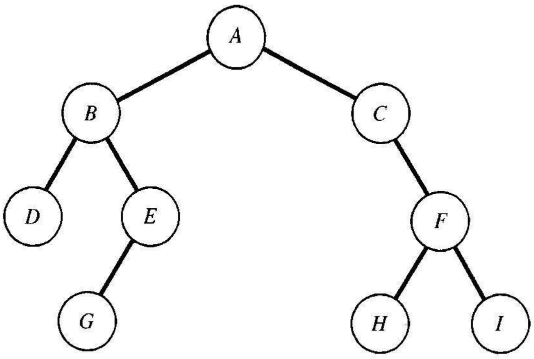
Binary Tree

A binary tree is a finite set of elements that is either empty or is partitioned into 3 disjoint subsets:

- a single element called the root
- two subsets which are themselves binary tree, called the left subtree and right subtree of the root

Each element of a binary tree is called a *node* of the tree.

An example binary tree:



Parent-child (father-son) relation

- The root of the subtree of node x are the children of x
- x is the parent of its children
- Two nodes are siblings (brothers) if they are left and right child of the same parent.
- In the above example, A is the parent of B and C.

Ancestor-descendant relation

- A simple path is a sequence of nodes n_1, n_2, …, n_k such that the nodes are all distinct and there is an edge between each pair of nodes (n_1, n_2), (n_2,n_3), …, (n_{k-1}, n_k)
- The nodes along the simple path from the root to node x are the ancestors of x.
- The descendants of a node x are the nodes in the subtrees of x.

Degree of a node = number of subtrees of a node

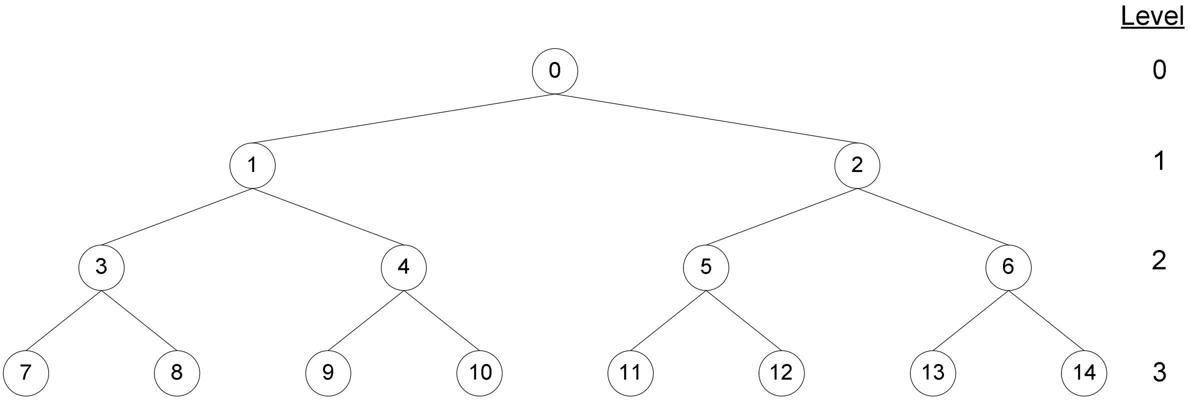
Nodes that have degree zero are called leaf nodes (terminal nodes)

Nodes that are not leaf nodes are called non-leaf nodes (non-terminal nodes or internal nodes)

If every internal node in a binary tree has nonempty left and right subtrees, the tree is termed strictly binary tree.

Level of a node

- root is at level 0 (some books define the root at level 1)
- if a node is at level i, then its children are at level i+1



Numbering of nodes of a *complete binary tree*

- root is labelled 0
- nodes on a level are numbered successively from left to right

1. The depth (or height) of a binary tree is the maximum level of any leaf in the tree (this is equal to the length of the longest path, i.e. number of branches from the root to any leaf).

   According to this definition, depth of a tree with only 1 node is equal to zero.

2. In some textbook, the height of a binary tree is defined to be the number of nodes on the longest path from the root to a leaf.

   According to this definition, the height of a tree with only 1 node is equal to 1. (This definition is used in the study of height-balanced tree).

The maximum number of nodes in a binary tree of depth k (definition 1) is $2^{k+1} - 1$.

A binary tree with n nodes and depth d is *complete* if $n = 2^{k+1} - 1$.

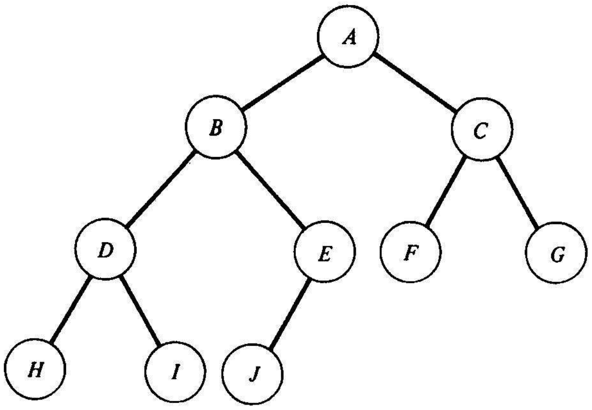
A binary tree with n nodes and depth d is *almost complete* iff its nodes correspond to the nodes which are numbered 0 to n–1 in the complete binary tree.

Some books use the term *full binary tree* for complete binary tree, and *complete binary tree* to mean almost complete binary tree.

Array representation of an almost complete binary tree with n nodes

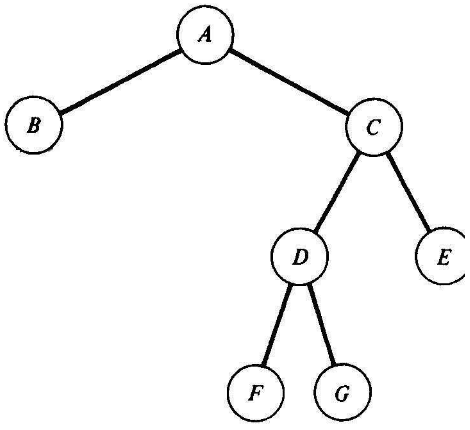
The nodes are numbered in level order.

The root is assigned the label 0 (nodes are labelled from 0 to n-1)
(a) parent(i) is at $\lfloor (i-1)/2 \rfloor$ if i $\neq$0. When i = 0, i is the root and has no parent.
(b) Lchild(i) is at 2i+1 if 2i+1 < n. If 2i+1 ≥ n, then i has no left child.
(c) Rchild(i) is at 2i+2 if 2i+2 < n. If 2i+2 ≥ n, then i has no right child.



0	1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I	J

In general, array representation of binary tree is not memory efficient.



0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C			D	E					F	G

Linked representation of binary tree

```
template<class Type>
struct treeNode
{
   Type info;
   treeNode<Type> *left, *right;
};
```

Traversal of binary trees

To visit systemically the nodes and process each of them in a specific order.

Four basic traversal orders

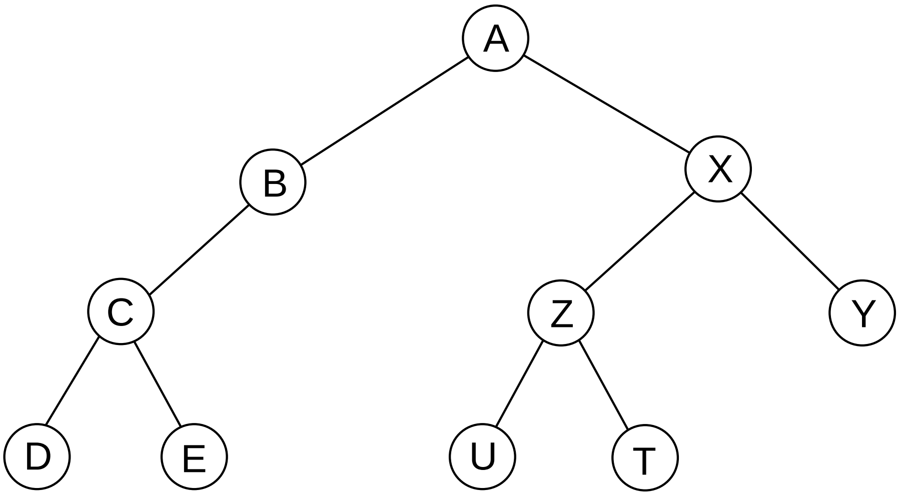
1. Preorder
   - visit the root
   - visit the left subtree in preorder
   - visit the right subtree in preorder

2. Inorder
   - visit the left subtree in inorder
   - visit the root
   - visit the right subtree in inorder

3. Postorder
   - visit the left subtree in postorder
   - visit the right subtree in postorder
   - visit the root

4. Level order
   - visit the nodes level by level starting from the root, and from left to right on each level

Example:



Preorder traversal:

Inorder traversal:

Postorder traversal:

Level order traversal:

Recursive algorithms to traverse a binary tree and print the node label

```
template<class Type>
void preorder(treeNode<Type> *p)
{
   if (p != nullptr)
   {
      cout << p->info << “ ”;  //visit the node
      preorder(p->left);
      preorder(p->right);
   }
}
```

```
template<class Type>
void inorder(treeNode<Type> *p)
{
   if (p != nullptr)
   {
      inorder(p->left);
      cout << p->info << “ ”;  //visit the node
      inorder(p->right);
   }
}
```

```
template<class Type>
void postorder(treeNode<Type> *p)
{
   if (p != nullptr)
   {
      postorder(p->left);
      postorder(p->right);
      cout << p->info << “ ”;  //visit the node
   }
}
```

Algorithm to traverse a binary tree in level order

```
#include <queue>

template<class Type>
void levelTrav(treeNode<Type> *tree)
{
   queue<treeNode<Type>*> Q;

   if (tree != nullptr)
      Q.push(tree);

   while (!Q.empty()) //there are nodes not yet visited
   {
      treeNode<Type>* p = Q.front();
      Q.pop();

      cout << p->info << “ ”;

      if (p->left != nullptr)
         Q.push(p->left);

      if (p->right != nullptr)
         Q.push(p->right);
   }
}
```

Non-recursive inorder traversal using a stack

```
#include <stack>

template<class Type>
void traverseLeft(treeNode<Type> *p,
                  stack<treeNode<Type>*>& S)
{
   while (p != nullptr)
   {
      S.push(p);
      p = p->left;
   }
}

template<class Type>
void inorder_2(treeNode<Type> *tree)
{
   stack<treeNode<Type>*> S;

   traverseLeft(tree, S);

   while (!S.empty()) //there are nodes not yet visited
   {
      treeNode<Type> *p = S.top();
      S.pop();

      cout << p->info << “ ”;
      traverseLeft(p->right, S);
   }
}
```

Algorithm to determine the height of a binary tree

```
template<class Type>
int height(const treeNode<Type> *tree)
{
   if (tree == nullptr)
      return 0;

   if ((tree->left == nullptr) && (tree->right == nullptr))
      return 0; // return 1 (if definition 2 is used)
                // or simply delete this if-statement

   int HL = height(tree->left);
   int HR = height(tree->right);

   if (HL > HR)
      return 1+HL;
   else
      return 1+HR;
}
```

Algorithm to count the number of nodes

```
template<class Type>
int nodeCount(const treeNode<Type> *tree)
{
  if (tree == nullptr)
    return 0;

  return 1 + nodeCount(tree->left) + nodeCount(tree->right);
}
```

Algorithm to count the number of leaf nodes

```
template<class Type>
int leafCount(const treeNode<Type> *tree)
{
  if (tree == nullptr)
    return 0;

  if ((tree->left == nullptr) && (tree->right == nullptr))
    return 1;
  else
    return leafCount(tree->left) + leafCount(tree->right);
}
```

Algorithm to copy a binary tree

```
template<class Type>
void copyTree(treeNode<Type>*& newTree,
              const treeNode<Type> *other)
{
   if (other == nullptr)
      newTree = nullptr;
   else
   {
      newTree = new treeNode<Type>;
      newTree->info = other->info;

      //copy the left subtree
      copyTree(newTree->left, other->left);

      //copy the right subtree
      copyTree(newTree->right, other->right);
   }
}
```

```
//alternative version

template<class Type>
treeNode<Type>* copyTree_2(const treeNode<Type> *other)
{
   if (other == nullptr)
      return nullptr;

   treeNode<Type> *newTree = new treeNode<Type>;
   newTree->info = other->info;
   newTree->left = copyTree_2(other->left);
   newTree->right = copyTree_2(other->right);

   return newTree;
}
```

Algorithm to compare two binary trees

```
template<class Type>
bool equal(const treeNode<Type> *t1, const treeNode<Type> *t2)
{
  if ((t1 == nullptr) && (t2 == nullptr))
    return true;

  if (t1 != nullptr && t2 != nullptr)
     return (t1->info == t2->info) &&
            equal(t1->left, t2->left)&&
            equal(t1->right, t2->right);
  else
     return false;
}
```

Algorithm to print a binary tree in a table format

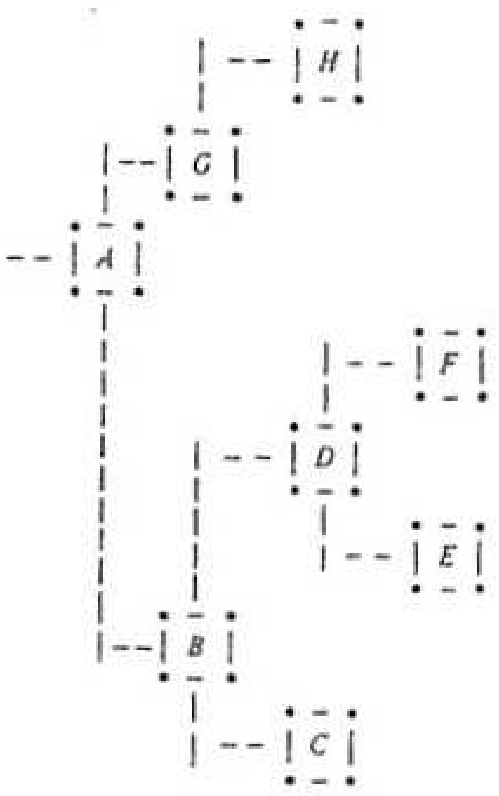
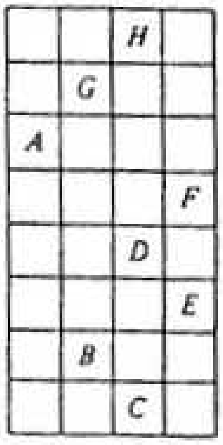


Figure 5.16
A "computer printed" version of Figure 5.5(a)



		H	
	G		
A			
			F
		D	
			E
	B		
		C	

Figure 5.17
A simplified version of Figure 5.16

```
#include <iomanip> //setw(), set width

template<class Type>
void printTree(const treeNode<Type> *p, int indent)
{
    if (p != nullptr)
    {
       //print right subtree, root, and then left subtree

       printTree(p->right, indent+3);
       cout << setw(indent) << p->info << endl;
       printTree(p->left, indent+3);
    }
}
```

Reconstruction of Binary Tree from its Preorder and Inorder sequences.

To simplify the discussion, assume all the node labels are distinct

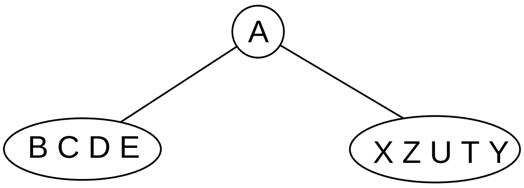
Preorder sequence : A B C D E X Z U T Y

Inorder sequence: D C E B A U Z T X Y

From Preorder, we know that A is the root.
From Inorder, we know that BCDE are in the left subtree and XZUTY are in the right subtree

Preorder **A** B C D E *X Z U T Y*

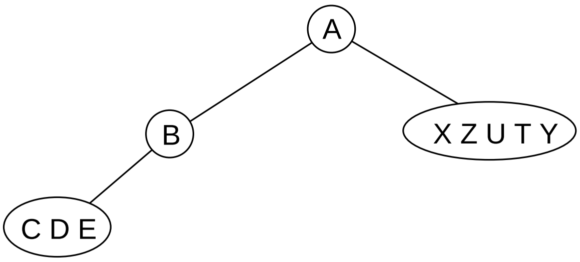
Inorder D C E B **A** *U Z T X Y*



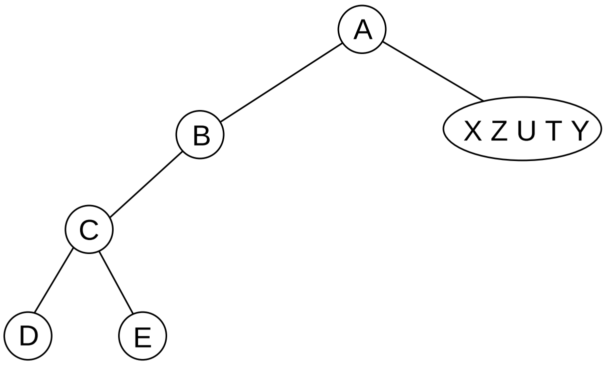
Consider the left subtree BCDE:
From preorder, B is the root;
From inorder, CDE are in the left subtree of B

Preorder **A** **B** C D E *X Z U T Y*

Inorder D C E **B** **A** *U Z T X Y*



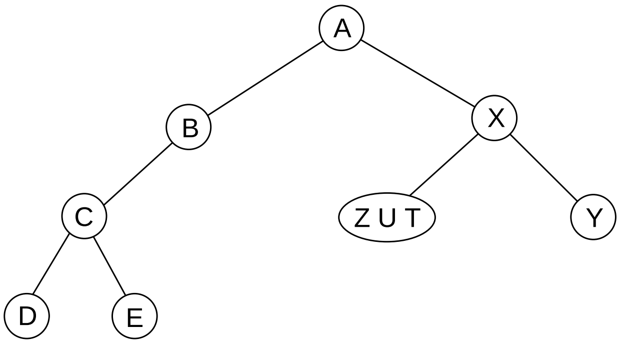
Consider the left subtree CDE:
C is the root; D is in the left subtree of C and E is in the right subtree of C



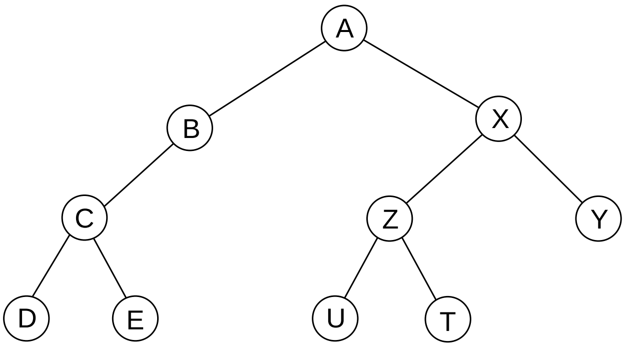
Consider the right subtree XZUTY:
X is the root; ZUT are in the left subtree of X; Y is in the right subtree of X;

Preorder *A B C D E* $\overline{\textbf{X} \text{ Z U T Y}}$

Inorder *D C E B A* $\overline{\text{U Z T } \textbf{X} \text{ Y}}$



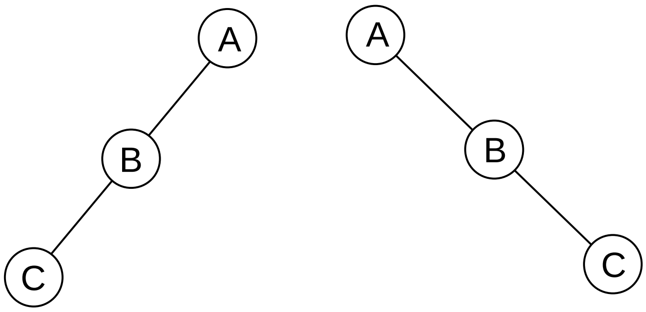
And the final result is



A binary tree may NOT be uniquely defined by its preorder and postorder sequences.

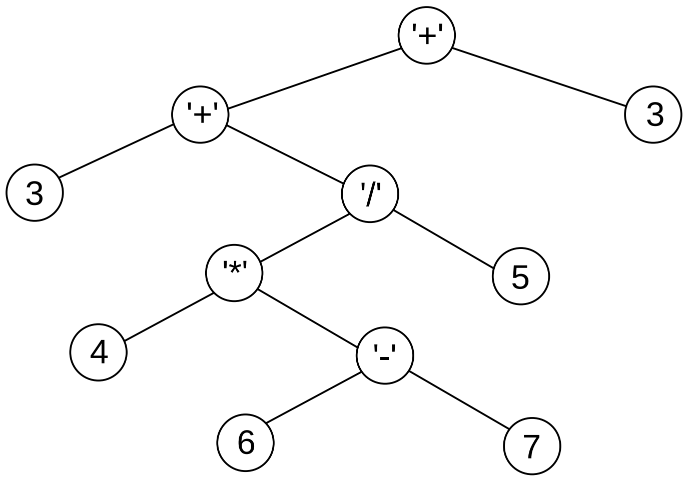
Example:

Preorder sequence:	ABC
Postorder sequence:	CBA



2 different binary trees can give you the same preorder and postorder sequences.

Representation of arithmetic expression using a binary tree (*expression tree*)



Binary tree representation of 3+4*(6–7)/5+3

```
#define operand 0
#define operator 1

struct infoRecord
{
   char dataType;
   union
   {
     char opr;
     double val;
   };  // store either opr or val, not both
}

double evalExprTree(treeNode<infoRecord> *tree)
/* Precondition: the expression tree is nonempty and has no
   syntax error.
   The algorithm is based on postorder traversal. */
{
  if (tree->info.dataType == operand)
     return tree->info.val;
  else
  {
     double d1 = evalExprTree(tree->left);
     double d2 = evalExprTree(tree->right);
     char symb = tree->info.opr;

     return evaluate(symb, d1, d2);
  }
}
```

Huffman Code

- we have an alphabet of *n* symbols
- by assigning a bit-string code to each symbol of the alphabet, a message can be encoded by concatenating the individual codes of the symbols making up the message

For example,

Symbol	Code
A	00
B	01
C	10
D	11

The code for the message ABCACADA would be 0001100010001100, which requires 16 bits.

To reduce the length of the encoded message, we can use variable-length code.

- Frequent symbols are assigned shorter codes, less frequent symbols are assigned longer codes.
- The code for each symbol cannot be a prefix of the code for another symbol.

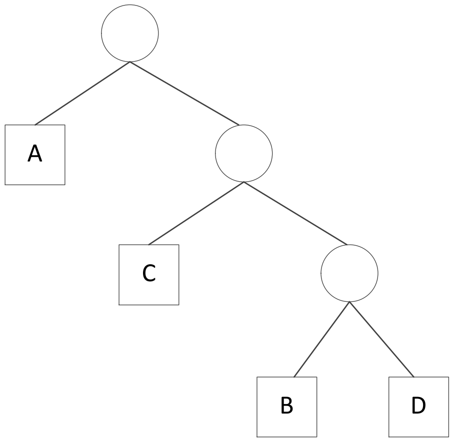
For example,

Symbol	Code
A	0
B	110
C	10
D	111

The code for the message ABCACADA would become 01101001001110, which requires 14 bits.

If the frequency of the characters within a message is known a priori, then an optimal encoding that minimizes the total length of the encoded message can be determined using the Huffman algorithm.

A binary tree is used to construct/decode Huffman code. The binary tree is known as the Huffman tree.



Huffman tree of the above example.

To decode Huffman code, we interpret a ‘0’ as a left branch and a ‘1’ as a right branch.

```
void decode(const treeNode<char> *root, const char *code,
            char *msg)
{
   // Precondition: User is responsible for creating
   // sufficient storage space for the decoded message.
   // code[] is an array of {‘0’, ‘1’}.
   // In real-life, code may be a bit-vector.

   treeNode<char> *p = root;
   int k = 0;

   for (int i = 0; code[i] != ‘\0’; i++)
   {
      if (code[i] == ‘0’)
         p = p->left;
      else
         p = p->right;

      if (p->left == nullptr && p->right == nullptr)
      {
         msg[k++] = p->info;
         p = root;
      }
   }
   msg[k] = ‘\0’;
}
```

Algorithm to construct the Huffman tree

Inputs to the algorithm: the set of symbols and their weights.

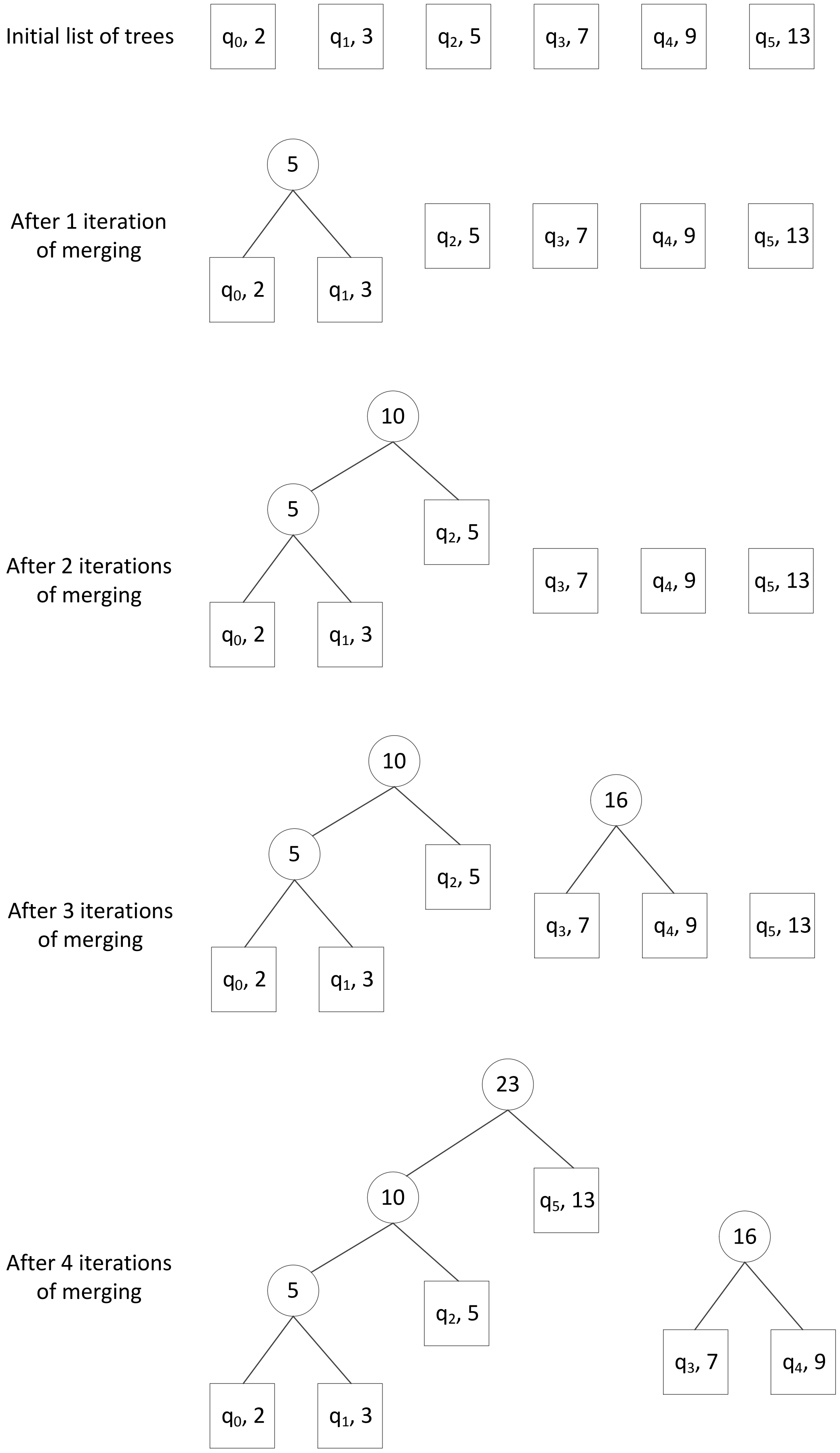
```
struct symbRec
{
   int weight;
   char symbol;
};
```

Suppose there are *n* symbols, *n* binary trees corresponding to the *n* symbols are created, each consists of one node.

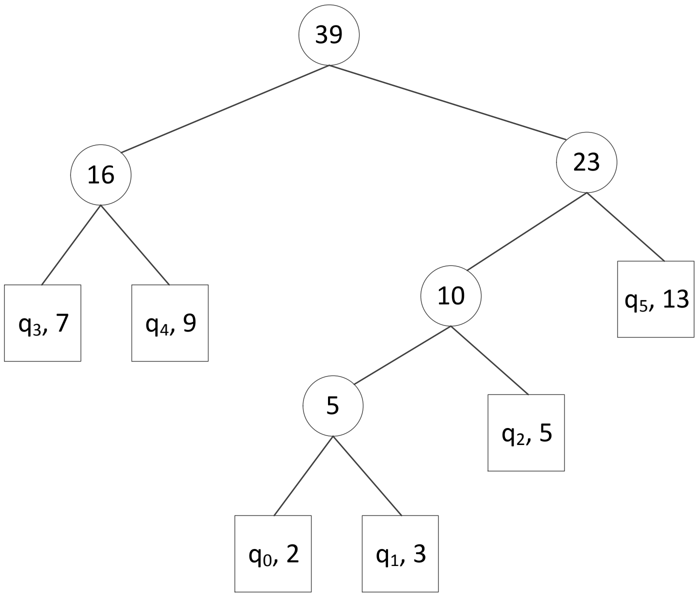
The binary trees are maintained in a (priority) list L.

```
//L is a list of trees
treeNode<symbRec>* huffmanTree(PriorityList& L)
{
   //pseudo code
   int n = L.length();
   for(int i = 1; i < n; i++)
   {
      treeNode<symbRec> *t = new treeNode<symbRec>;

      t->left = L.least(); // retrieve and remove the tree
                           // in L with smallest weight
      t->right = L.least();

      t->info.weight = t->left->info.weight +
                       t->right->info.weight;

      L.insert(t);
   }
   return L.least();
}
```

For example, there are 6 symbols with frequencies $(q_0,2), (q_1,3), (q_2,5), (q_3,7), (q_4,9), (q_5,13)$



Final Huffman Tree



Huffman codes for the 6 symbols

symbol	code
q_0	1000
q_1	1001
q_2	101
q_3	00
q_4	01
q_5	11

Priority queue

- The element to be deleted is the one with the highest priority.
- If 2 or more elements are having the same highest priority, remove 1 of the elements (not specified which one is removed).

Linear array implementation of priority queue

Case 1: Unordered list

insertion:	Insert at the rear which takes *O*(1) time.
deletion:	Search for the highest priority element and move the record at the rear to fill the hole. This takes *O*(*n*) time.

Case 2: Ordered list

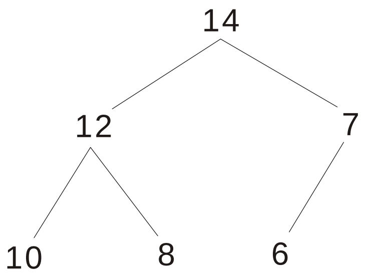
insertion:	Insert into an ordered list takes *O*(*n*) time.
deletion:	The highest priority element is at the rear. Removing the rear element takes *O*(1) time.

Implement a priority queue using a Heap

A max tree is a tree in which the key value in each node is no smaller than the key values in its children (if any).

Similarly, a min tree is a tree in which the key value in each node is no larger than the key values in its children (if any).

A max-heap (descending heap) is an almost complete binary tree that is also a max tree.



an example max-heap

A min-heap (ascending heap) is an almost complete binary tree that is also a min tree.

Function to insert an element to a heap.

To simplify the discussion, we assume the data field is an integer in the following functions.

```
struct heap
{
   int *store;  // array to store the elements
   int maxSize; // physical size of the array
   int size;    // no. of elements in the heap
}

void insert(heap& h, int x)
{
   if (h.size >= h.maxSize)  // expand the array
   {
      h.maxSize *= 2;
      int *newStore = new int[h.maxSize];

      for (int i = 0; i < h.size; i++)
         newStore[i] = h.store[i];

      delete[] h.store;
      h.store = newStore;
   }

   // i references the vacant node under consideration
   int i = h.size;
   h.size++;

   while (i > 0 && x > h.store[(i-1)/2])
   {
      // x > element in parent node

      // move element from parent node to node i
      h.store[i] = h.store[(i-1)/2];

      i = (i-1)/2; // update i to point to its parent
   }
   h.store[i] = x;
}
```

Function to delete the largest element from a heap.

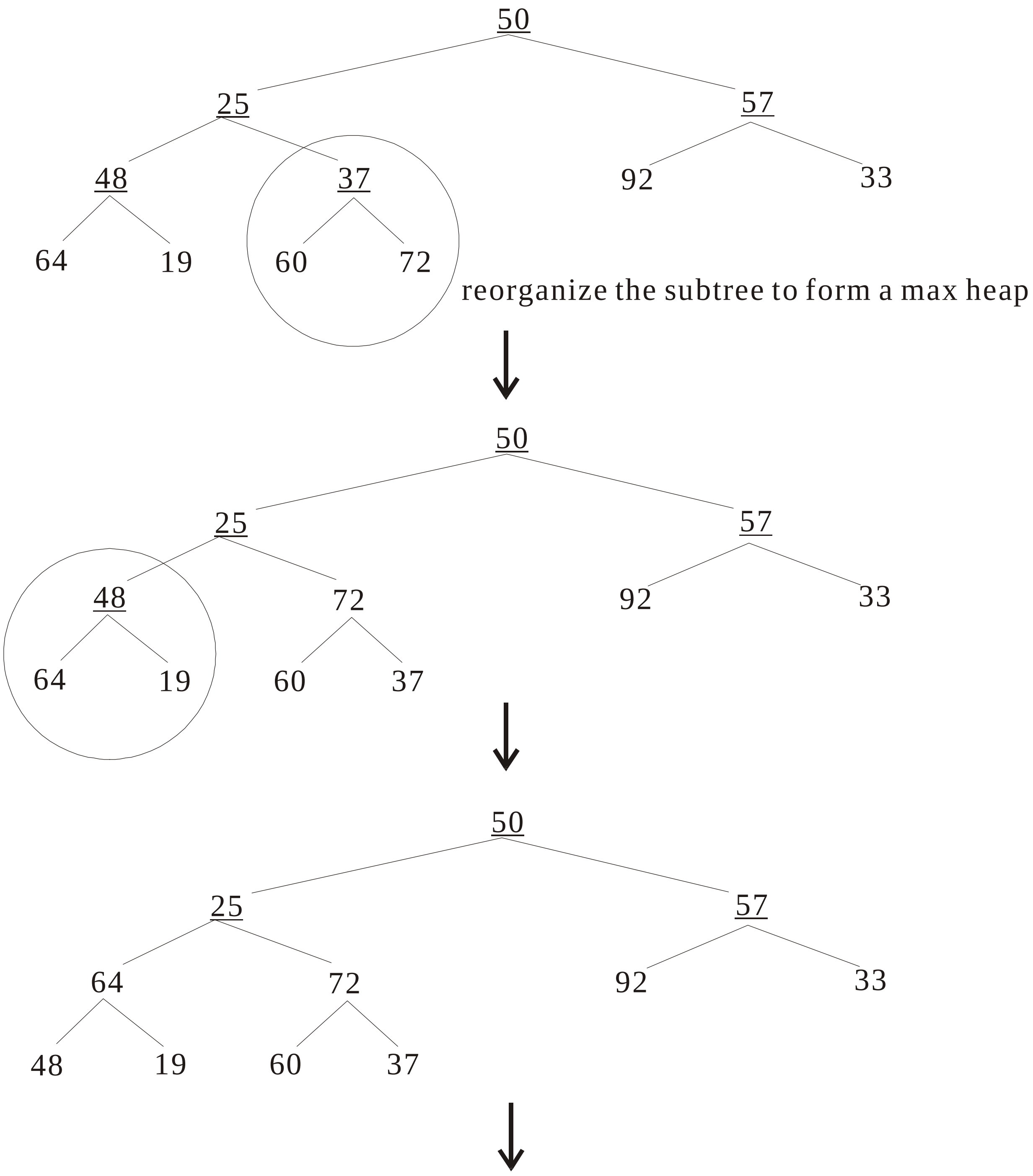
```
int delete(heap& h)
{
   // Precondition: h.size > 0
   // Remove and return the max value in the heap.

   int x = h.store[0];         //element to be removed
   int r = h.store[h.size-1]; //element to be relocated
   h.size--;

   bool done = false;
   int i = 0;  // i refers to the empty node
   int j = 1;  // node j is a child of node i

   /*  Remark:
       If h.size <= 1 after the removal of the root,
       the while-loop is not executed (since j >= h.size).
   */

   while (j < h.size && !done)
   {
      // determine the larger child of node i
      if (j < h.size - 1)
         if (h.store[j] < h.store[j+1])
            j++;

      if (r >= h.store[j])
         done = true;
      else
      {
         //move larger child to parent node
         h.store[i] = h.store[j];
         i = j;
         j = 2*i + 1;
      }
   }
   h.store[i] = r;
   return x;
}
```

Heapsort

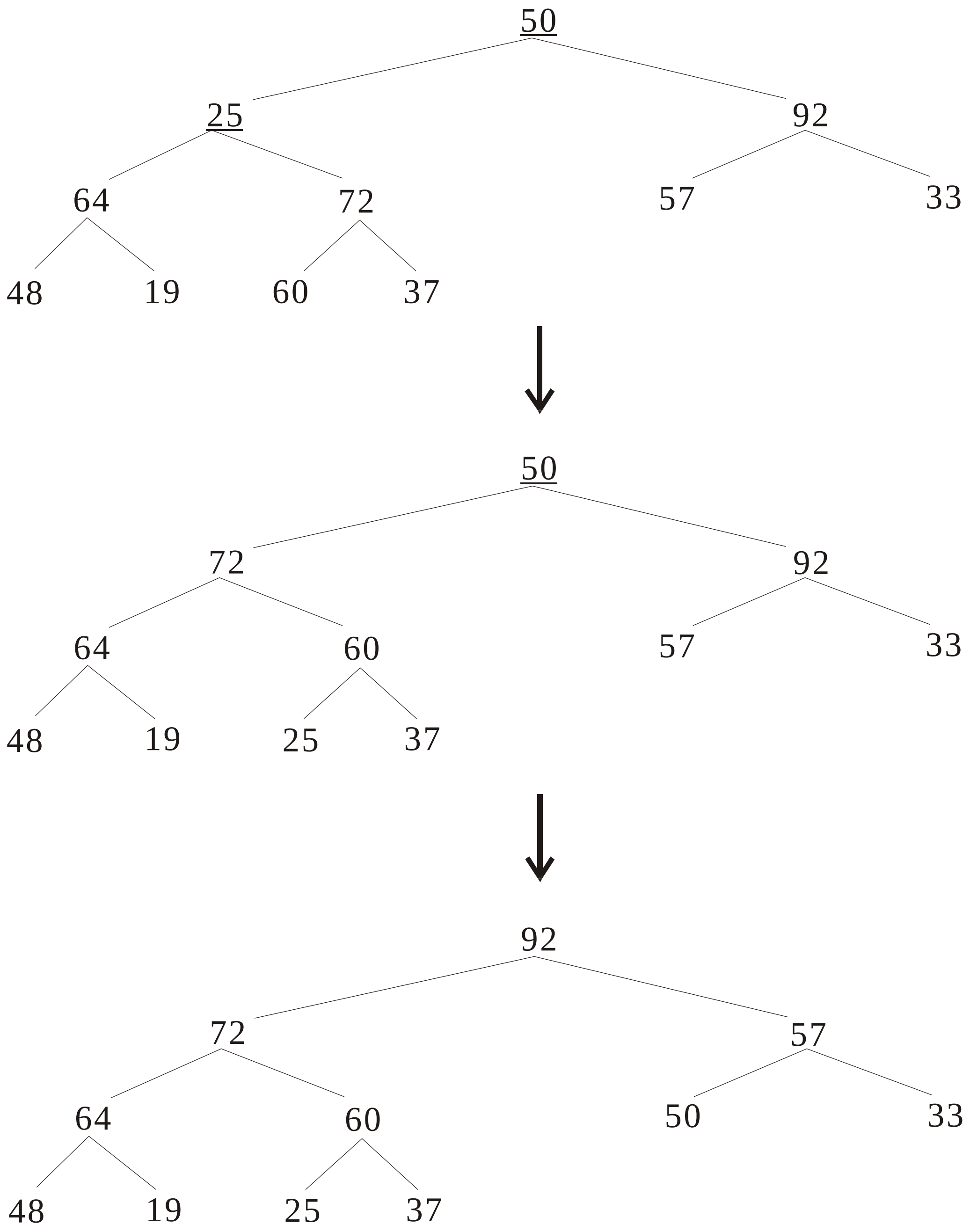
1. Organize the input array of numbers as a max heap.
2. Iteratively remove the root of the heap (largest value in the subgroup of numbers to be sorted) and re-adjust the array to form a max heap.

1. Steps to organize the input array to form a max heap

input array: [50 25 57 48 37 92 33 64 19 60 72]
processing order ← | [← leaf nodes →]

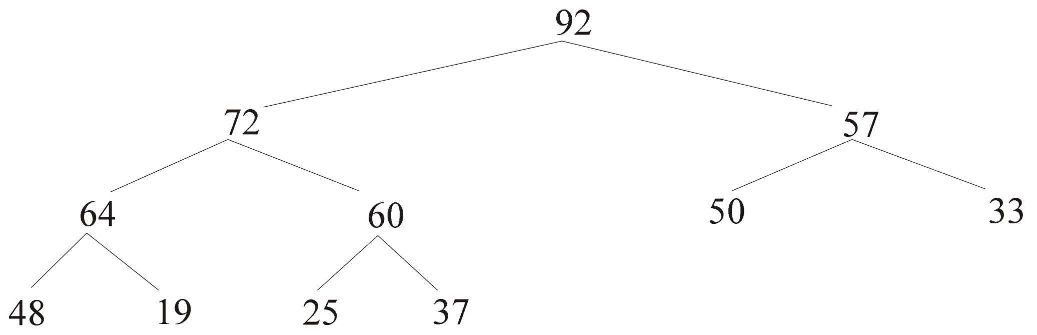


reorganize the subtree to form a max heap


50
25
92
64
72
57
33
48
19
60
37
50
72
92
64
60
57
33
48
19
25
37
92
72
57
64
60
50
33
48
19
25
37

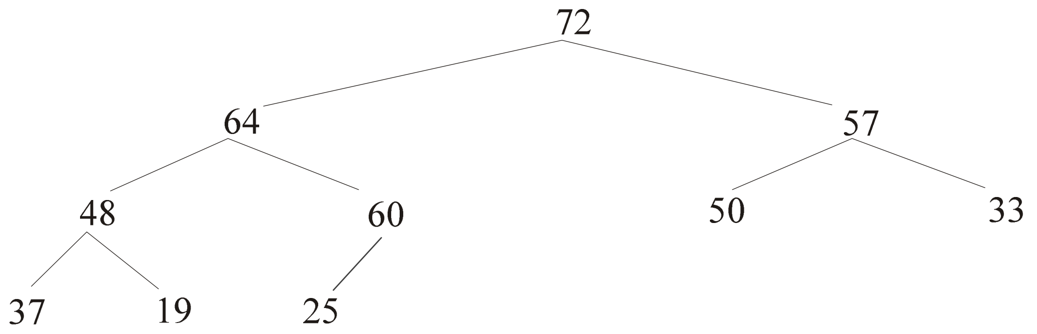
2. Sorting phase

Structure of the heap:



input array: [92 72 57 64 60 50 33 48 19 25 37]

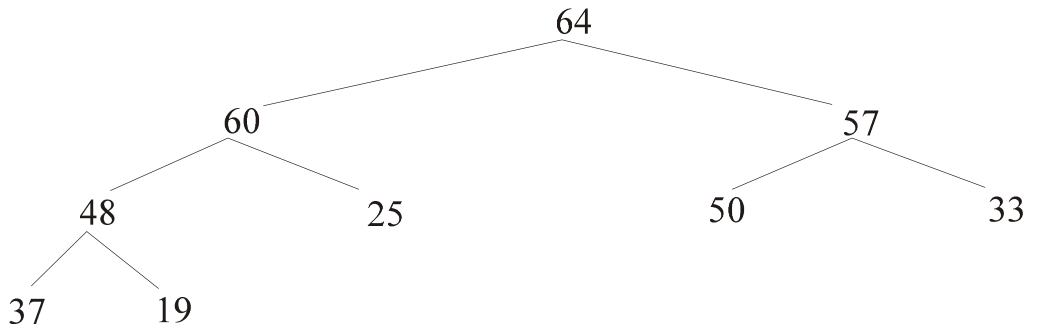
- Remove the root (swap it with the last element of the heap), and readjust the heap.



input array: [72 64 57 48 60 50 33 37 19 25 92]

|→ sorted sublist

- Repeat the process until the size of the heap is reduced to 1.



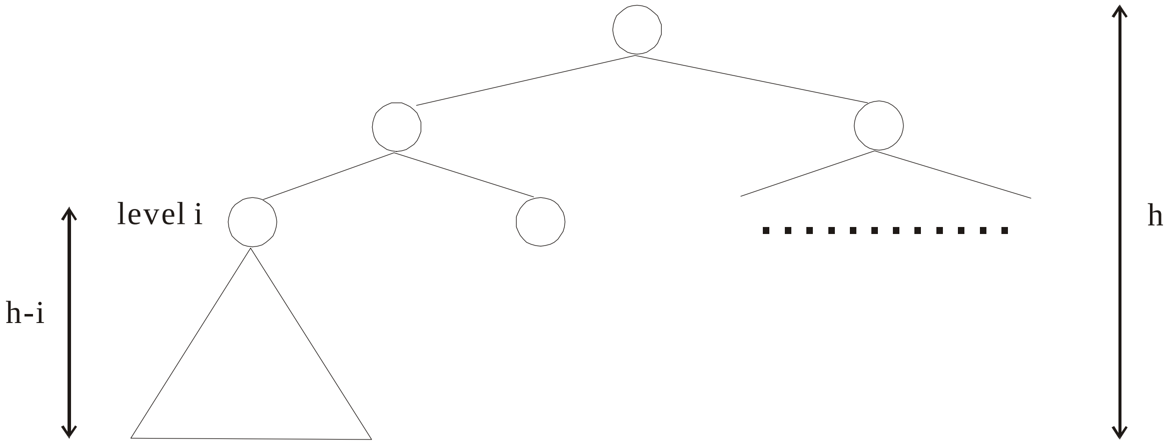
input array: [64 60 57 48 25 50 33 37 19 72 92]

|→ sorted sublist

```
void adjust(int x[], int i, int n)
// Adjust the binary tree with root i to satisfy the heap
// property. No node in the tree has index >= n.
{
   int done = 0;
   int value = x[i];
   int j = 2*i + 1;  // j is a child of i

   while (j < n && !done)
   {
      if (j < n-1)  // j has a right sibling
         if (x[j] < x[j+1])
            j = j+1;  // j is the larger child of i

      if (value >= x[j])
         done = 1;
      else  // move x[j] to its parent’s position
      {
         x[(j-1)/2] = x[j];
         j = 2*j + 1;
      }
   }
   x[(j-1)/2] = value;
}


void heapsort(int x[], int n)
{
   // Organize the array to form a max heap
   for (int i = (n-2)/2; i >= 0; i--)
      adjust(x, i, n);

   // Sort the array into ascending order
   for (int i = n-1; i > 0; i--)
   {
      swap(x, 0, i);
      adjust(x, 0, i);  // adjust the remaining tree of
                        // size i to form a max heap
   }
}
```

Complexity of heapsort

Time to organize the array to form a max heap:



root node is on level 1

h = height of the tree

n = number of nodes in the tree, $2^{h-1} \leq n \leq 2^{h}$

number of nodes on level *i* = 2^{i-1}

Total number of swaps $= \sum_{i=1}^{h}(h-i)2^{i-1}$

$$= \sum_{j=0}^{h-1} j2^{h-j-1} \quad \text{(substitute j} = h-i\text{)}$$

$$= 2^{h-1}\sum_{j=0}^{h-1}\frac{j}{2^{j}}$$

$$< n\sum_{j=0}^{h-1}\frac{j}{2^{j}}$$

$$< 2n$$

Total time of heapsort = time to organize the array to form a max heap + time to sort the array

$= O(n) + O(n \log n)$

$= O(n \log n)$

How fast can we sort?

- A set of n elements has $n!$ permutations.

- The decision tree for a sorting method based on compare-and-exchange has $n!$ leaves.

- A binary tree with height k can have at most 2^k leaves.

- Hence, the height of a binary tree with $n!$ leaves is at least
  $\log_2 (n!) = O(n \log n)$

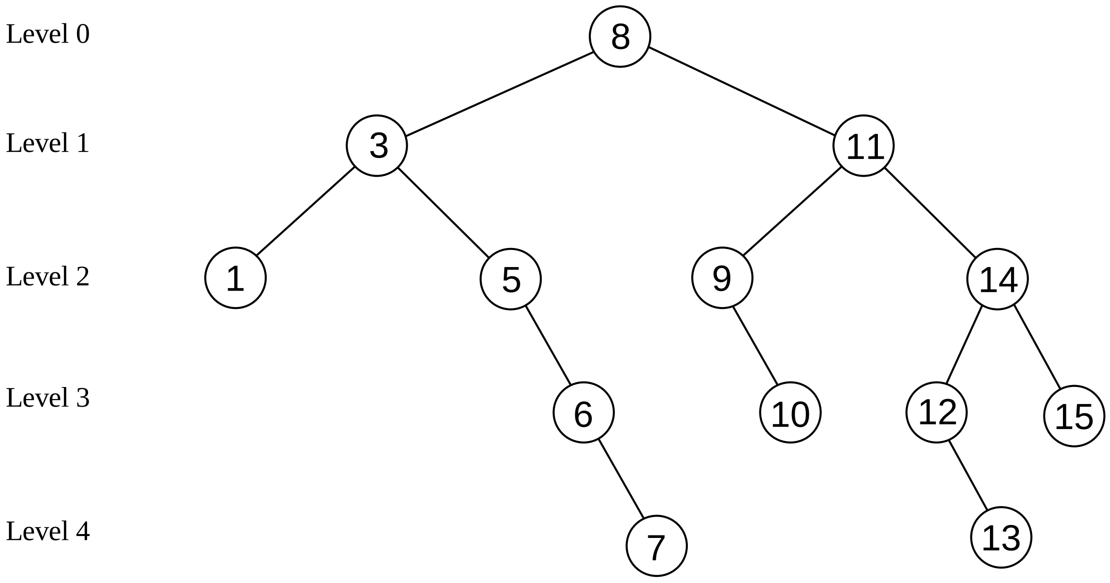
- Any sorting method which is based on compare-and-exchange will have a worst cast time complexity of $\Omega(n \log n)$.

Binary Search Tree (BST)

A binary search tree is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties:

1. Every element has a key field and no two elements in the BST have the same key, i.e. all keys are distinct. (Example, student ID is a key field in the student record.)
2. The keys (if any) in the left subtree are smaller than the key in the root.
3. The keys (if any) in the right subtree are greater than the key in the root.
4. The left and right subtrees are also BST.

Example BST (keys are integers):



Remarks:

- I shall only introduce the conceptual idea of BST without getting into the details of the implementation of the BST as a C++ class.
- In the C++ STL, the container set is implemented as BST.

Non-recursive algorithm to search a BST.

To simplify the discussion, we assume the data field in `treeNode` is an integer in the following examples.

```
treeNode<int>* search(const treeNode<int> *tree, int x)
{
  treeNode<int> *p = tree;

  while (p != nullptr && x != p->info)
  {
     if (x < p->info)
       p = p->left;
     else
       p = p->right;
  }
  return p;
}
```

Recursive algorithm to search a BST

```
treeNode<int>* search(const treeNode<int> *p, int x)
{
  if (p == nullptr)
    return nullptr;

  if (x == p->info)
    return p;

  if (x < p->info)
    return search(p->left, x);
  else
    return search(p->right, x);
}
```

Time complexity of the search operation is proportional to the height of the BST.

Height of a binary tree with *n* nodes

worst case	*n*
best case	$\lceil \log_2 (n+1) \rceil$
average case (random insertions)	$1.38 \log_2 n$

Insertion into a BST

The procedure consists of two major steps:

1. verify that the new element x is not in the BST
2. determine the point of insertion

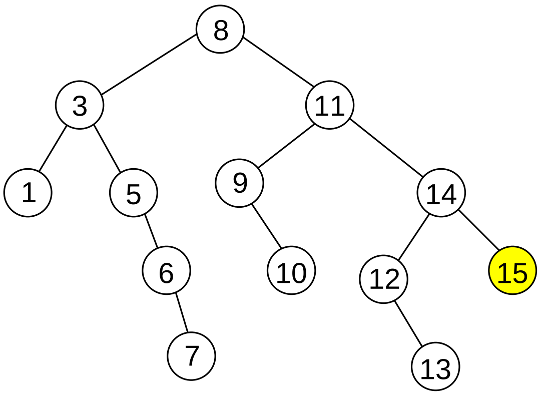
The insertion function returns the pointer to the newly inserted node or the node with the given key value.

```
treeNode<int>* insert(treeNode<int>*& root, int x)
{
  treeNode<int> *p, *q;

  q = nullptr;  // q is the parent of p
  p = root;
  while (p != nullptr)
  {
    if (x == p->info)
      return p; //element already exists

    q = p;
    if (x < p->info)
      p = p->left;
    else
      p = p->right;
  }

  treeNode<int> *v = new treeNode<int>;
  v->info = x;
  v->left = v->right = nullptr;

  if (q == nullptr)
     root = v;
  else if (x < q->info)
    q->left = v;
  else
    q->right = v;

  return v;
}
```

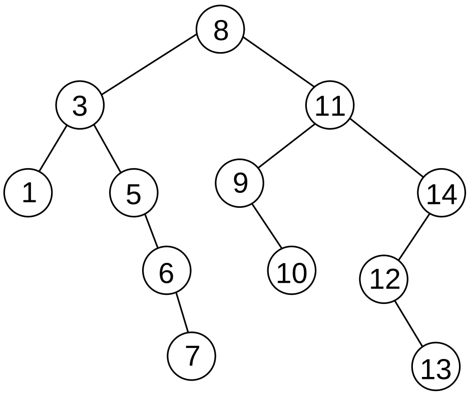
Delete an element x from a BST: there are 3 different cases.

1. x is a leaf
   a) simply remove x

   Example: delete 15

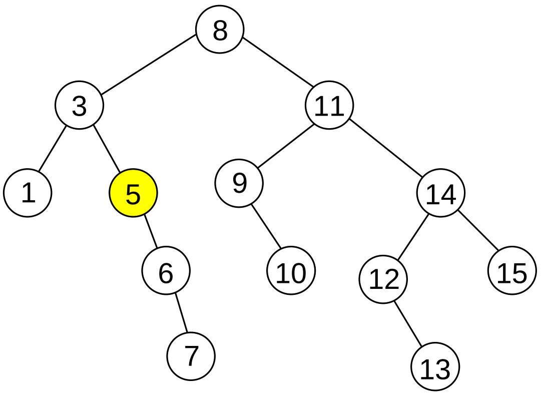


BST after deleting 15

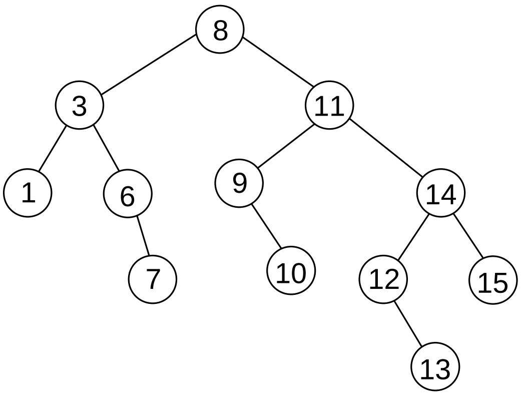


2. x has one non-empty subtree whose root is y
   a) if x is the leftchild (rightchild) of q, make y to become the leftchild (rightchild) of q
   b) remove x

   Example: delete 5

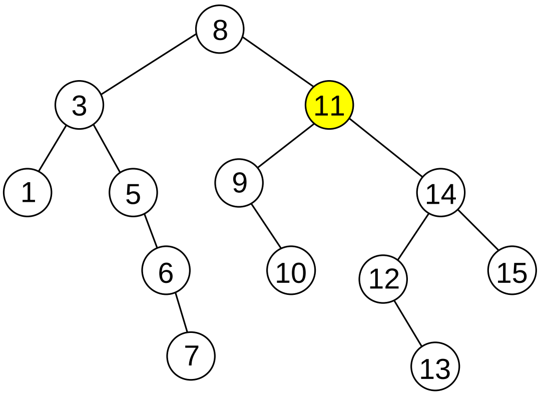


BST after deleting 5

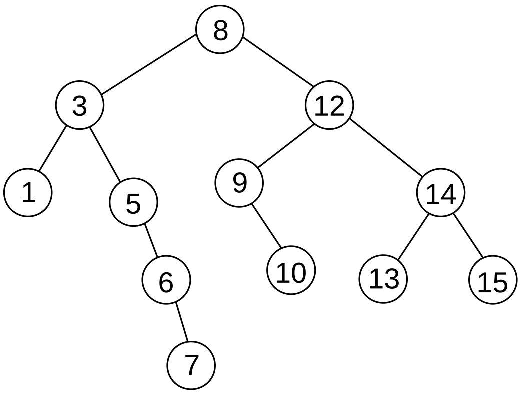


3. x has two nonempty subtrees
   a) replace x by z, where z is the inorder successor (or predecessor) of x
   b) remove z in turn (it is guaranteed that z has at least one empty subtree)

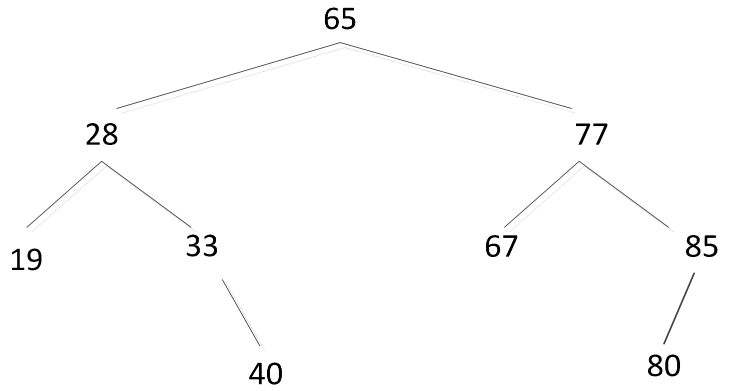
   Example: delete 11



BST after deleting 11

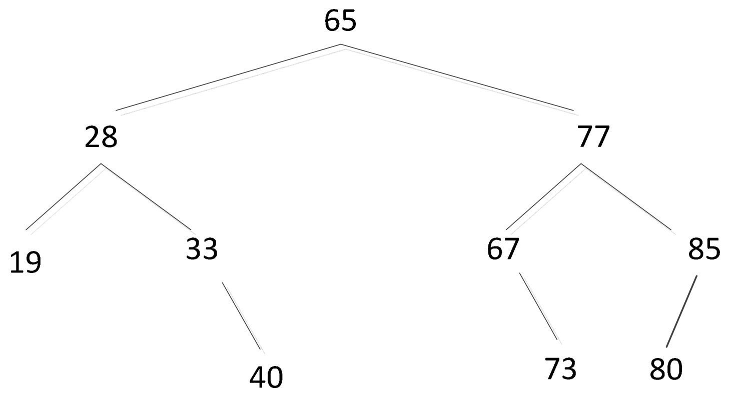


Example: Initial structure of a binary search tree

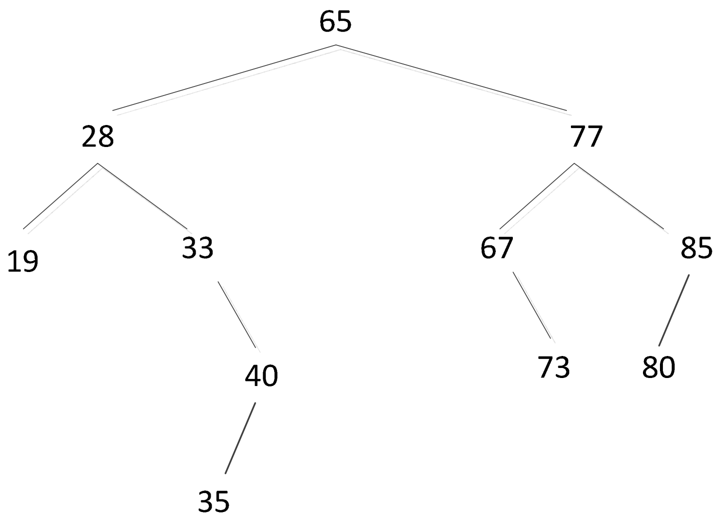


Structure of the binary search tree after performing a sequence of insert and delete operations: insert(73), insert(35), delete(33), delete(65)

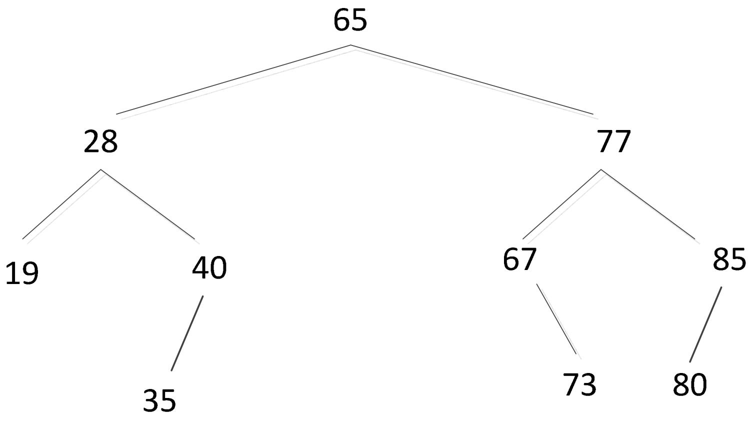
Insert 73



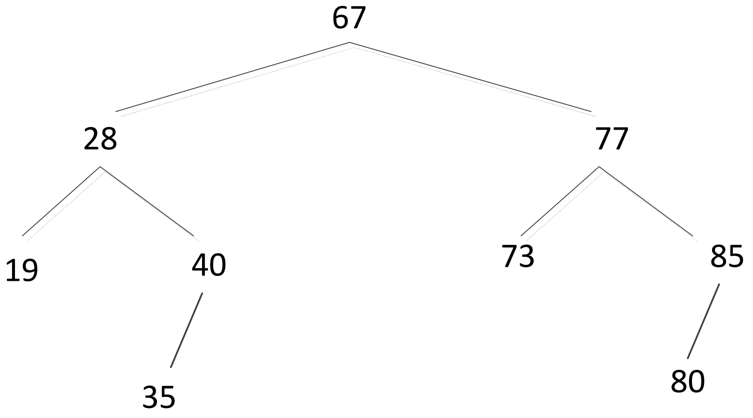
Insert 35



Delete 33



Delete 65

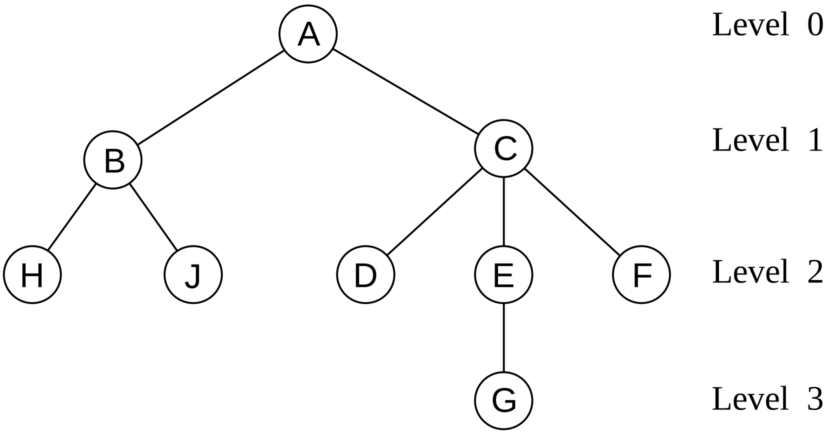


General tree

A tree is defined as a finite set T of one or more nodes such that

a) there is one specially designated node called the root of the tree, and

b) the remaining nodes (excluding the root) are partitioned into $m \geq 0$ disjoint sets $T_1, T_2, ..., T_m$ and each of these sets in turn is a tree. The trees $T_1, T_2, ..., T_m$ are called the subtrees of the root.



A sample tree

The definitions of parent-child relation, ancestor-descendant relation, leaf/internal nodes, level of nodes, depth, etc. are the same as in binary tree.

The major difference is that there is no limit on the degree of a node in a general tree.

Representation methods:

Nested parentheses representation :

(A (B (H) (J)) (C (D) (E (G)) (F)))

This representation is suitable for specifying a general tree in text format (e.g. in a data file).

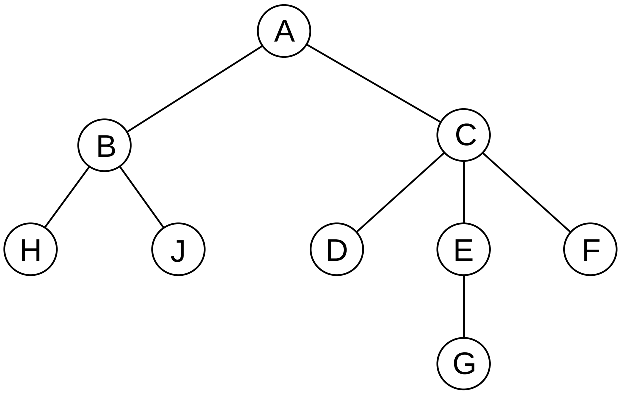
Linked representation using k-ary tree: A node has k pointers.

Data	link 1	link 2	link 3	. . . . .	link k

Disadvantages of this representation:

- The maximum degree of the tree is assumed, i.e. k-ary tree.
- If the actual degree of the tree exceeds the assumed value, the program fails.

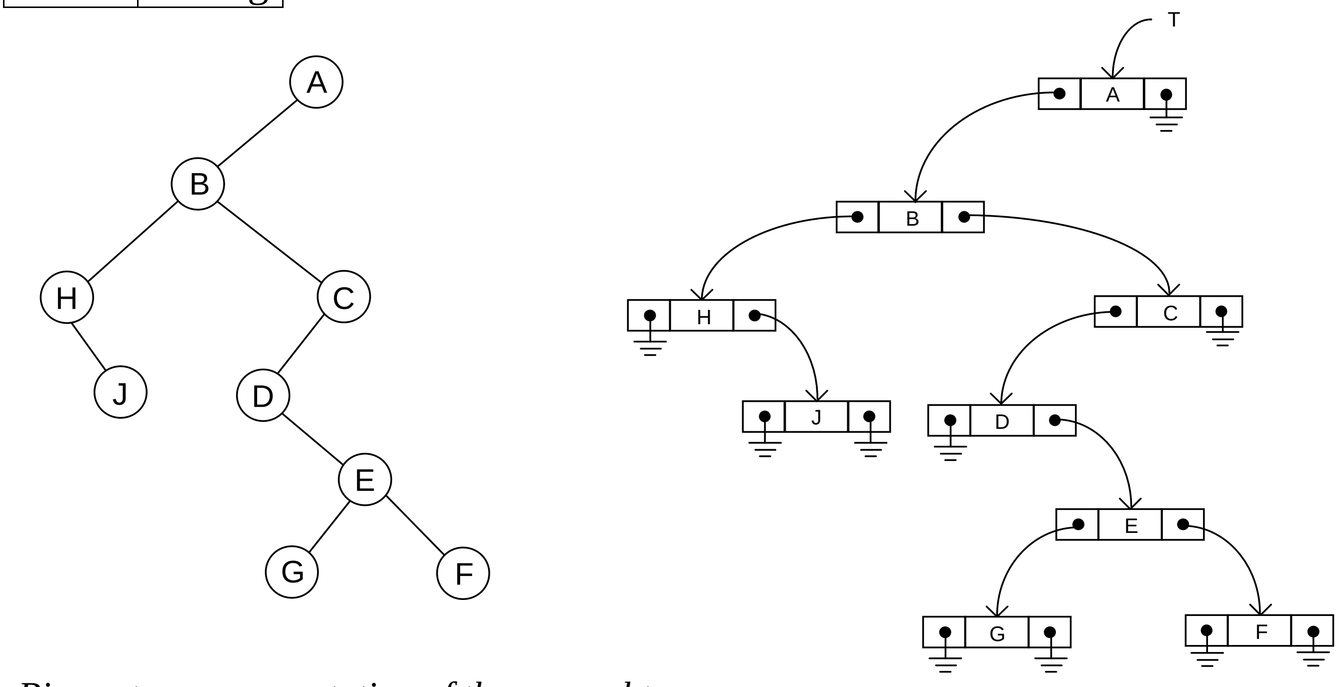
Representing a general tree using binary tree:



General tree

Node structure

Data	
child	sibling



Binary tree representation of the general tree

Algorithm to count the number of leaf nodes in a general tree represented as a binary tree

```
template<class Type>
int countLeaf(treeNode<Type> *p)
{
   if (p == nullptr)  // tree is empty
      return 0;

   if (p->left == nullptr) // root has no subtree
      return 1;

   /* root has 1 or more subtree.
      number of leaf nodes = sum of leaf nodes in the
                             subtrees of the root
   */

   int count = 0;
   p = p->left;
   while (p != nullptr)  // for each subtree
   {
      count += countLeaf(p);

      p = p->right;      // move on to the next subtree
   }

   return count;
}
```

Algorithm to determine the height of a tree represented as a binary tree

```
template<class Type>
int height(treeNode<Type> *p)
{
   // Definition used in this example:
   // height of a tree with only the root is equal to 1

   int h, t;

   if (p == nullptr)
      return 0;

   h = 0;
   p = p->left;
   while (p != nullptr)
   {
      t = height(p);
      if (t > h)
         h = t;
      p = p->right;
   }

   // h = max height of all subtrees
   return h+1;
}
```

Typical structure of the recursive algorithm that operate on a general tree represented as binary tree

```
returnType functionName(treeNode<Type> *tree)
{
   if (tree == nullptr)
      // base case

   process the root;

   treeNode<Type> *p = tree->left;

   while (p != nullptr) // loop to process children of root
   {
      // process subtree whose root is p using recursion

      p = p->right;  // move on to the next subtree
   }

   return final_result;
}
```

Function to construct a general tree from the nested parenthesis representation.
A node label is represented by a single letter.
The general tree is represented as a binary tree.

```
treeNode<char>* buildGT(const char *t, int& i)
{
   // Input data format:
   //    each node label is a single letter
   //    no space char in the array t[]

   // Precondition: the tree is not empty
   //               t[i] is the ‘(’, and
   //               t[i+1] is the root label

   treeNode<char> *root = new treeNode<char>;
   root->left = root->right = nullptr;
   root->info = t[i+1];
   i += 2;

   if (t[i] != ‘)’)  // root node has 1 or more child
   {
      root->left = buildGT(t, i);
      treeNode<char> *p = root->left;

      while (t[i+1] != ‘)’)  // child node has sibling
      {
         i += 1;
         p->right = buildGT(t, i);
         p = p->right;
      }
      i += 1;
   }
   // index i is advanced to point to the ‘)’ of the
   // subtree in the input array

   return root;
}
```