Trees

Introduction to Data Structures

Kyuseok Shim

ECE, SNU.

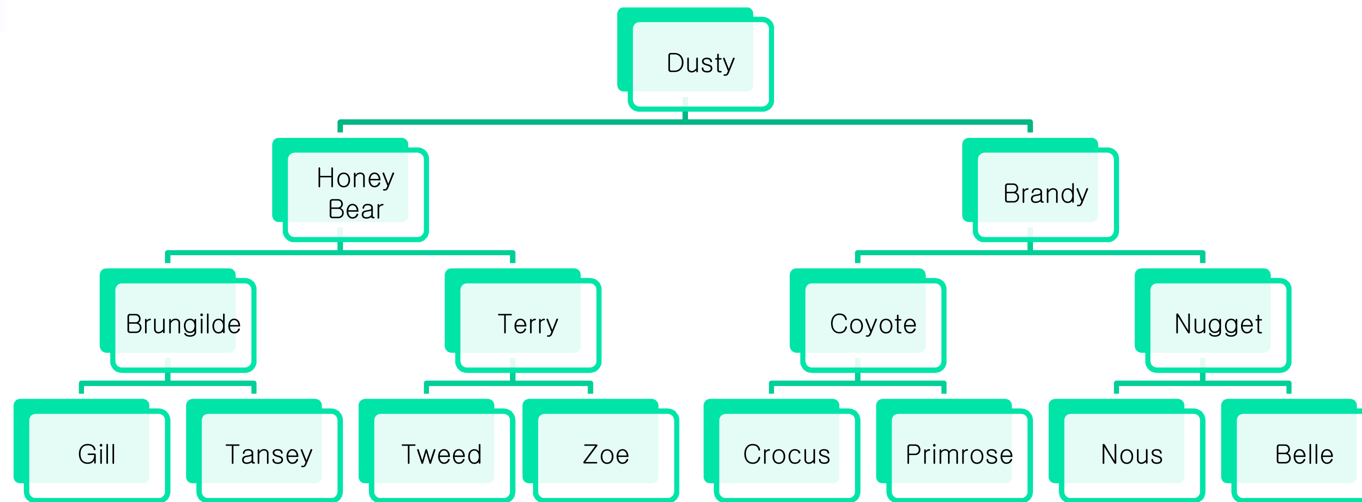
Terminology

- Tree : A finite set of one or more nodes such that
  - There is a specially designated node called the root
  - The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree. $T_1, \ldots, T_n$ are called the subtrees of the root.
- Degree of a node : the number of subtrees of a node
- Leaf (terminal node) : a node that has degree zero
- Internal node (nonterminal node) : the other nodes
- Children, parent, siblings

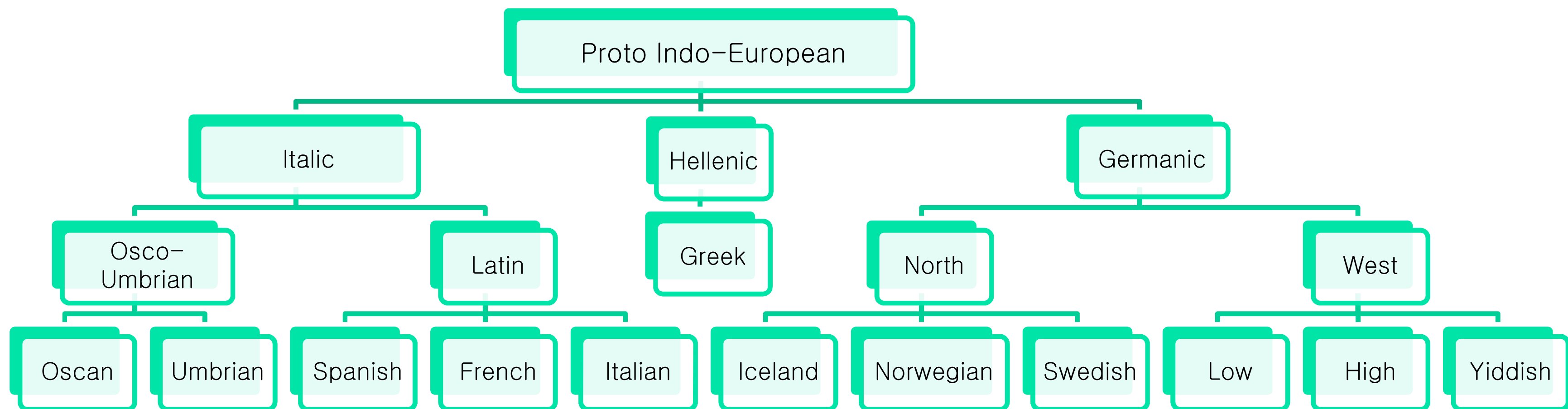
Terminology

- Degree of a tree : the maximum of degree of the nodes in the tree
- Ancestors of a node : all the nodes along the path from the root to that node
- Level of a node : the distance from the root+1
- Height (or depth) of a tree : the maximum level of any node in the tree

Terminology

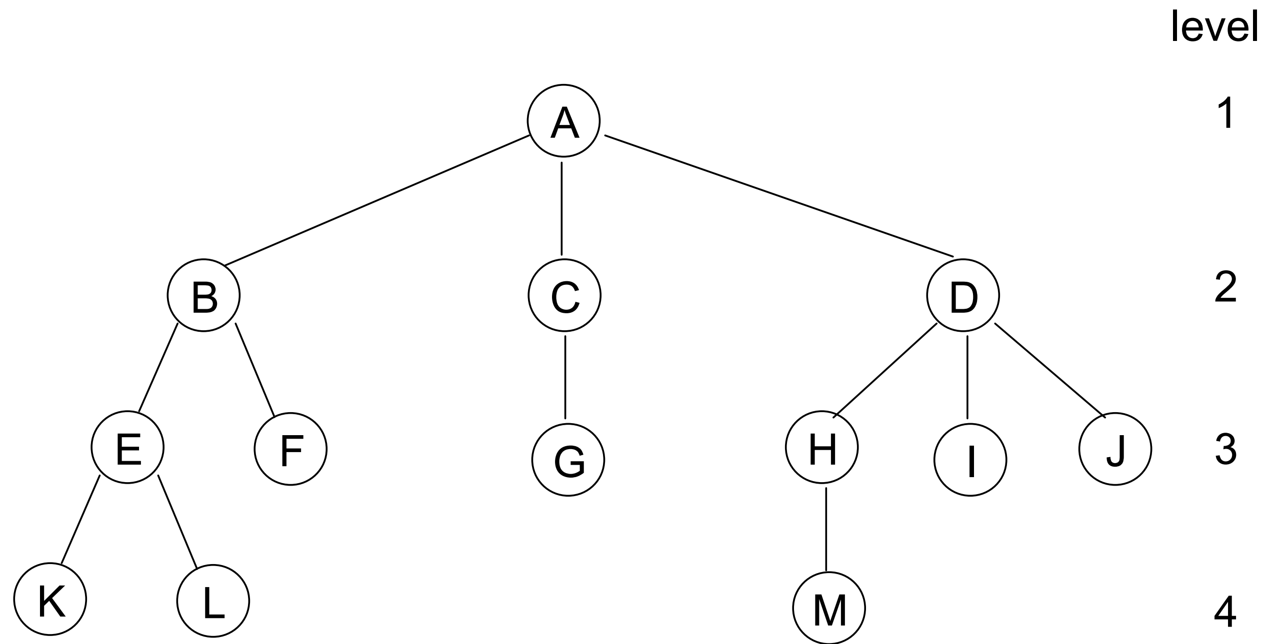


(a) Pedigree



(b) Lineal

Terminology



(A(B(E(K,L),F),C(G),D(H(M),I,J)))

Figure 5.2 : A sample tree

Representation of Trees

- List representation
  - The tree of Figure 5.2
    - (A(B(E(K,L),F),C(G),D(H(M),I,J)))
  - The degree of each node may be different
    - Possible to use memory nodes with a varying number of pointer fields
    - Easier to write algorithms when the node size is fixed

Representation of Trees

- List representation
  - nodes of a fixed size
    - for a tree of degree k

DATA	$CHILD_1$	$CHILD_2$	…	$CHILD_k$

    - Lemma 5.1: If T is a k-ary tree with n nodes, each having a fixed size, then n(k-1)+1 of the nk child fields are 0, $n \geq 1$.
    - Proof: Since each non-zero child field points to a node and there is exactly one pointer to each node other than the root, the number of child fields in a k-ary tree with n nodes is nk. Hence, the number of zero fields is nk – (n-1) = n(k-1)+1

Representation of Trees

- Left child-right sibling representation
  - Node structure

data	
left child	right sibling

  - The left child field of each node points to its leftmost child (if any), and the right sibling field points to its closest right sibling (if any)

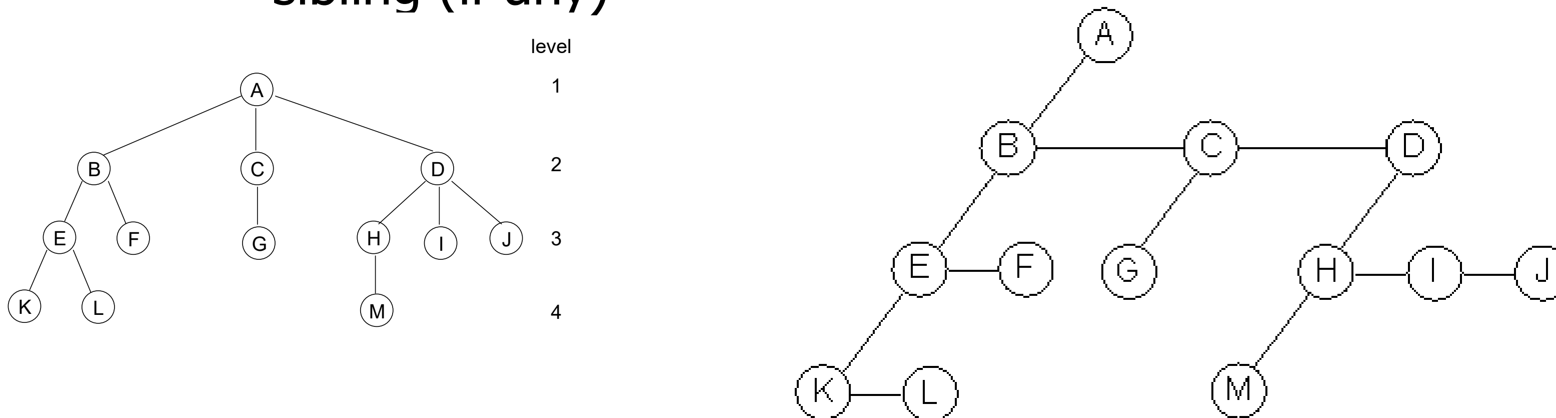


Figure 5.6 : Left child-right sibling representation of the tree of Figure 5.2

Representation of Trees

- Representation as a degree-two tree
  - Rotate the right-sibling pointers clockwise by 45 degrees
  - The two children of a node are referred to as the left and right children

Representation of Trees

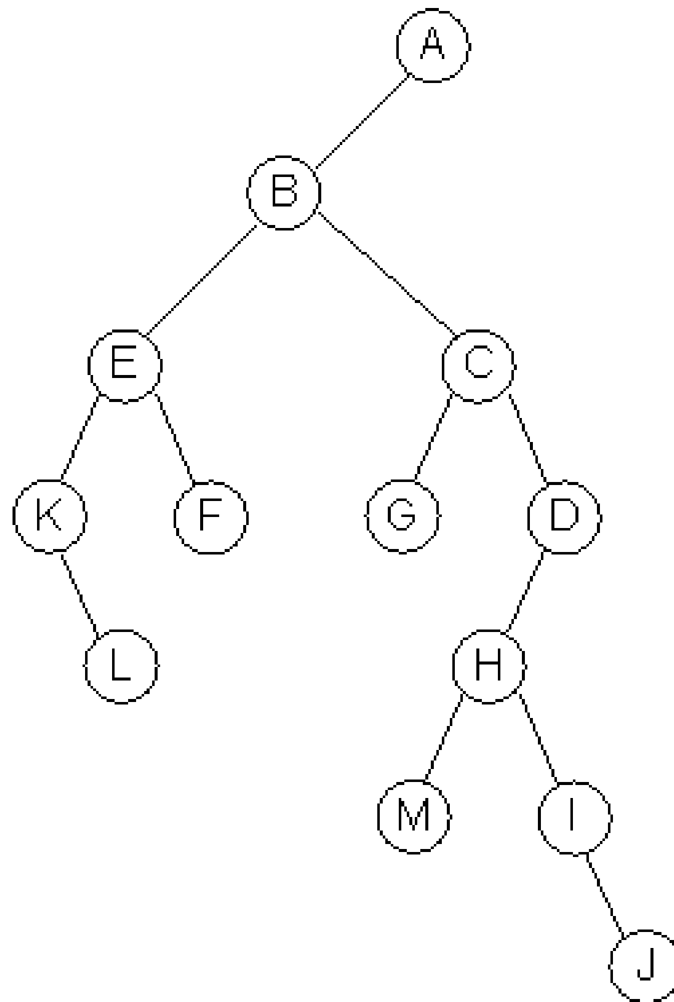


Figure 5.7 : Left child-right child tree representation

Representation of Trees

- Additional examples

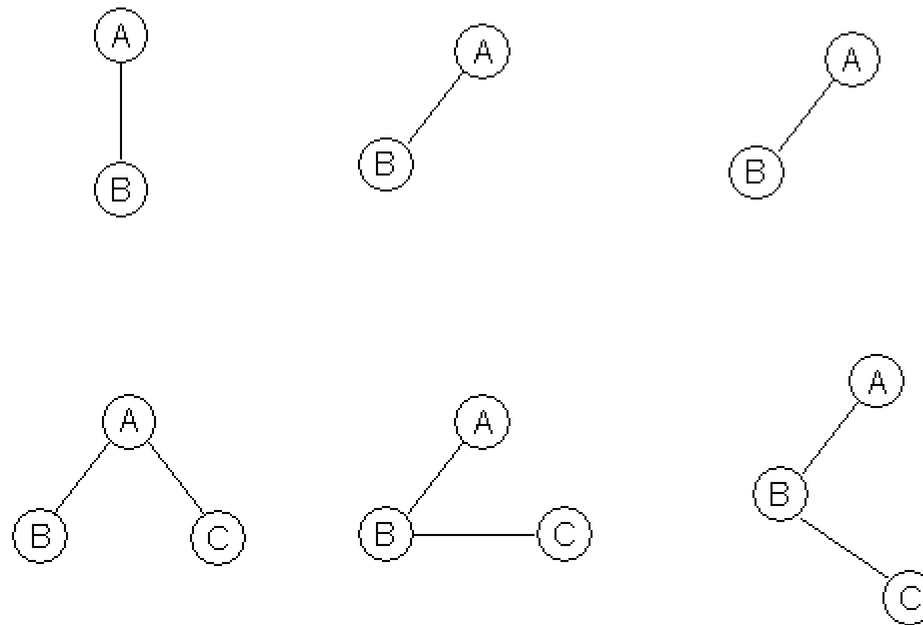


Figure 5.8 : Tree representations

  - Left child-right child tree : binary tree
  - Any tree can be represented as a binary tree

Binary Trees

- A binary tree
  - A finite set of nodes that either is empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree
- Differences between a binary tree and a tree
  - There is an empty binary tree
  - The order of the children is distinguished in a binary tree

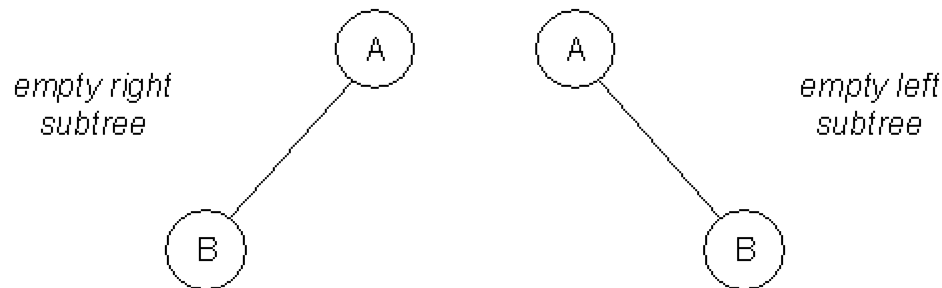


Figure 5.9 : Two different binary trees

Binary Trees

```
template <class T>
class BinaryTree
{ // objects: A finite set of nodes either empty or consisting of a
  //  root node, left BinaryTree and right BinaryTree.
public:
   BinaryTree();
   // creates an empty binary tree

   bool IsEmpty();
   // return true if the binary tree is empty

   BinaryTree(BinaryTree<T>& bt1,  T& item, BinaryTree<T>& bt2);
   // creates a binary tree whose left subtree is bt 1, whose right
   // subtree is bt 2, and whose root node contains item

   BinaryTree<T> LeftSubtree();
   // return the left subtree of *this

   BinaryTree<T> RightSubtree();
   // return the right subtree of *this

   T RootData();
   // return the data in root node of *this
};
-----------------------------------------------------------------
ADT 5.1 : Abstract data type BinaryTree
```

Binary Trees

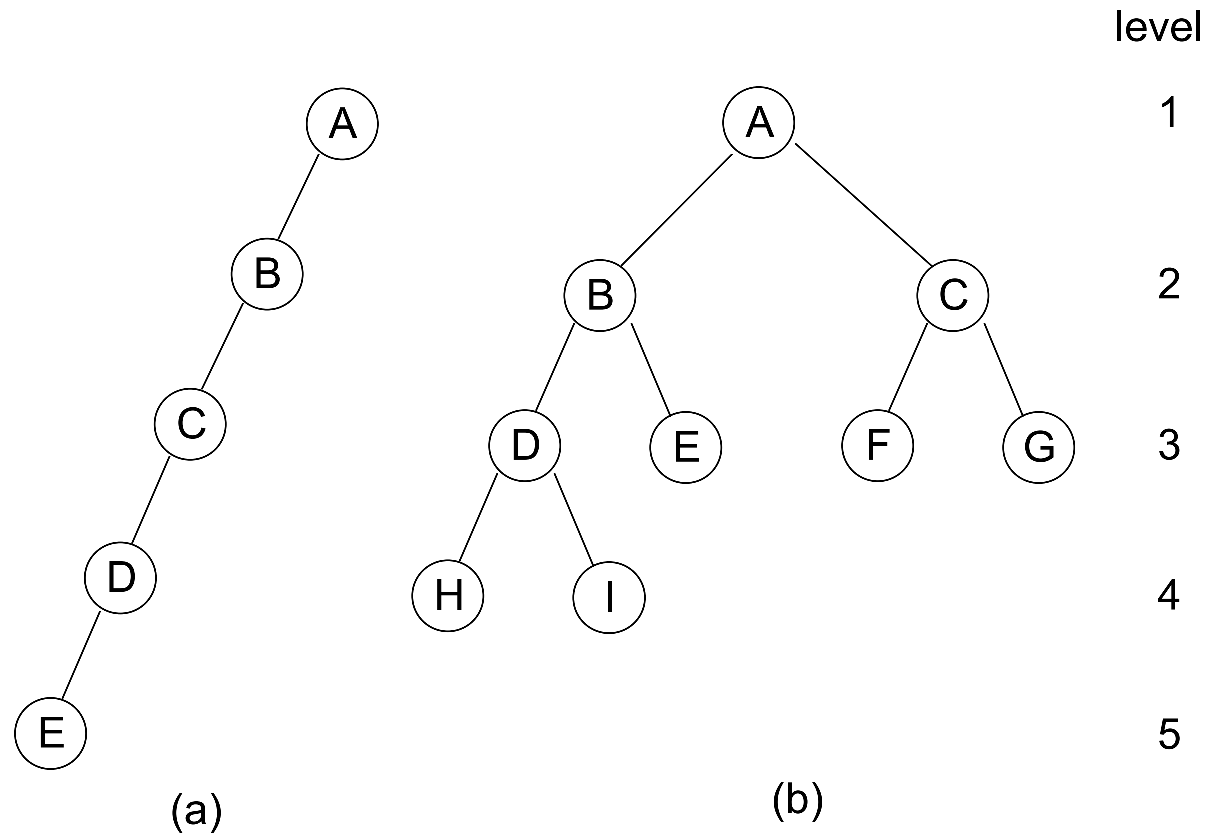


Figure 5.10 : Skewed and complete binary trees

Properties of Binary Trees

- Lemma 5.2 [Maximum number of nodes]:

  (1) The max number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$

  (2) The max number of nodes in a binary tree of depth k is 2^k-1, $k \geq 1$

Properties of Binary Trees

- Proof:
  - (1) The proof is by induction on i.
    - **Induction Base** : The root is the only node on level i = 1.
      Hence, the maximum number of nodes on level i = 1 is $2^{i-1}=2^0=1$
    - **Induction Hypothesis** : Let i be an arbitrary positive integer greater than 1.
      Assume that the maximum number of nodes on level i-1 is 2^{i-2}
    - **Induction Step** : The maximum number of nodes on level i-1 is 2^{i-2}
      by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level i-1, or 2^{i-1}
  - (2) The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=1}^{k}(\max imum\ number\ of\ nodes\ on\ level\ i) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

Properties of Binary Trees

- Lemma 5.3 [Relation between number of leaf nodes and degree-2 nodes]:
- For any non-empty binary tree, T,
  if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0=n_2+1$

Properties of Binary Trees

- Proof : Let n_1 be the number of nodes of degree one and n the total number of nodes. Since all nodes in T are at most of degree two, we have $n=n_0+n_1+n_2$
  If we count the number of branches in a binary tree, we see that every node except the root has a branch leading into it. If B is the number of branches, then $n=B+1$. All branches stem from a node of degree one or two. Thus, $B=n_1+2n_2$. Hence, we obtain $n = B+1 = n_1+2n_2+1$. We get $n_0 = n_2 + 1$
- Def : A full binary tree of depth k
  - A binary tree of depth k having 2^k-1 nodes, $k \geq 0$

Properties of Binary Trees

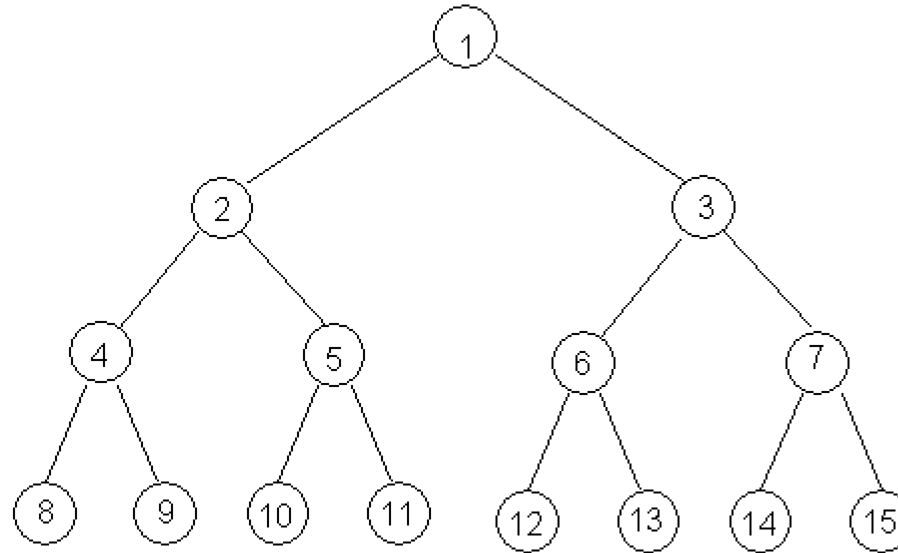


Figure 5.11 : Full binary tree of depth 4 with sequential node numbers

- A binary tree with n nodes and depth k is complete
  - Its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k
- The height of a complete binary tree with n nodes is

$$\lceil \log_2(n+1) \rceil$$

Binary Tree Representation: Array Representations

- Lemma 5.4 : If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \leq i \leq n$, we have
  - (1) parent(i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i=1$, i is at the root and has no parent
  - (2) leftChild(i) is at 2i if $2i \leq n$. If $2i > n$, then i has no left child
  - (3) rightChild(i) is at 2i+1 if $2i+1 \leq n$. If $2i+1 > n$, then i has no right child

Binary Tree Representation: Array Representations

- Proof : We prove (2). (3) is an immediate consequence of (2) and the numbering of nodes on the same level from left to right. (1) follows from (2) and (3). We prove (2) by induction on i.
  For $i=1$, clearly the left child is at 2 unless $2>n$, in which chase i has no left child. Now assume that for all j, $1 \le j \le i$, leftChild(j) is at 2j.
  Then the two nodes immediately preceding leftChild(i+1) are the right and left children of i. The left child is at 2i. Hence, the left child of i+1 is at $2i+2=2(i+1)$ unless $2(i+1)>n$, in which case i+1 has no left child

Array Representations

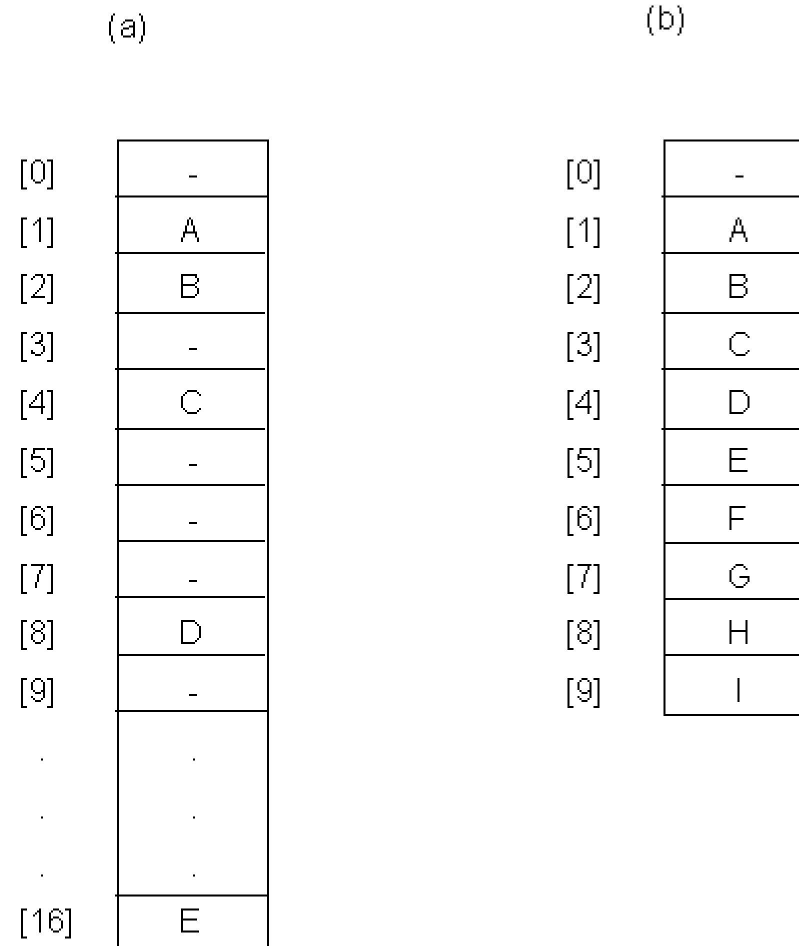


Figure 5.12 : Array representation of the binary trees of Figure 5.10

Linked representation

- Classes to define a tree

```
class Tree;  //forward declaration
class TreeNode {
    friend class Tree;
    private:
        TreeNode *LeftChild;
        char data;
        TreeNode *RightChild;
};
class Tree {
    public:
        // Tree operations
        ...
    private:
        TreeNode *root;
};
```

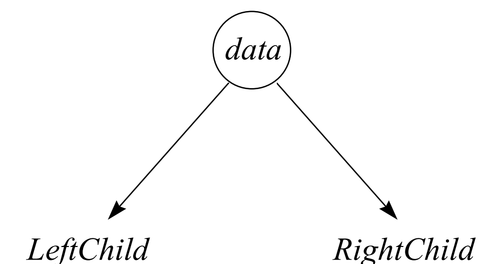


Figure 5.13 : Node representations

Linked representation

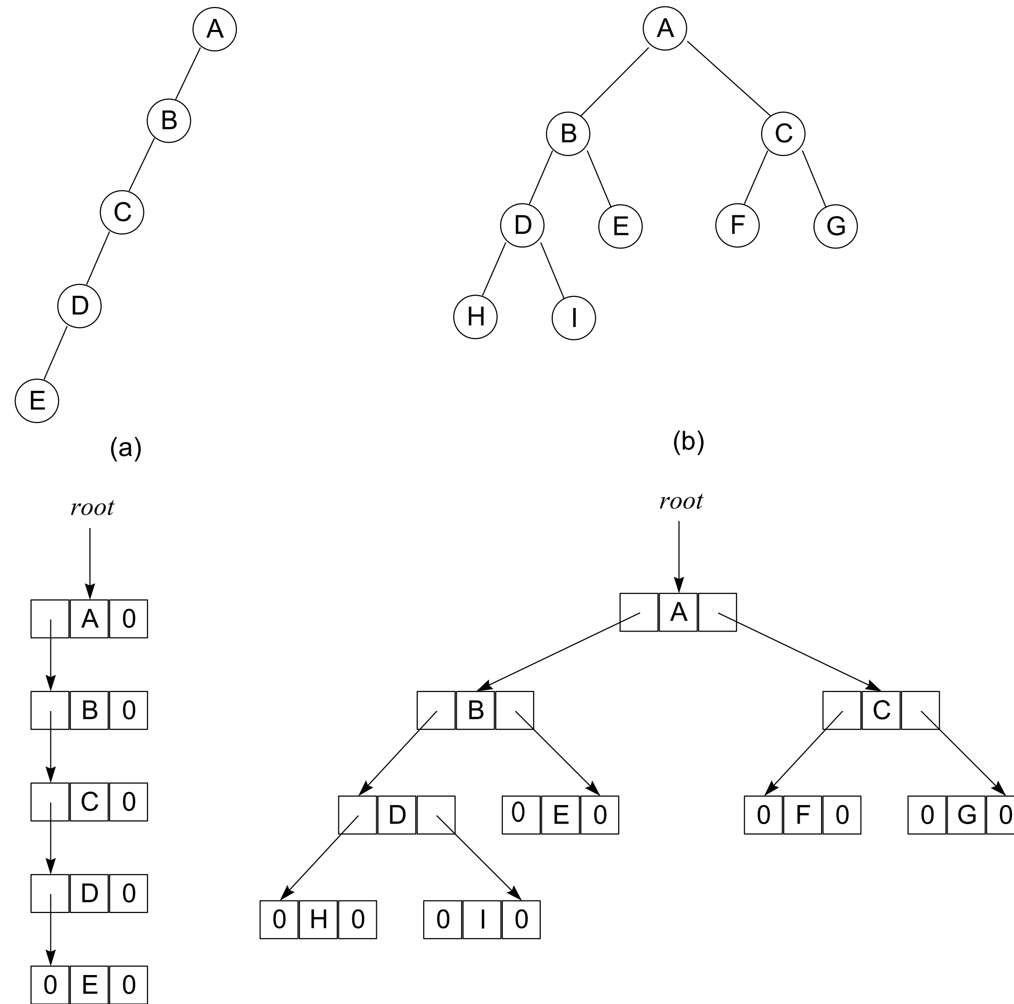


Figure 5.14 : Linked representation for the binary trees of Figure 5.10

Binary Tree Traversal and Tree Iterators

- Tree traversal
  - Visiting each node in the tree exactly once
  - A full traversal produces a linear order for the nodes
- Order of node visit

  L : move left

  V : visit node

  R : move right
  - Possible combinations : LVR, LRV, VLR, VRL, RVL, RLV
  - Traverse left before right
- LVR : Inorder
- LRV : Postorder
- VLR : Preorder

Introduction

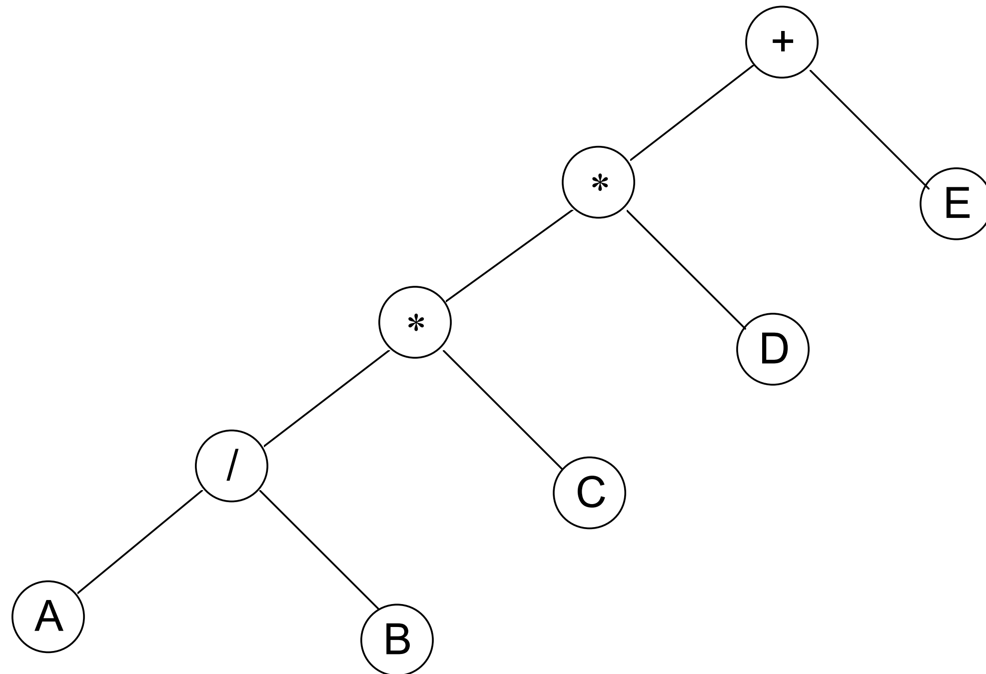
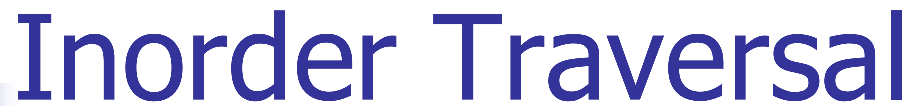


Figure 5.16 : Binary tree with arithmetic expression



Inorder Traversal

- LVR

```
※ Visit(TreeNode<T> *CurrentNode) {
       cout << currentNode->data
   }
```

```
template<class T>
void Tree::inorder()
{ // Driver call workhorse for traversal of entire tree. The driver is
  // declared as a public member function of Tree.
       inorder(root);
}
template<class T>
void Tree<T>::inorder(TreeNode<T> *CurrentNode)
{ // Workhorse traverses the subtree rooted at CurrentNode
  // The workhorse is declared as a private member function of Tree.
        if (CurrentNode) {
              inorder(CurrentNode->leftChild);
              Visit(currentNode);
              inorder(CurrentNode->rightChild);
       }
}
-----------------------------------------------------------------
Program 5.1 : Inorder traversal of a binary tree
```

Inorder Traversal

Call of in*order*	Value in *CurrentNode*	Action	Call of in*order*	Value in *CurrentNode*	Action
Driver	+		10	C	
1	*		11	0	
2	*		10	C	cout<<'C'
3	/		12	0	
4	A		1	*	cout<<'*'
5	0		13	D	
4	A	cout<<'A'	14	0	
6	0		13	D	cout<<'D'
3	/	cout<<'/'	15	0	
7	B		Driver	+	cout<<'+'
8	0		16	E	
7	B	cout<<'B'	17	0	
9	0		16	E	cout<<'E'
2	*	cout<<'*'	18	0	

Figure 5.17 : Trace of Program 5.1

- Output : A/B*C*D+E
  - Infix form of the expression

Preorder Traversal

- VLR

```
template <class T>
void Tree<T>::preorder()
{ // Driver
        preorder(root);
}
template <class T>
void Tree<T>::preorder(TreeNode<T> *CurrentNode)
{ // Workhorse
       if (CurrentNode) {
              Visit(CurrentNode);
              preorder(CurrentNode->leftChild);
              preorder(CurrentNode->rightChild);
       }
}
-------------------------------------------------
Program 5.2 : Preorder traversal of a binary tree
```

· Output : +**/ABCDE
  - Prefix form of the expression

Postorder Traversal

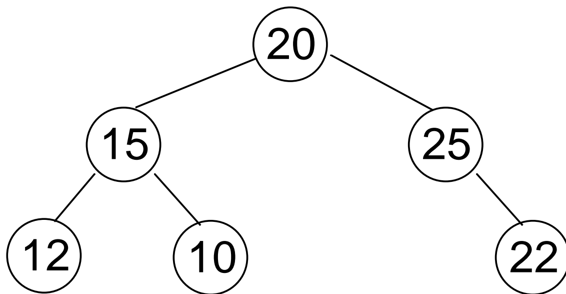
- LRV

```
template <class T>
void Tree<T>::Postorder()
{ // Driver
        postorder(root);
 }
template <class T>
void Tree::postorder(TreeNode *CurrentNode)
{ // Workhorse
        if (CurrentNode) {
             postorder(CurrentNode->LeftChild);
             postorder(CurrentNode->RightChild);
             Visit(currentNode);
        }
}
---------------------------------------------------------------
Program 5.3 : Postorder traversal of a binary tree
```

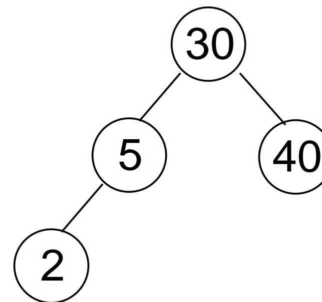
· Output : AB/C*D*E+
– Postfix form of the expression

Binary Search Trees: Definition

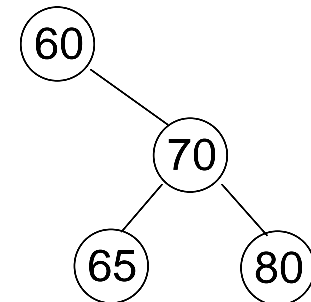
- Binary tree which may be empty
  - If not empty
    - (1) every element has a distinct key
    - (2) keys in left subtree < root key
    - (3) keys in right subtree > root key
    - (4) left and right subtrees are also binary search trees



(a)



(b)



(c)

Figure 5.28 : Binary trees

Dictionary

```
template <class K, class E>
Class Dictionary {
public:
    virtual bool IsEmpty() const = 0;
        // return true iff the dictionary is empty
    virtual pair<K, E> *Get(const K&) const =0;
        // return pointer to the pair with specified key; return 0 if no such pair
    virtual void Insert(const pari<K, E> &) = 0;
        // insert the given pair; if key is a duplicate update associated element
    virtual void Delete(const K&) = 0;
        // delete pair with specified key
}
```

ADT 5.3 : A dictionary

Tree Structure

```
template <class T> class Tree; // forward declaration

template <class T>
class TreeNode {
friend class Tree<T>;
private:
        T data;
        TreeNode<T> *leftChild;
        TreeNode<T> *rightChild;
};
template <class T>
class Tree{
public:
      // Tree operation
private:
      TreeNode<T> *root;
};
```

Searching Binary Search Tree

- Recursive search by key value
  - definition of binary search tree is recursive
  - key(element) = x
    - x=root key : element=root
    - x<root key : search left subtree
    - x>root key : search right subtree

Searching Binary Search Tree

```
template <class K, class E> // Driver
pair<K, E>* BST<K, E>::Get(const K& k)
{ // Search the binary search tree (*this) for a pair with key k
   // If such a pair is found, return a pointer to this pair; otherwise, return 0
            return Get(root, k);
}

template <class K, class E> // Workhorse
pair<K, E>* BST<K, E>::Get(TreeNode<pair<K, E> >* p, const K& k)
{
            if(!p) return 0;
            if(k<p->data.first) return Get(p->leftChild, k);
            if(k>p->data.first) return Get(p->rightChild, k);
            return &p->data;
}
---------------------------------------------------------------
```

Program 5.18 : Recursive search of a binary search tree

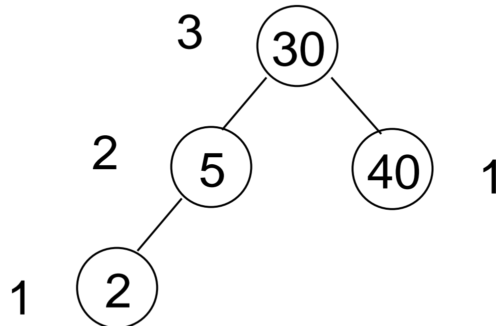
Searching Binary Search Tree

```
template <class K, class E> // Driver
pair<K, E>* BST<K, E>::Get(const K& k)
{
        TreeNode<pair<K,E> > *currentNode = root;
        while(currentNode)
        {
                if (k<currentNode->data.first)
                        currentNode = currentNode->leftChild;
                else if ( k>currentNode->data.first)
                        currentNode = currentNode->rightChild;
                else return &currentNode->data;
        }
        //no matching pair
        return 0;
}
-----------------------------------------------------------------
```

Program 5.19 : Iterative search of a binary search tree

Searching Binary Search Tree

- Search by rank
  - node needs LeftSize field
  - LeftSize=1 + #elements in left subtree



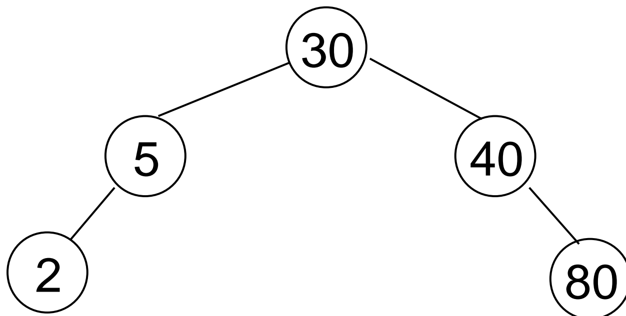
Searching Binary Search Tree

```
template <class K, class E> // search by rank
pair<K,E>* BST<K,E>::RankGet(int r)
{ // Search the binary search tree for the rth smallest pair
        TreeNode<pair<K,E> > *currentNode = root;
        while (currentNode)
                if(r<currentNode->leftSize)
                        currentNode = currentNode->leftChild;
                else if (r>currentNode->leftSize)
                {
                        r -= currentNode->leftSize;
                        currentNode = currentNode->rightChild;
                }
                else return &currentNode->data;
        return 0;
}
-----------------------------------------------------------
```

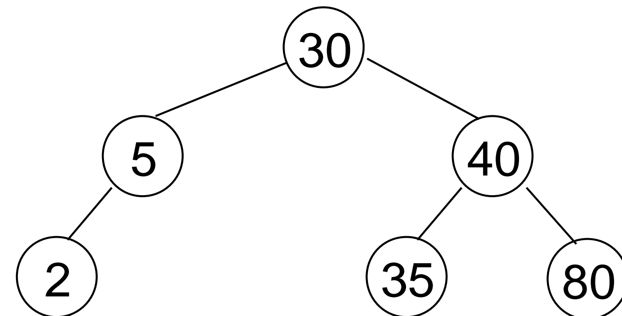
Program 5.20 : Searching a binary search tree by rank

Insertion into Binary Search Tree

- New element x
  - Search x in the tree
    - success : x is in the tree
    - fail : insert x at the point the search terminated



(a) Insert 80



(b) Insert 35

Figure 5.29 : Inserting into a binary search tree

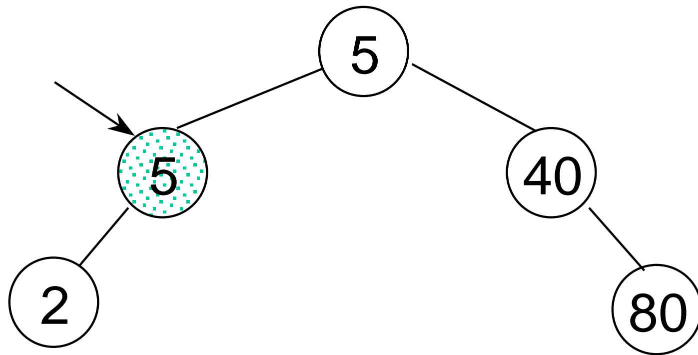
Insertion into Binary Search Tree

```
template <class K, class E>
void BST<K, E>::Insert(const pair<K, E>& thePair)
{ // Insert thePair into the binary search tree.
        // search for thePair.first, pp is parent of p
        TreeNode<pair<K, E> > *p = root, *pp = 0;
        while(p) {
                pp = p;
                if (thePair.first < p->data.first) p = p->leftChild;
                else if(thePair.first > p->data.first) p = p->rightChild;
                else // duplicate, update associated element
                    { p->data.second = thePair.second; return; }
        }
        //perform insertion
        p = new TreeNode<pair<K, E> >(thePair);
        if(root) // tree not empty
                if (thePair.first<pp->data.first) pp->leftChild=p;
                else pp->rightChild = p;
        else root = p;
}
-----------------------------------------------------------
Program 5.21 : Insertion into a binary search tree
```

Deletion from Binary Search Tree

- Leaf node
  - Corresponding child field of its parent is set to 0
  - The node is disposed
- Nonleaf node with one child
  - The node is disposed
  - Child takes the place of the node
- Nonleaf node with two children
  - Node is replaced by either
    - The largest node in its left subtree
    - The smallest node in its right subtree
  - Delete the replacing node from the subtree

Deletion from Binary Search Tree



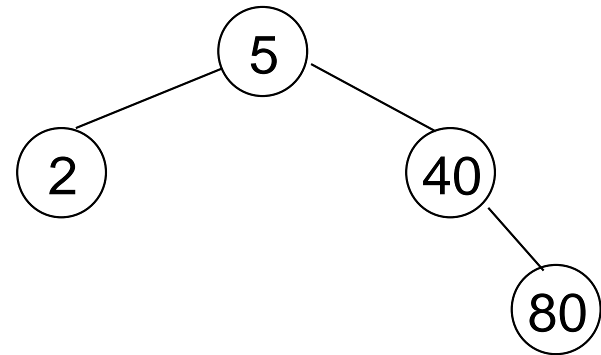


Figure 5.30 : Deletion from a binary search tree

Height of Binary Search Tree

- Height of BST with n nodes
  - Worst-case : n
  - Average : O(log n)
- Balanced search trees
  - Worst-case height : O(log n)
- Some perform search, insert, delete in O(h)