinNode

data

Leaf node的基本操作

1. 返回结点值 Val( )
2. 设置结点值 setVal(const E&)
3. 是否叶子: isLeaf()

lchild	data	rchild

Internal node 所涉及的基本操作

1. 返回结点值 Val( )
2. 设置结点值 setVal(const E&)
3. 是否叶子: isLeaf()
4. 返回左孩子 left()
5. 设置左孩子 setLeft(BinNode*)
6. 返回右孩子 right()
7. 设置右孩子 setRight(BinNode*)

30

30

Binary Tree Node —VarBinNode(1)

```
class VarBinNode {   // Abstract base class
public:
   virtual  ~VarBinNode() {}
   virtual bool isLeaf() = 0;
};

class LeafNode : public VarBinNode { // Leaf
private:
  double var;                 // Operand value
public:
  LeafNode( const double& val)
    { var = val; }    // Constructor
  bool isLeaf() { return true; }
  double Val() { return var; }
  void setVal(const double& val)
    { var = val; }
};
```

31

31

Binary Tree Node Class—VarBinNode(2)

```
// Internal node
class IntlNode : public VarBinNode {
private:
  VarBinNode* left;      // Left child
  VarBinNode* right;     // Right child
  char  opx;           // Operator value
public:
  IntlNode(const char& op,
             VarBinNode* l, VarBinNode* r)
    { opx = op; left = l; right = r; }
  bool isLeaf() { return false; }
  VarBinNode* left() { return left; }
  void setLeft(VarBinNode* l) {left = l;}
  varBinNode* right() { return right; }
  void setRight(VarBinNode* r) {right = r;}
  char Val() { return opx; }
  void setVal(char& op) {opx = op; }
};
```

32

32

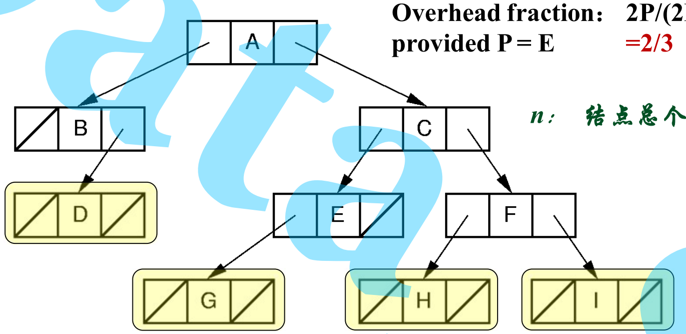
5.3.2 Space Requirements and Overhead analysis(1)

Overhead：存放数据所需空间之外的空间

所需空间：nE+2nP (Overhead)

Overhead fraction： 2P/(2P+E)

provided P = E =2/3

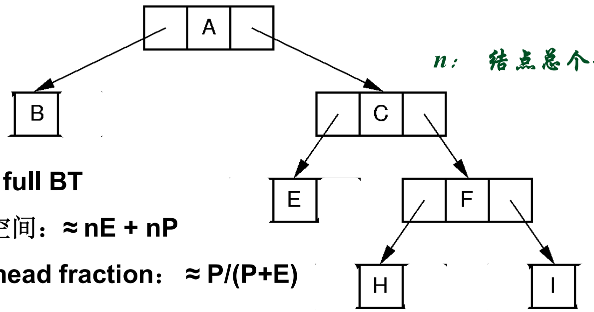


n： 结点总个数

二叉 Node

33

Space Requirements and Overhead analysis(2)



n： 结点总个数

For a full BT

所需空间：≈ nE + nP

Overhead fraction：≈ P/(P+E)

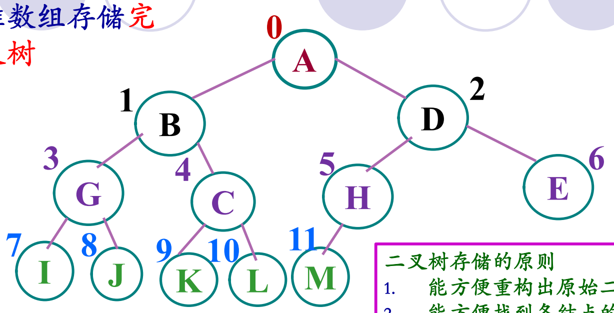
provided P = E = 1/2

`VarBinNode`

34

5.3.3 Array-based implementation for CBT

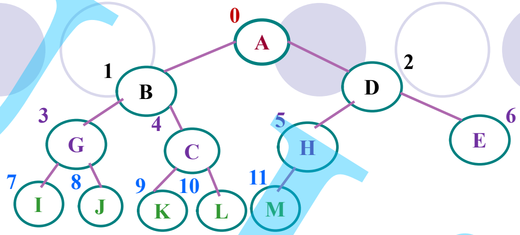
用一维数组存储完全二叉树



二叉树存储的原则
1. 能方便重构出原始二叉树
2. 能方便找到各结点的孩子

0	1	2	3	4	5	6	7	8	9	10	11
A	B	D	G	C	H	E	I	J	K	L	M

35



Position(数组下标)	0	1	2	3	4	5	6	7	8	9	10	11
BT元素	A	B	D	G	C	H	E	I	J	K	L	M
Parent	--	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	--	--	--	--	--	--
Right Child	2	4	6	8	10	--	--	--	--	--	--	--
Left Sibling	--	--	1	--	3	--	5	--	7	--	9	--
Right Sibling	--	2	--	4	--	6	--	8	--	10	--	--

36

For complete binary tree, the position relation can be calculated:

Parent(r) = (r-1)/2	if r > 0 and r < n.
Leftchild(r) = 2r + 1	if 2r + 1 < n.
Rightchild(r) = 2r + 2	if 2r + 2 < n.
Leftsibling(r) = r - 1	if r is even, r > 0, and r < n.
Rightsibling(r) = r + 1	if r is odd and r + 1 < n.

n： CBT中结点总个数

37

37

二叉树存储的原则

1. 能方便重构出原始二叉树
   - ✓ 链式二叉树：
     本身就是树的形式
   - ✓ Array-based CBT
     从数组中第0个元素开始，按层构建，直到最后一个元素
2. 能方便找到各结点的孩子
   - ✓ 链式二叉树：
     从结点的lc，rc指针即可直接找到其左右孩子所在结点
   - ✓ Array-based CBT：设某个元素在数组中的下标为r
     Leftchild(r) = 2r + 1
     Rightchild(r) = 2r + 2

38

38

链式非CBT与基于数组CBT总结

- 非CBT
  - 链式存储
  - 1个根指针—指向二叉树的根结点
  - 结点结构： 二叉结构，可变结构
- CBT
  - 基于数组存储
  - 1个数组和2个整形变量
    - 1个数组：按层存放结点元素值，即数组中每个元素对应一个结点
    - maxSize—数组大小，size—树中结点个数
  - 由任意数组元素下标可求出对应结点在CBT中的具体位置，及其孩子和双亲所对应的结点在数组中的位置

39

39

5.4 Binary Tree Traversal(遍历)

5.4.1 Depth-First Traversal (深度优先遍历)

① Preorder traversal（前序遍历）

② Inorder traversal（中序遍历）

③ Postorder traversal（后序遍历）

5.4.2 Breadth-First Traversal（广度优先遍历）

40

40

Traversals（遍历）

Any process for **visiting** each node **once and only once** in a **predetermined sequence**(预先确定的顺序） is called **traversal**.

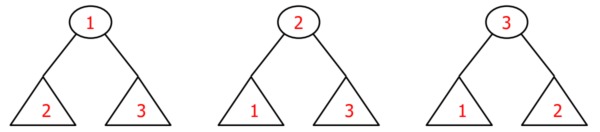
“visting” 的含义可以很广，如：输出结点的信息,比较结点值与某一值的大小关系，修改节点数据等

- **Depth-First（深度优先）Traversal**
  - Preorder（前序） traversal: NLR
  - Postorder（后序） traversal: LRN
  - Inorder（中序） traversal: LNR
- **Breadth-First（广度优先） Traversal**

41

41

Depth-First Traversal

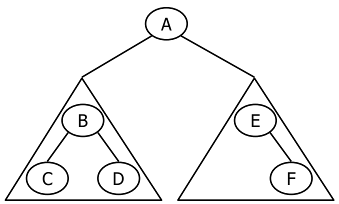


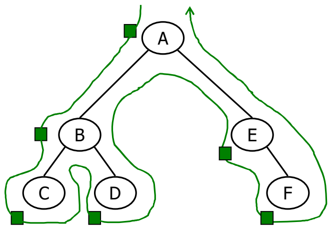
PreOrder(前序) NLR　InOrder(中序) LNR　PostOrder(后序) LRN

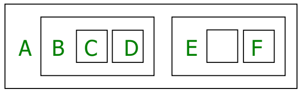
42

42

5.4.1 Preorder traversal（前序遍历）







Walking order

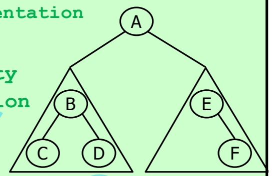
Processing order:
ABCDEF

43

43

PreOrder Traversal c++ code

```
template <class E>      // Good implementation
void preOrder(BSTNode<E>* root) {
  if (root == NULL) return;  // Empty
  visit(root); // Perform some action
  preorder(root->left());
  preorder(root->right());
}
```

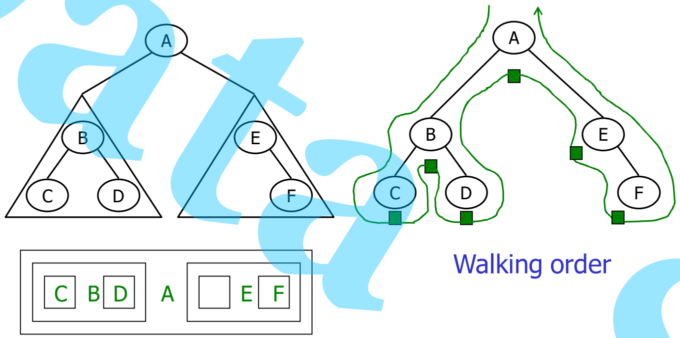



```
template <class Elem>  // Bad implementation
void preOrder2(BSTNode<Elem>* root) {
  visit(root); // Perform some action
  if (root->left() != NULL)
    preorder2(root->left());
  if (root->right() != NULL)
    preorder2(root->right());
}
```

44

44

5.4.2 Inorder traversal（中序遍历）



Processing order

CBDAEF

45

45

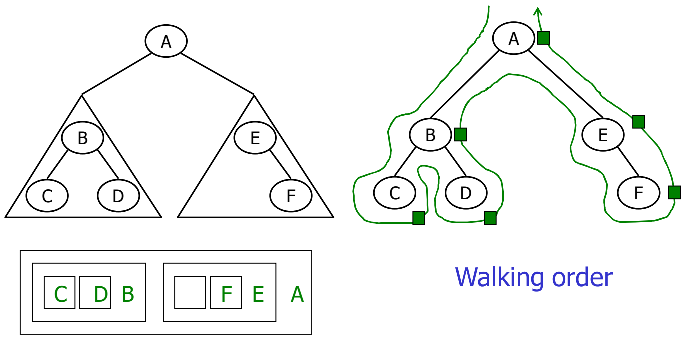
InOrder Traversal c++ code

```
template <class E>
void inOrder(BSTNode<E>* root) {
  if (root == NULL) return;  // Empty
  inorder(root->left());
  visit(root);  // Perform some action
  inorder(root->right());
}
```

46

46

5.4.3 Postorder traversal（后序遍历）



Processing order

CDBFEA

47

47

PostOrder Traversal c++ code

```
template <class E>
void postOrder(BSTNode<E>* root) {
  if (root == NULL) return;  // Empty
  postorder(root->left());
  postorder(root->right());
  visit(root);  // Perform some action
}
```

48

48

由深度优先遍历序列恢复二叉树

- 已知前序序列和中序序列可唯一恢复二叉树

  example: 前序**ABCDEF**， 中序**CBDAEF**

- 已知后序序列和中序序列可唯一恢复二叉树

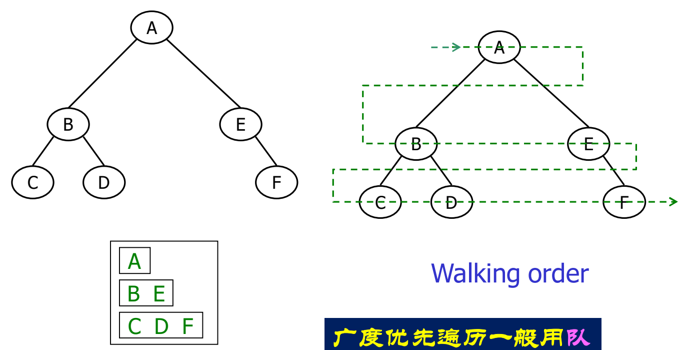
  example: 后序**CDBFEA**， 中序**CBDAEF**

- 已知前序序列和后序序列不可唯一恢复二叉树

49

49

5.4.4 Breadth-First Traversal（广度优先遍历）



Walking order

广度优先遍历一般用队列的方式实现

Processing order

ABECDF

50

50

Breadth-First Traversal pseudocode

Algorithm breadthFirst (BSTNode* root)
1 pointer = root
2 while (pointer not null)
  1 visit (pointer)
  2 if (pointer –> left not null)
    1 enQueue (pointer –> left)
  3 if (pointer –> right not null)
    1 enQueue (pointer –> right)
  4 if (not emptyQueue)
    1 deQueue (pointer)
  5 else
    1 pointer = null
End breadthFirst

有兴趣的同学课后自己编写对应的C++代码

51

51

DFT & BFT实现总结

- 深度优先遍历（DFT）一般用递归(栈)的方式实现
- 广度优先遍历（BFT）一般用队列的方式实现

52

52

本章作业一

- 5.2
- 5.3
- 5.6