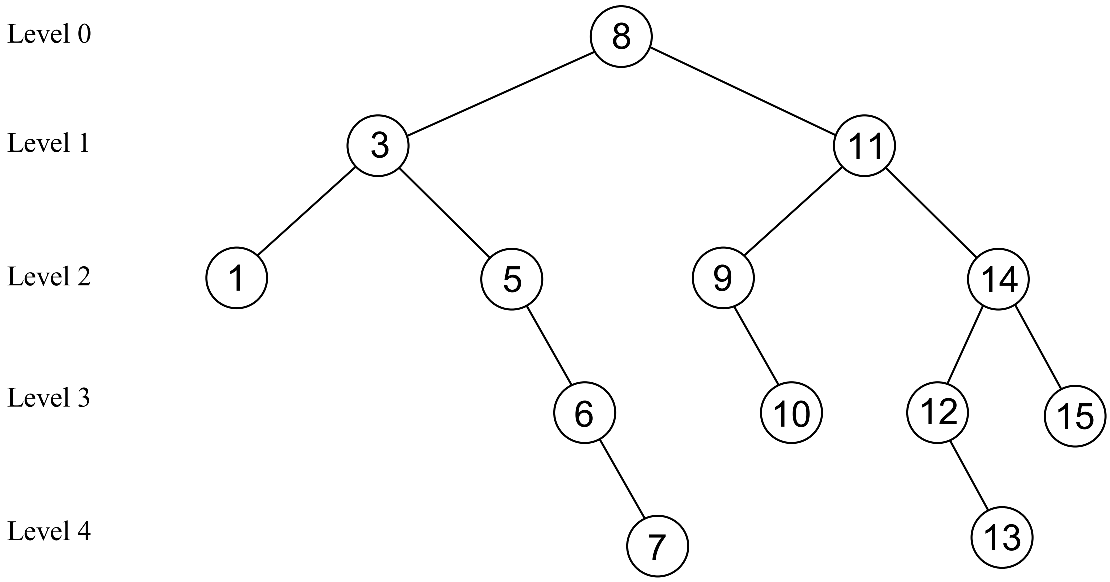
Binary Search Tree (BST)

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 larger 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

```
template<class Type>
treeNode<Type>* search(const Type& x)
{
  treeNode<Type> *p = root;

  //in general, comparison is based on the key field
  while (p != NULL && x != p->info)
  {
     if (x < p->info)
       p = p->left;
     else
       p = p->right;
  }
  return p;
}

/* Remark:
   If search() is a public member function of class BST,
   then the return value should not be treeNode<Type>*
   (to prevent exposing internal structure).

   Instead, the public search() function should return an
   iterator that refers to the node containing x.
*/
```

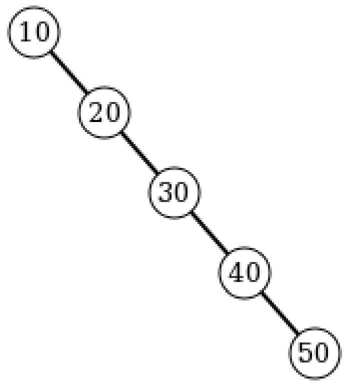
Recursive algorithm to search a BST

```
template<class Type>
treeNode<Type>* search(treeNode<Type> *p, const Type& x)
{
  if (p == NULL)
    return NULL;

  if (x == p->info)  //comparison is based on the key field
    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 (skewed tree)	n
best case (complete/almost complete tree)	$\lceil \log_2 (n+1) \rceil$
average case (random insertions)	$1.38 \log_2 n$



Skewed tree

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

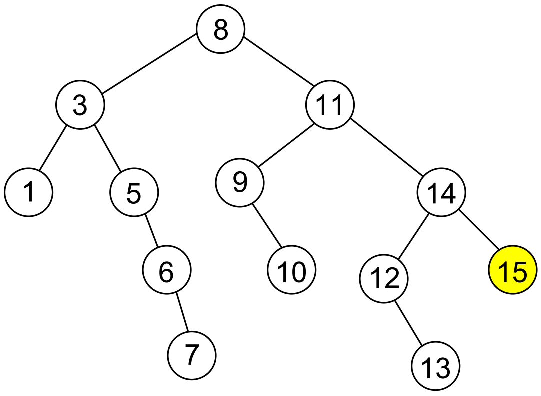
The insertion function returns the pointer to the newly inserted node or the node with the given key value.

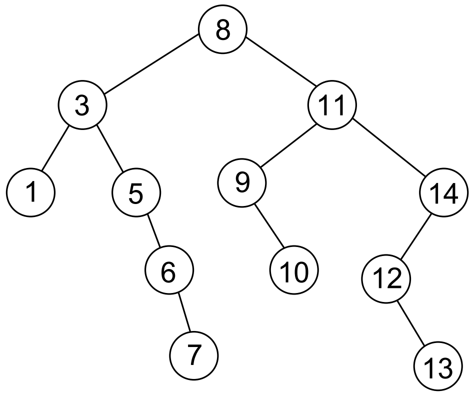
```
template<class Type>
treeNode<Type>* insert(const Type& x)
{
  treeNode<Type> *p, *q;

  q = NULL;  // q is the parent of p
  p = root;
  while (p != NULL)
  {
    //in general, comparison is based on the key field
    if (x == p->info)
       return p; //element already exists

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

  treeNode<Type> *v = new treeNode<Type>;
  v->info = x;
  v->left = v->right = NULL;

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

  return v;
}
```

Delete an element x from a BST

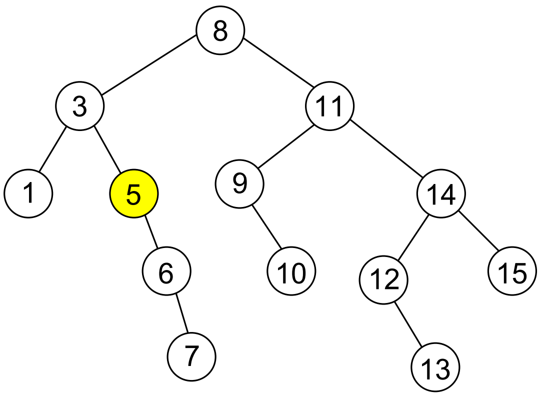
There are three different cases:

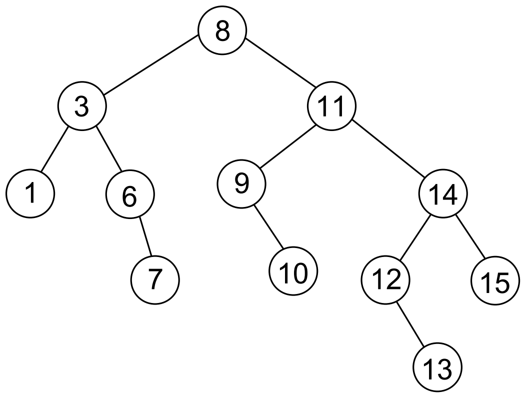
1. x is a leaf
   a) simply remove x



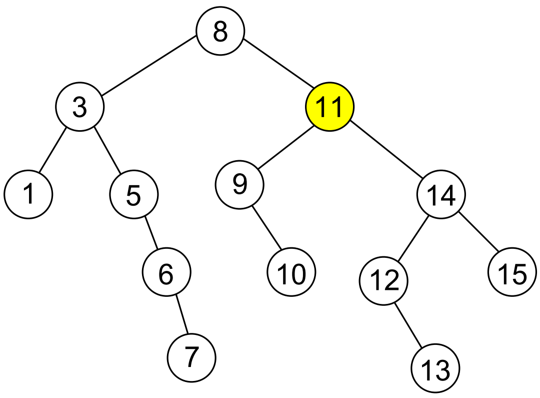


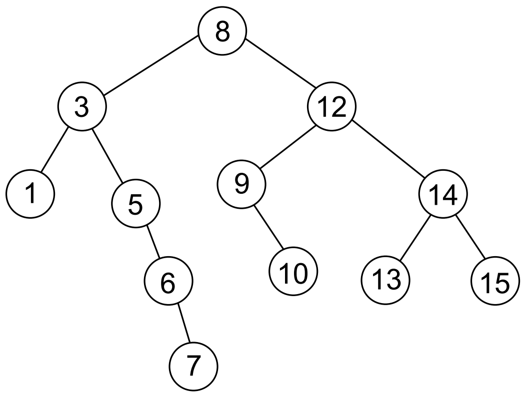
2. x has one non-empty subtree whose root is y
   a) if x is the leftchild (rightchild) of q, make y to become the leftchild (rightchild) of q
   b) remove x





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)



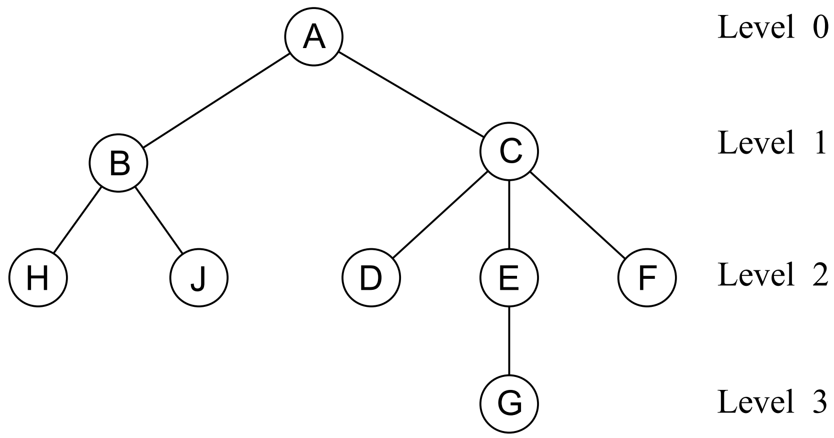


General tree

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.

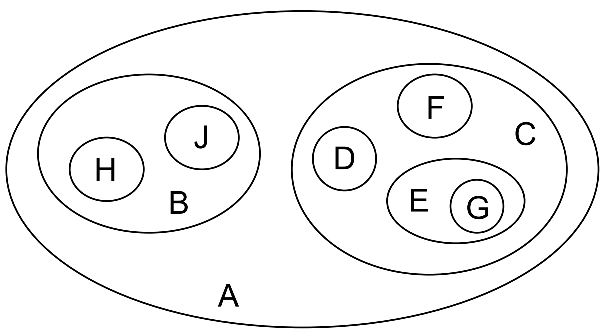
The major different is that there is no limit on the degree of a node in a general tree.

Representation methods:

Nested parentheses representation

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

Nested sets



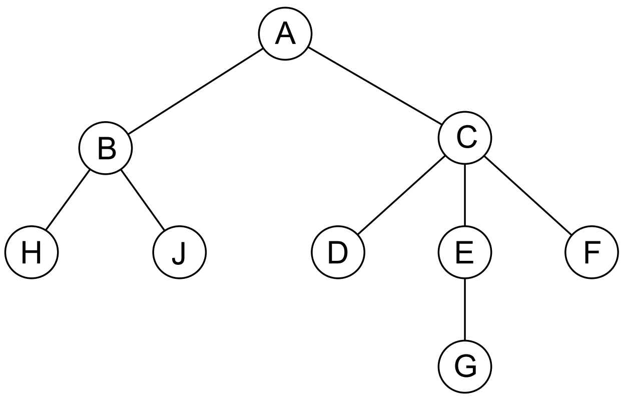
Linked representation using k-ary tree

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.

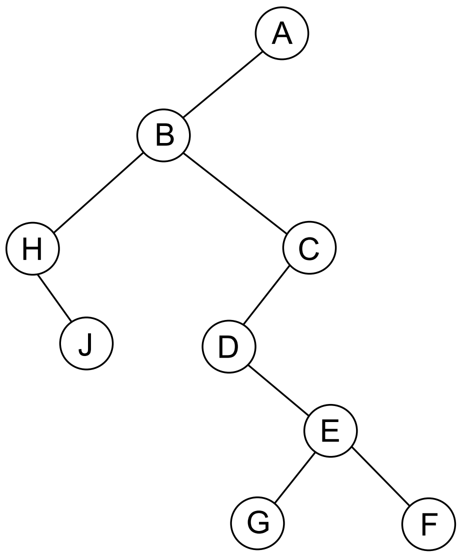
Representing a general tree using binary tree:

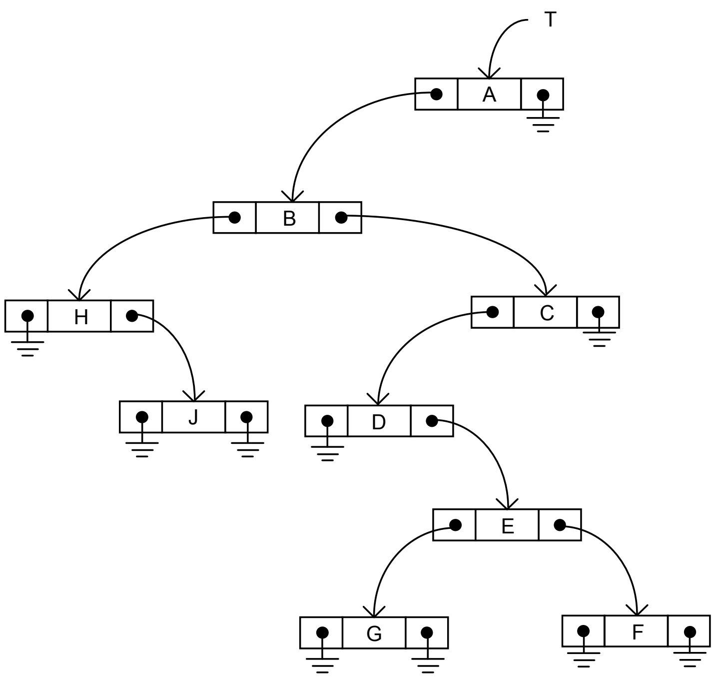


General tree

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)
{
   int count;

   if (p == NULL)  // tree is empty
      return 0;

   if (p->left == NULL) // 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  */

   count = 0;
   p = p->left;
   while (p != NULL)  //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 == NULL)
      return 0;

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

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