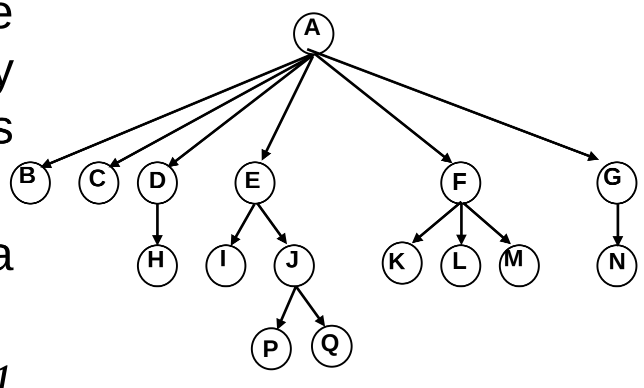
Chap. 4 Trees

4.1 Preliminaries

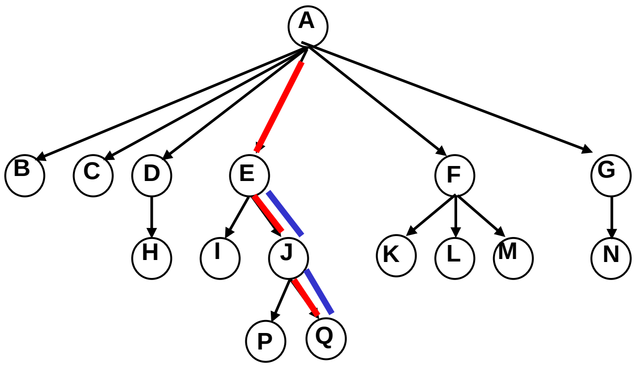
– Definitions

- ***Tree*** (recursive definition): A tree is a collection of nodes. If the collection is not empty, it must consist of a unique ***root*** node, r, and zero or more nonempty subtrees T_1, $T_2,\ldots$, T_k, with roots connected by a direct ***edge*** from r
- The root r_i of each subtree is called a ***child*** of r, and r is called the ***parent*** of r_i
- For a tree of N nodes, there must be N-1 edges.



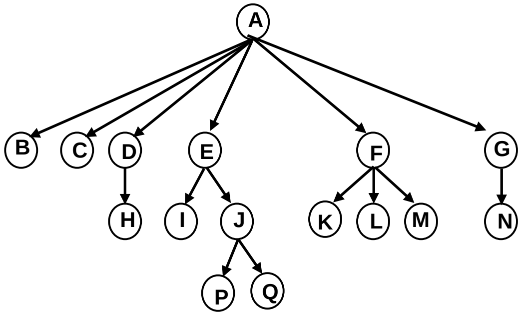
- A node with no child is called a ***leaf*** node; nodes with the same parents are ***siblings.***

- *<u>Path</u>*: a path from node n_1 to n_k is a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} $(1 \leq i < k)$; The no. of edges in the path is call the *<u>length</u>* of the path; A length 0 path is a path from a node to itself.
- There is a unique path from root to any node n_i; The length of this path is called the *<u>depth</u>* of n_i; thus, the root is a depth 0 node.
- The *<u>height</u>* of a node n_i is the length of the longest path from n_i to a leaf node, thus, the height of a leaf node is 0.
- The <u>height of a tree</u> is the height of its root node. The depth of a tree is the depth of the deepest leaf node, which is the same as the height of the tree.
- If there is a path from n_1 to n_2, then n_1 is an <u>ancestor</u> of n_2, and n_2 is a <u>descendant</u> of n_1. If $n_1 \neq n_2$, n_1 is a proper ancestor of n_2, n_2 is a proper descendant of n_1.



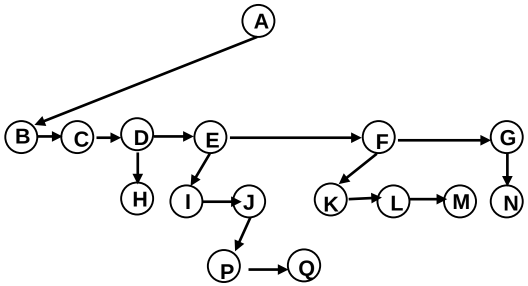
– The implementation

- Direct links to all children: only good if the number of children are relatively uniform and known in advance.



- One direct link to the first child node, with a sibling linked list

```
Structure TreeNode  {
    Object element;
    TreeNode  *firstchild;
    TreeNode  *nextsibling;
 };
```



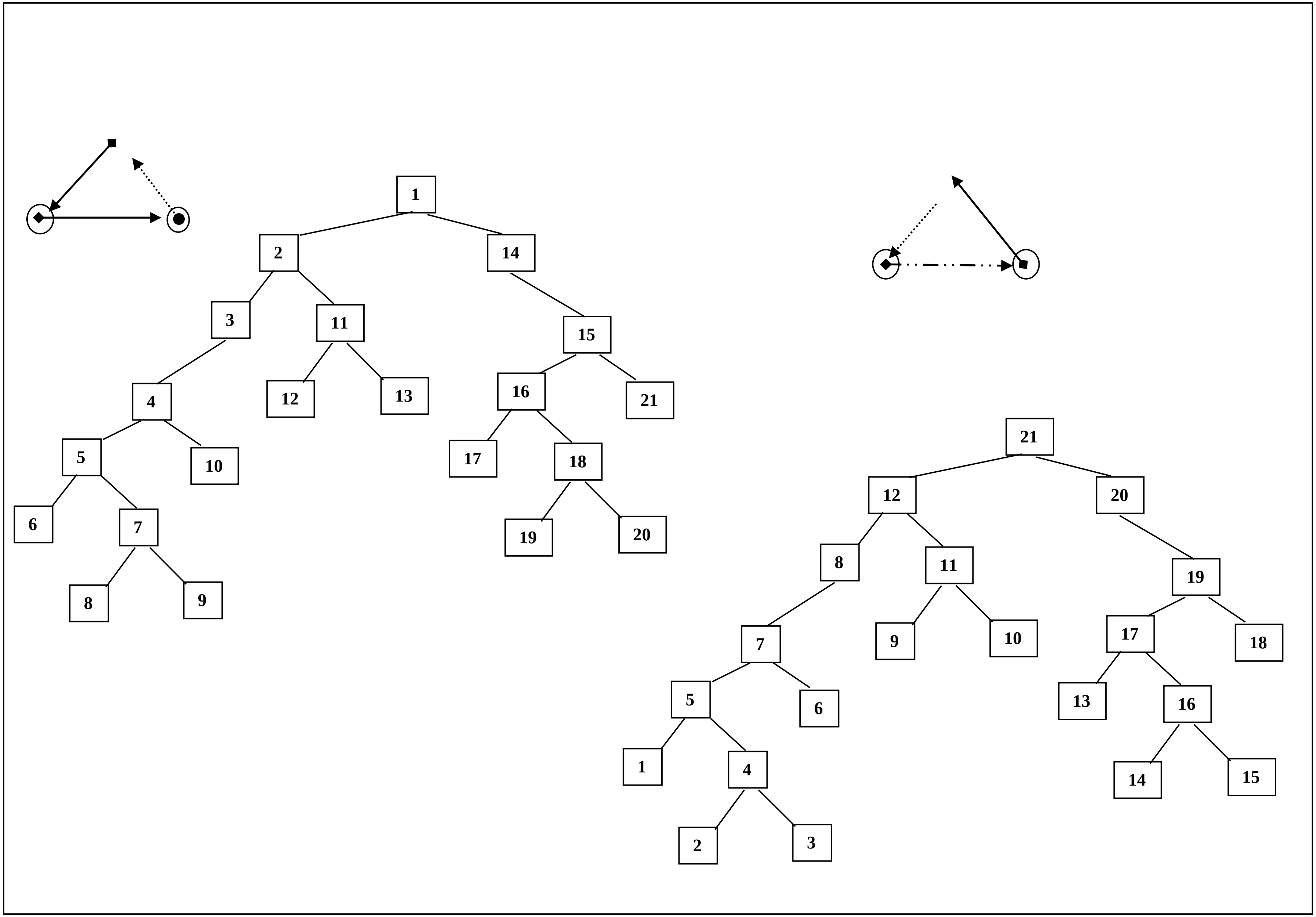
– Tree traversal

- Unix directory
- Preorder traversal: a node is processed before its children (subtree) are processed.

```
Void FileSystem::listAll(int depth = 0) const  {
    printName(depth);
    if (isDirectory( ))
      for (each file c is this directory)
           c.listAll (depth + 1);
}       //   The result is in Fig. 4.7
```

- Postorder traversal: a node is processed after its children are processed.

```
int FileSystem:: size ( ) const  {
    int totalSize = SizeOfThisFile ( );
    if (isDirectory ( ))
      for (each file c in this directory)
           total size += c. size ( );
    return (totalSize);
}     //  The result is in Fig. 4.10
```


1
2
14
3
11
15
4
12
13
16
21
5
10
17
18
6
7
19
20
8
9
21
12
20
8
11
19
7
9
10
17
18
5
6
13
16
1
4
14
15
2
3

4.2 Binary trees

- Definition: A binary tree is a tree in which no nodes can have more than two children.
- Average depth of a binary tree is $O(\sqrt{n})$, but the worst case is $O(N)$; and a well "balanced" binary tree has depth $O(logN)$.
- Implementation

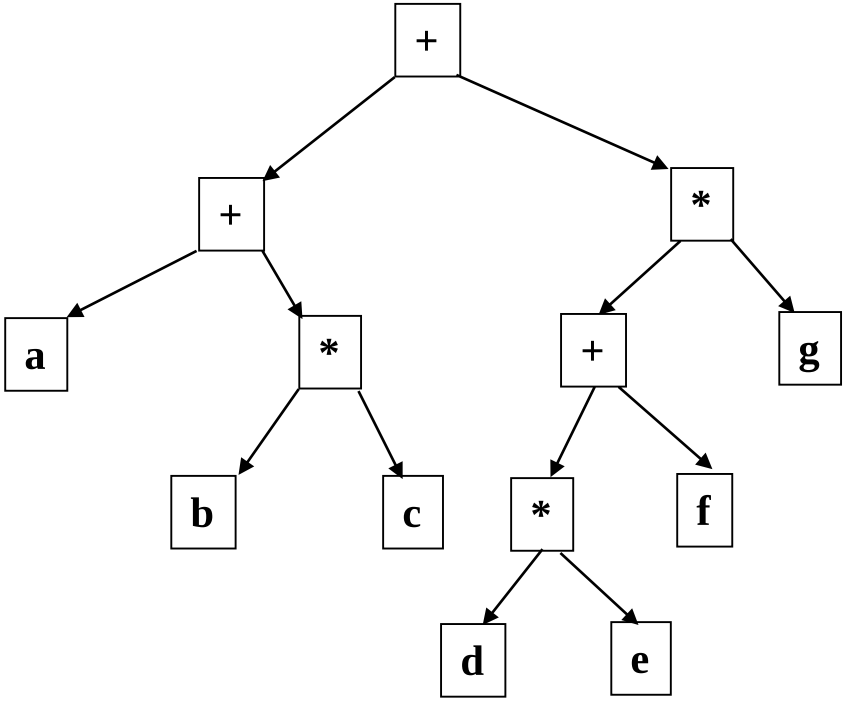
```
Struct BinaryNode {
    object element
    BinaryNode *left;
    BinaryNode *right;
}
```

- Inorder traversal: the left child is processed first, followed by the node, and then the right child.

– Expression tree

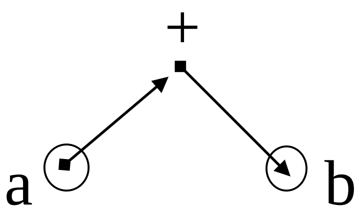
- Example of expression tree

(a+(b*c))+(((d*e)+f)*g)



abc*+de*f+g*+

- Inorder traversal: infix expression (binary tree only)
- Postorder: postfix expression
- Preorder: prefix expression



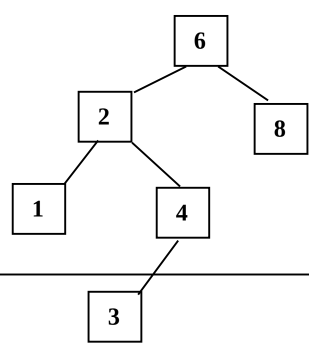
- (`Fig. 4.14 postorder expression to inorder expression`)

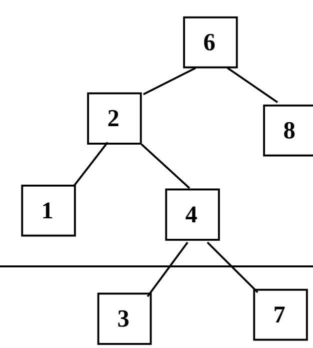
```
If symbol==operand then {
   create one-node tree;
   push a pointer into Stack
} else
If symbol==operator then {
   pop two operands from stack
   create a tree whose root is the operator and whose
  children point to the operands
   a pointer to the tree is pushed to stack
}
```

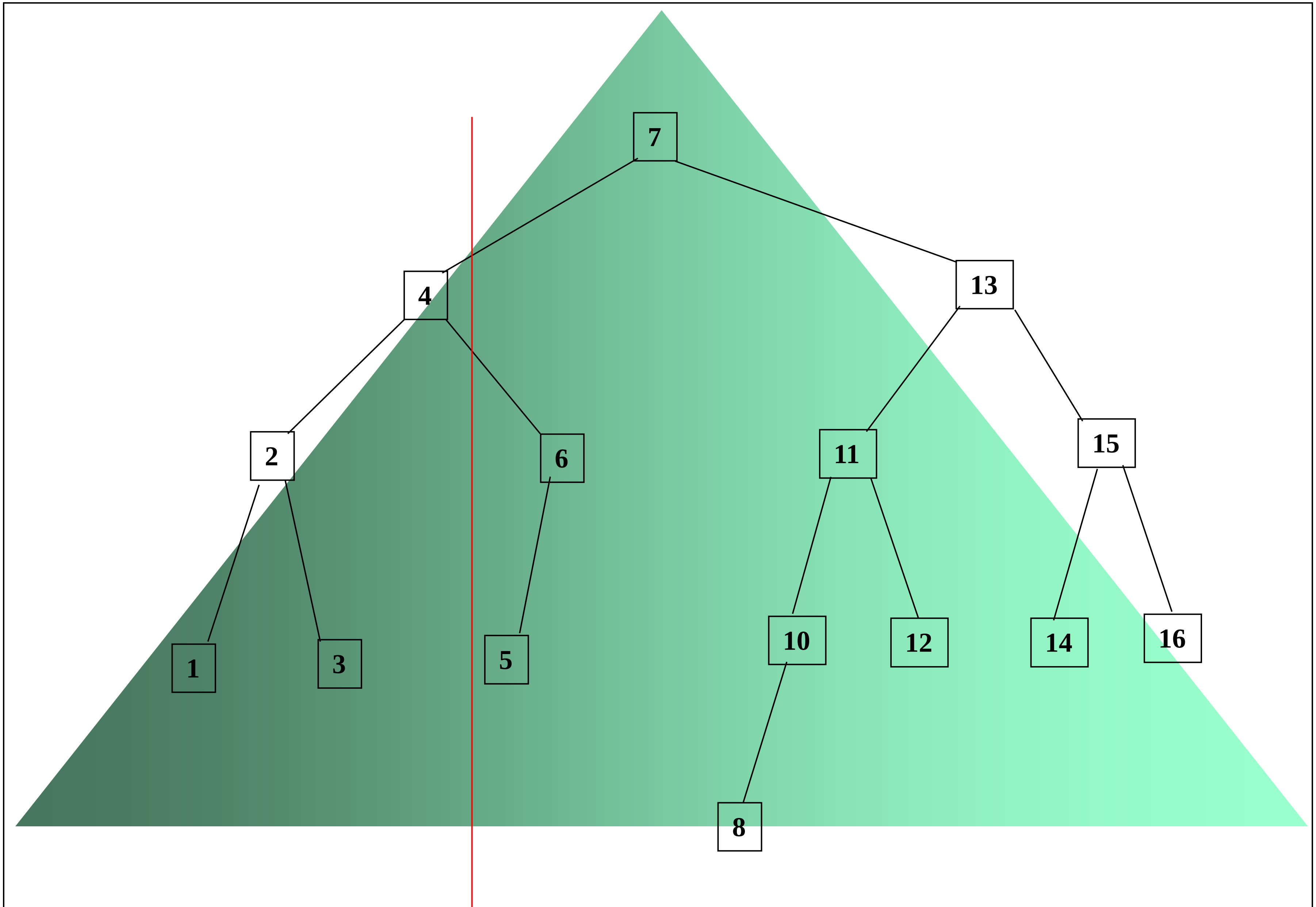
ab+cde+**

4.3 Binary Search tree

– Definitions
  - A binary search tree is a binary tree with the property that for every node, X, in the tree, the values of all the items in its left subtree are smaller (according to some pre-defined order) than the item in X, and the values of all the items in its right subtree are larger than the item in X.
  - Average depth of binary search tree is $O(logN)$ (to be proved later)
  - Operations in binary search tree can be implemented by recursive (stack depth is on average $O(logN)$)
  - C++ implementation (`Fig. 4.16, Fig. 4.17, Fig. 4.18`)

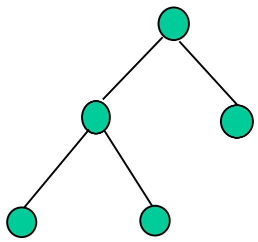






7
4
13
2
6
11
15
1
3
5
10
12
14
16
8

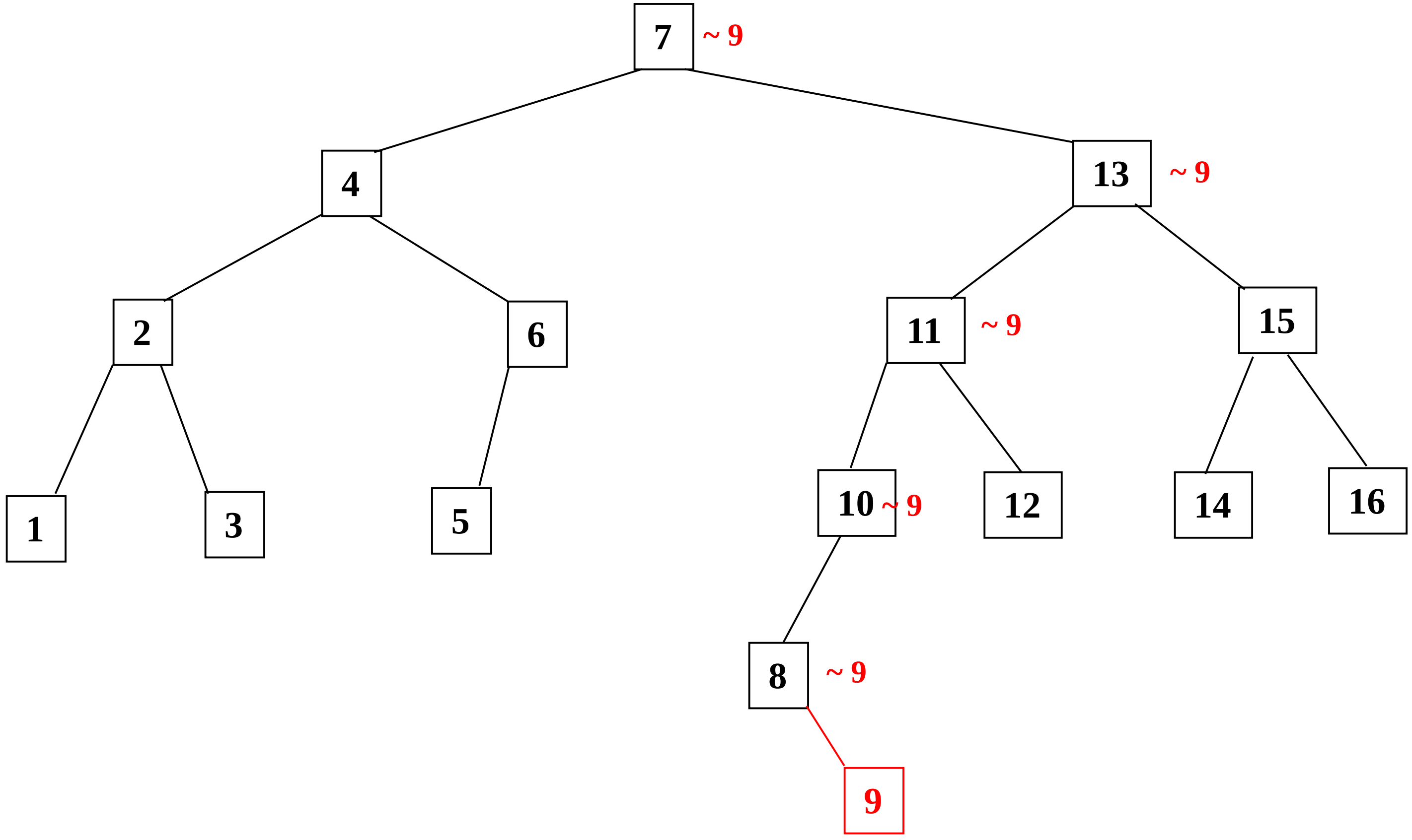
```
template <class Comparable>
class BinarySearchTree;


template <class Comparable>
class BinaryNode
{
        Comparable element;
        BinaryNode *left;
        BinaryNode *right;
BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt ): element( theElement ),
left( lt ), right( rt ) { }
friend class BinarySearchTree<Comparable>;
};


 // BinarySearchTree class  // CONSTRUCTION:  ITEM_NOT_FOUND object used to signal failed finds
       // void insert( x )       --> Insert x
      // void remove( x )       --> Remove x
      // Comparable find( x )   --> Return item that matches x
      // Comparable findMin( )  --> Return smallest item
      // Comparable findMax( )  --> Return largest item
      // boolean isEmpty( )     --> Return true if empty; else false
      // void makeEmpty( )      --> Remove all items
      // void printTree( )      --> Print tree in sorted order
```



```
template <class Comparable>
    class BinarySearchTree
    {
     public:
      explicit BinarySearchTree( const Comparable & notFound );
      BinarySearchTree( const BinarySearchTree & rhs );
      ~BinarySearchTree( );
      const Comparable & findMin( ) const;
      const Comparable & findMax( ) const;
      const Comparable & find( const Comparable & x ) const;
      bool isEmpty( ) const;
      void printTree( ) const;
      void makeEmpty( );
      void insert( const Comparable & x );
      void remove( const Comparable & x );
      const BinarySearchTree & operator=( const BinarySearchTree & rhs );
     private:
     BinaryNode<Comparable> *root;
     const Comparable ITEM_NOT_FOUND;
     const Comparable & elementAt( BinaryNode<Comparable> *t ) const;
     void insert( const Comparable & x, BinaryNode<Comparable> * & t ) const;
     void remove( const Comparable & x, BinaryNode<Comparable> * & t ) const;
     BinaryNode<Comparable> * findMin( BinaryNode<Comparable> *t ) const;
     BinaryNode<Comparable> * findMax( BinaryNode<Comparable> *t ) const;
     BinaryNode<Comparable> * find( const Comparable & x,
     BinaryNode<Comparable> *t ) const;
     void makeEmpty( BinaryNode<Comparable> * & t ) const;
     void printTree( BinaryNode<Comparable> *t ) const;
     BinaryNode<Comparable> * clone( BinaryNode<Comparable> *t ) const;
    };
    #include "BinarySearchTree.cpp"
    #endif
```

– *find ( )*

- recursive implementation: `(Fig. 4.19)`
- stack depth: $O(logN)$



```
/****** RECURSIVE VERSION*******/
template <class Comparable>
    BinaryNode<Comparable> *
    BinarySearchTree<Comparable>::
    find( const Comparable & x, BinaryNode<Comparable> *t ) const
    {
      if( t == NULL )
        return NULL;
      else if( x < t->element )
        return find( x, t->left );
      else if( t->element < x )
        return find( x, t->right );
      else
        return t;    // Match
    }
```

```
/****** NONRECURSIVE VERSION*********/
    template <class Comparable>
    BinaryNode<Comparable> *
    BinarySearchTree<Comparable>::
    find( const Comparable & x, BinaryNode<Comparable> *t ) const
    {
      while( t != NULL )
        if( x < t->element ) t = t->left;
        else if( t->element < x ) t = t->right;
        else  return t;    // Match
      return NULL;   // No match
    }
```

–*findMax ( ) & findMin ( )*

- recursive implementation: (`Fig. 4.20`)
- non-recursive implementation: (`Fig. 4.21`)

```
/** Internal method to find the smallest item in a
subtree t. Return node containing the smallest item.*/

template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::
findMin( BinaryNode<Comparable> *t ) const
{
   if( t == NULL )
      return NULL;
   if( t->left == NULL )
      return t;
   return findMin( t->left );
}
```

```
/** Internal method to find the largest item in a
subtree t. Return node containing the largest item. */

template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::
findMax( BinaryNode<Comparable> *t ) const
{
   if( t != NULL )
      while( t->right != NULL )
         t = t->right;
   return t;
}
```

Interface functions of BinarySearchTree

/* Find the smallest item in the tree. Return smallest item or ITEM_NOT_FOUND if empty.*/

```
template <class Comparable>
const Comparable &
BinarySearchTree<Comparable>::findMin( ) const
    {
        return elementAt( findMin( root ) );
    }
```

/** Find the largest item in the tree.

Return the largest item of ITEM_NOT_FOUND if empty.*/

```
template <class Comparable>
const Comparable &
BinarySearchTree<Comparable>::findMax( ) const
    {
        return elementAt( findMax( root ) );
    }
```

/** Find item x in the tree.

* Return the matching item or ITEM_NOT_FOUND if not found. */

```
    template <class Comparable>
    const Comparable &
BinarySearchTree<Comparable>::
    find( const Comparable & x ) const
    {
        return elementAt( find( x, root ) );
    }
```

/* Internal method to get element field in node t. Return the element field or ITEM_NOT_FOUND if t is NULL.*/

```
    template <class Comparable>
    const Comparable &
BinarySearchTree<Comparable>::
elementAt( BinaryNode<Comparable> *t ) const
{
   return t == NULL ? ITEM_NOT_FOUND : t->element;
}
```

– *insert ( )*

- recursive implementation: `(Fig. 4.23)`
- handling duplicate elements

  a.) discard duplicate, or

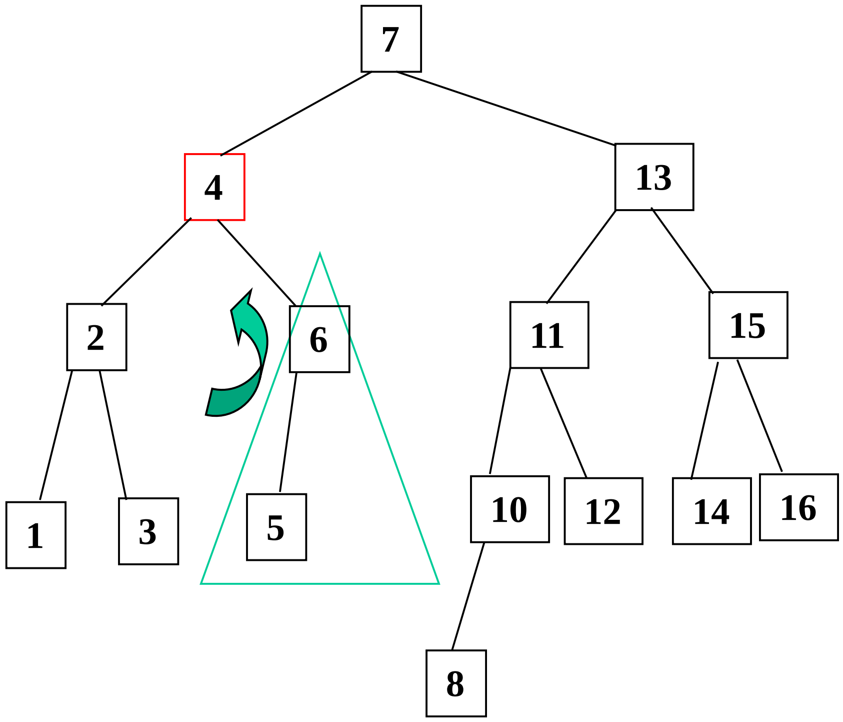
  b.) generate separate nodes, or

  c.) record into the same node.

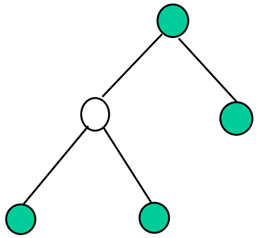
```
/**  Internal method to insert into a subtree. x is the item to insert. t is the node that roots the tree. Set the
new root.   */
        template <class Comparable>
        void BinarySearchTree<Comparable>::
        insert( const Comparable & x, BinaryNode<Comparable> * & t ) const
        {
           if( t == NULL ) t = new BinaryNode<Comparable>( x, NULL, NULL );
           else if( x < t->element )
              insert( x, t->left );
           else if( t->element < x )
              insert( x, t->right );
           else;  // Duplicate; do nothing
        }
```

– *remove ( )*

- Strategy: find the node first, if it is a leaf, delete;
- if it has one child, redirect its parent's link to its child;
- if it has two child, replace the data of this node with the smallest data of its right subtree & recursively delete that node.

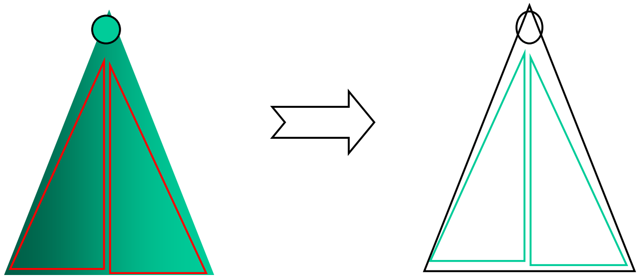


- Implementation: `(Fig. 4.26)`
- Lazy-deletion: keep the node, but mark it as "deleted".



– Destructor & copy assignment operator

- Make Empty: postorder deletion of all nodes `(Fig. 4.27)`
- Copy assignment: `(Fig. 4.28)`



```
/* Internal method to remove from a subtree. x is the item to remove. t is the node that roots the tree. Set the new root. */
template <class Comparable>
void BinarySearchTree<Comparable>::
remove( const Comparable & x, BinaryNode<Comparable> * & t ) const
    {
       if( t == NULL ) return;
       // Item not found; do nothing
       if( x < t->element ) remove( x, t->left );
       else if( t->element < x ) remove( x, t->right );
       else if( t->left != NULL && t->right != NULL )
       // Two children
       {
          t->element = findMin( t->right )->element;
          remove( t->element, t->right );
       } else {
          BinaryNode<Comparable> *oldNode = t;
          t = ( t->left != NULL ) ? t->left : t->right;
          delete oldNode;
       }
    }
```

– Average case analysis

- Most BST operations have running time $O(d)$ where d is the depth of the node to be accessed.
- Internal path length (IPL), $D(N)$, is the sum of the depths of all N nodes in a tree
- $D(N) = D(i) + D(N\text{-}i\text{-}1) + N\text{-}1$
- i: no. of nodes in the left subtree
- $D(i)$ & $D(N\text{-}i\text{-}1)$ is, on average, $\frac{1}{N}\sum_{j=0}^{N-1} D(j)$

$$\Rightarrow D(N) = \frac{2}{N}\sum_{j=0}^{N-1} D(j) + N - 1 = O(N \log N)$$ $\Rightarrow$ Average depth $O(logN)$

- Example: `(Fig. 4.29)`
- Deletion can possibly alter the balance of the BST (Fig.4.30)
- Changing the deletion strategy may solve the problem.

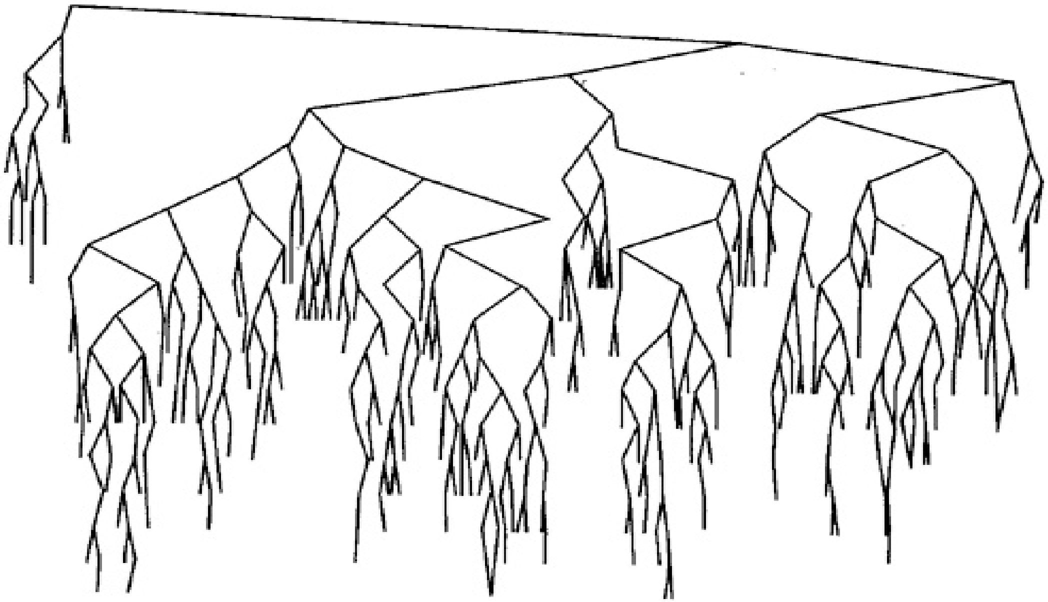


Figure 4.29 A randomly generated binary search tree

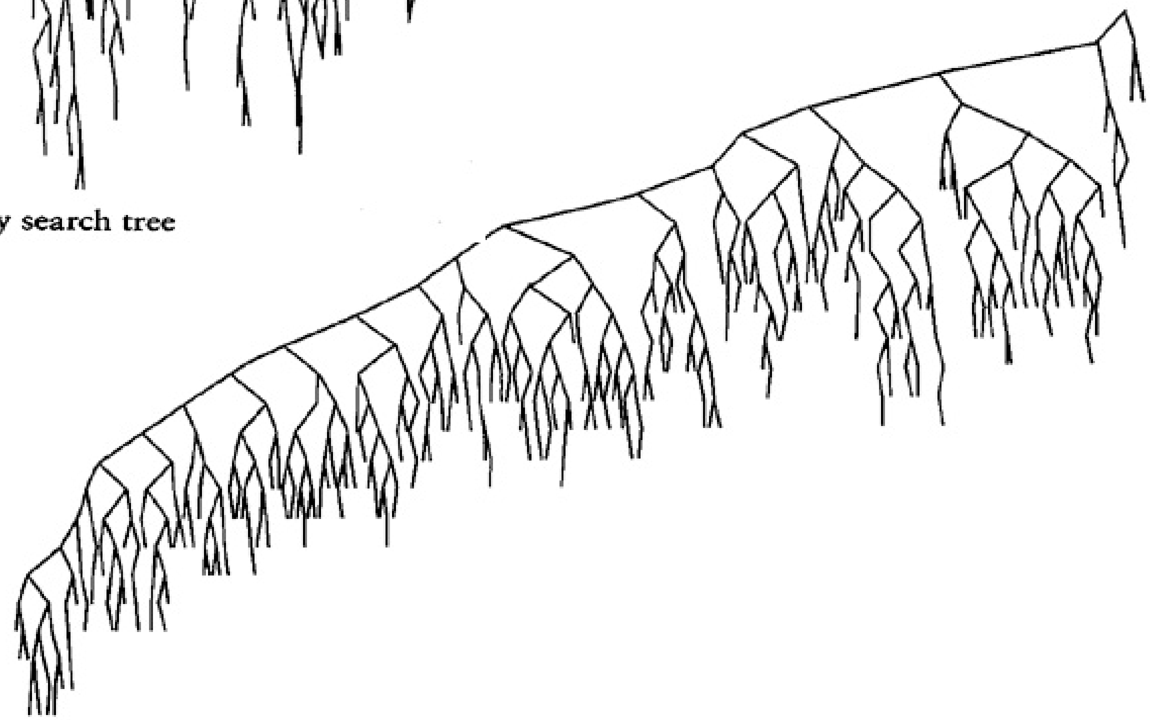
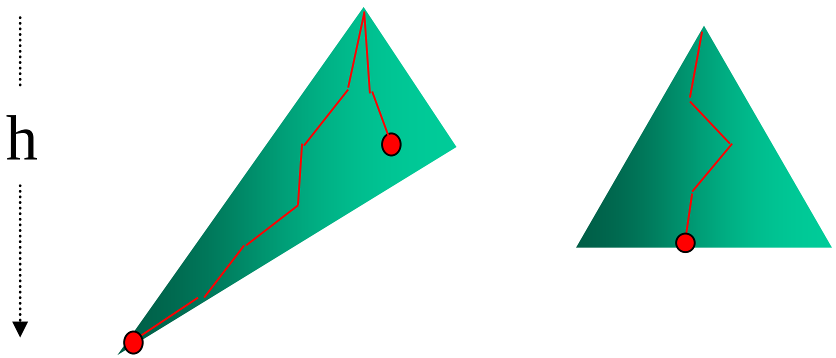
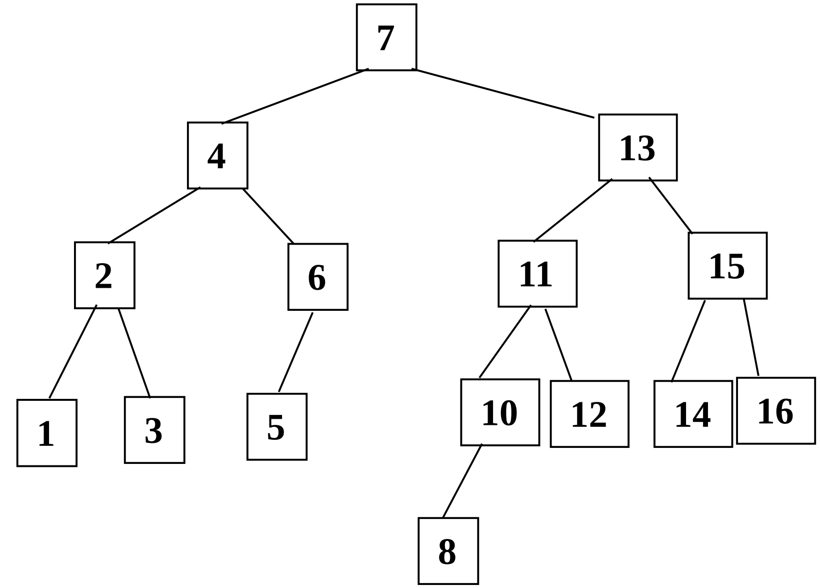


Figure 4.30 Binary search tree after $\Theta(N^2)$ insert/remove pairs

4.4 AVL Trees

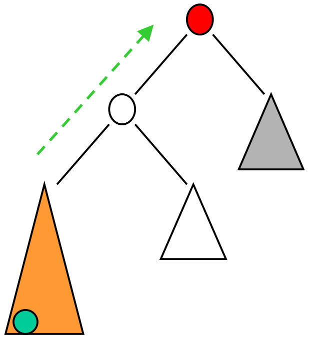


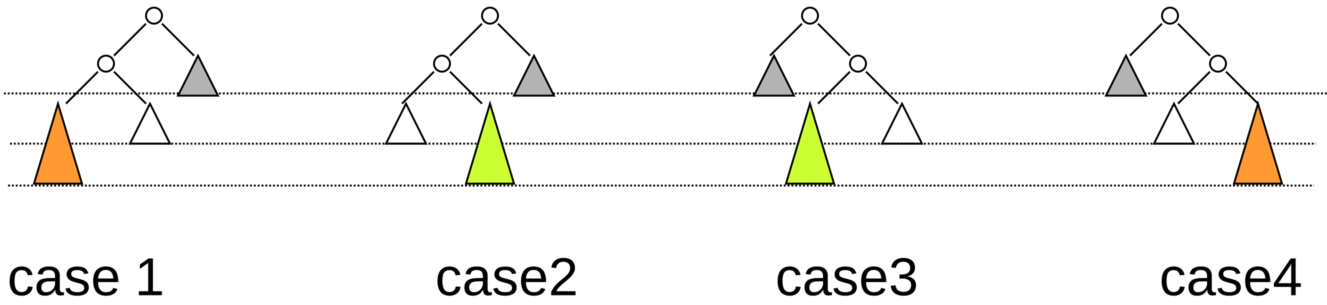


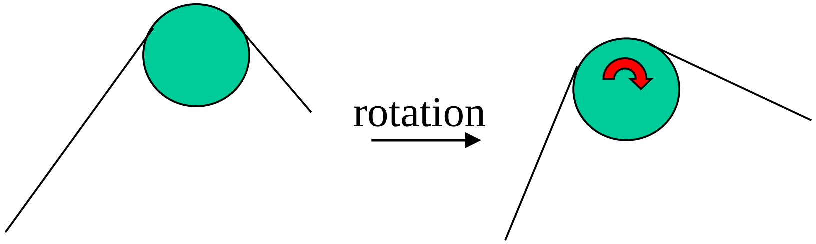
- Definitions
  - An AVL tree is a binary search tree with the restriction that for every node in the tree, the heights of the left & right subtrees can diff by at most 1.
  - The height of an AVL tree is at most $1.44(N+2)-0.328$, the best case is $O(logN)$, and on average just slightly above $logN$
  - $S(h)$: the minimum number of nodes in an AVL tree of height h.

    (Fig. 4.33) $S(h) = S(h\text{-}1) + S(h\text{-}2) +1$

- Insertions
  - Insertion requires modifications of the AVL tree (rotation operations) to maintain the AVL height requirement.
  - Only nodes on the path from the insertion point to the root may have their balance violated.
  - Follow this path upward to the root and fix the first (deepest) unbalanced node. There are four cases:

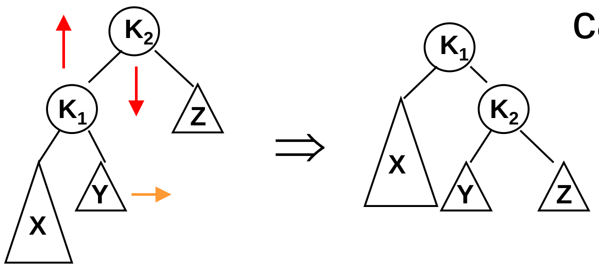




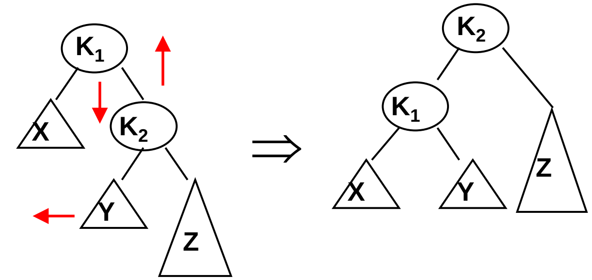


– Single rotation

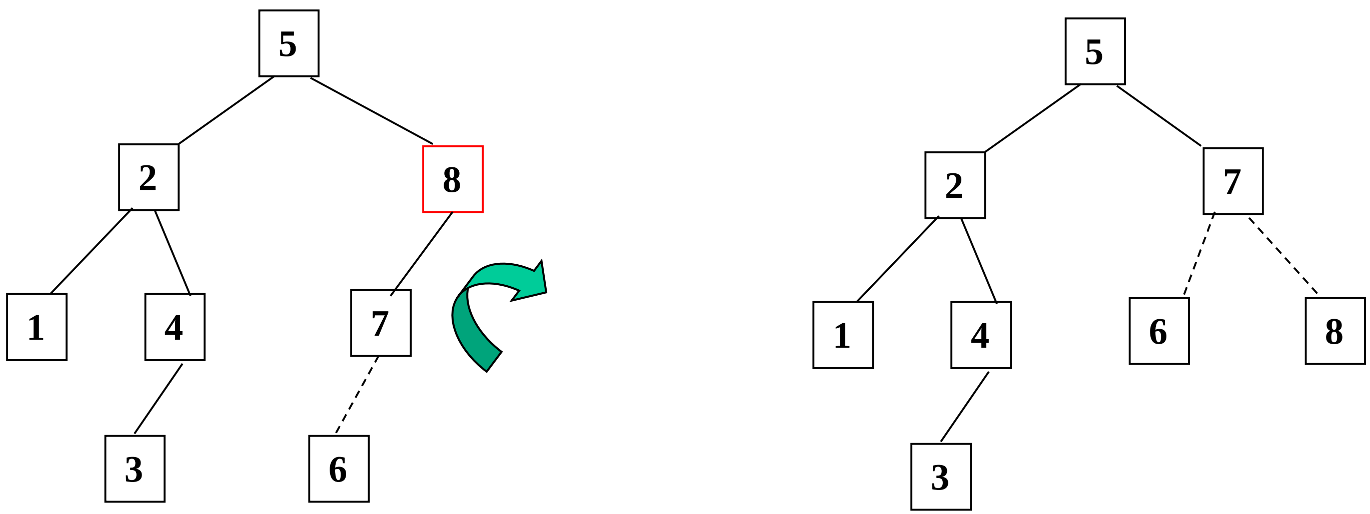
- case 1:



case 4:

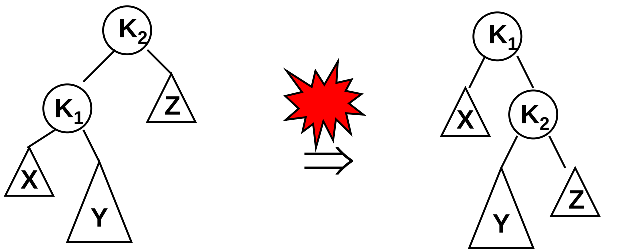


- since the balanced subtree has left & right subtree of the same height, which is also the height of the subtree before insertion, no more rebalancing will be needed for nodes above it.
- example: `(pp147-148)`

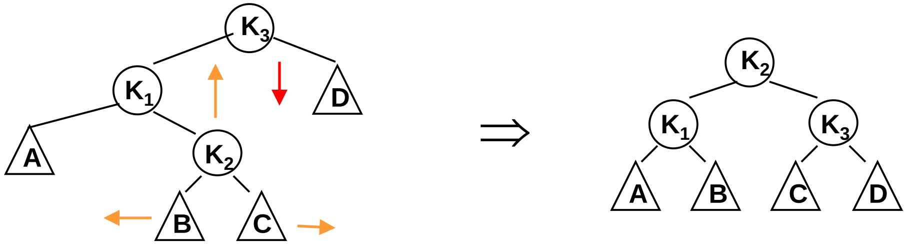


– Double rotation

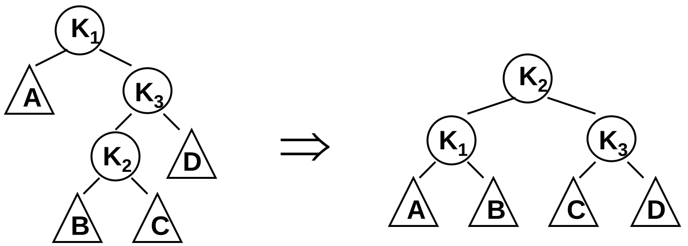
- single rotation fails with case 2 & 3



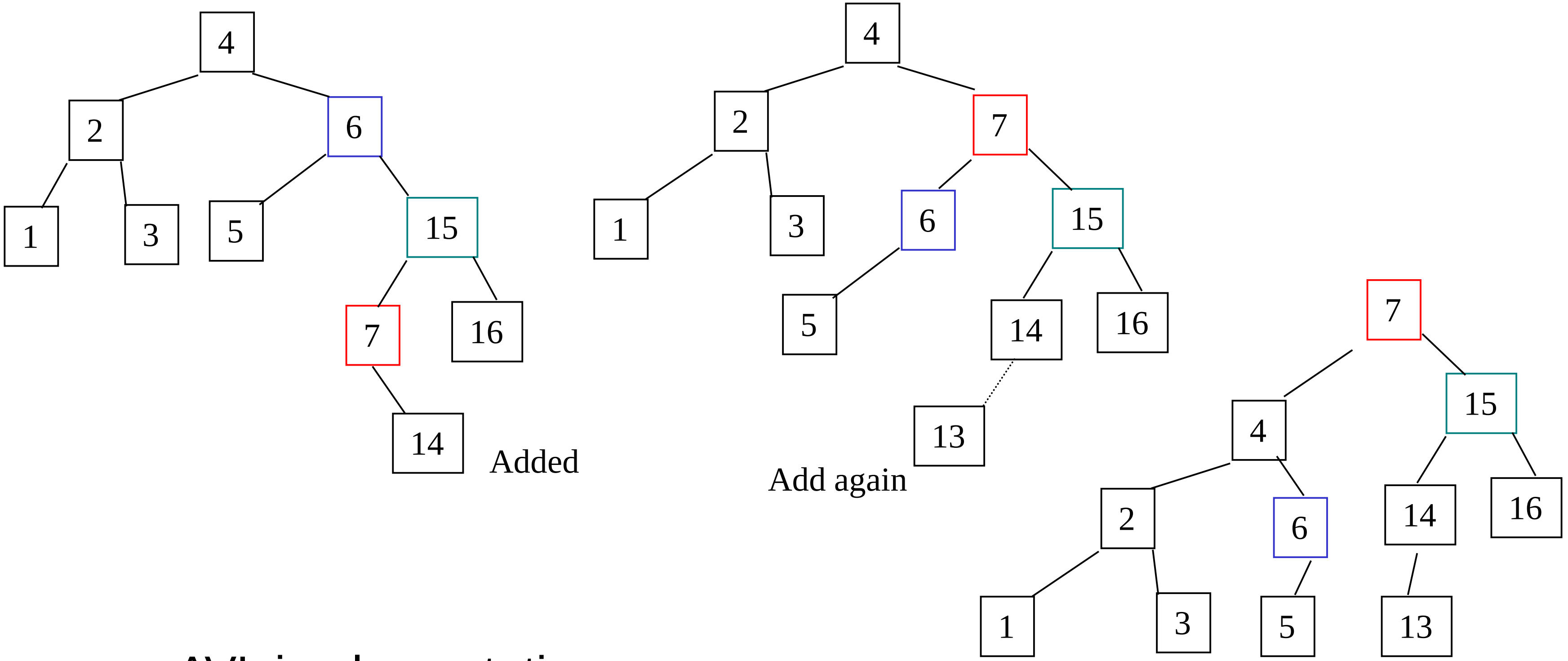
- double rotation for case 2:



- double rotation for case 3:

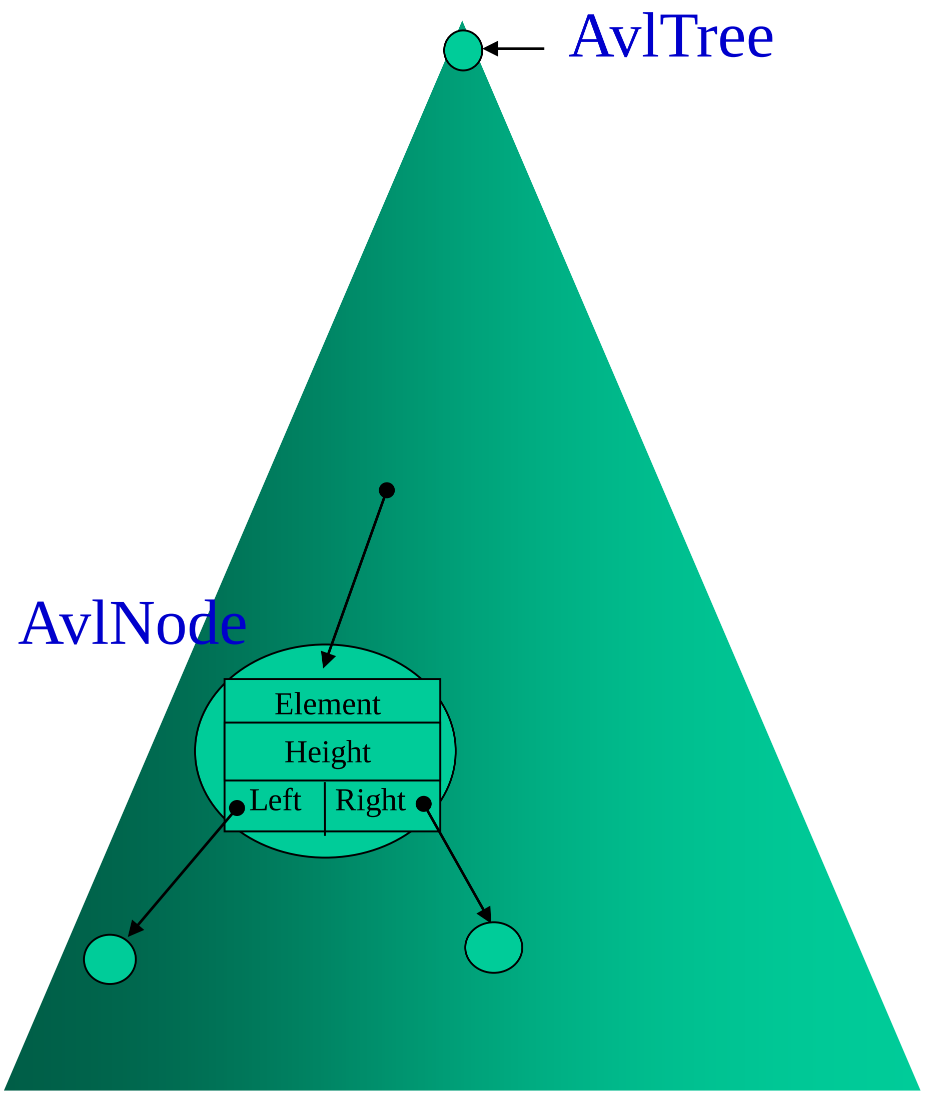


example: `(pp. 149-151)`



- AVL implementation
  - AVL tree: `(Fig. 4.40, Fig. 4.41)`
  - Insertion: `(Fig. 4.42 - 4.46)`
- Deletion
  - Lazy-deletion is commonly used in AVL tree
  - True deletion can also be implemented similar to insertion

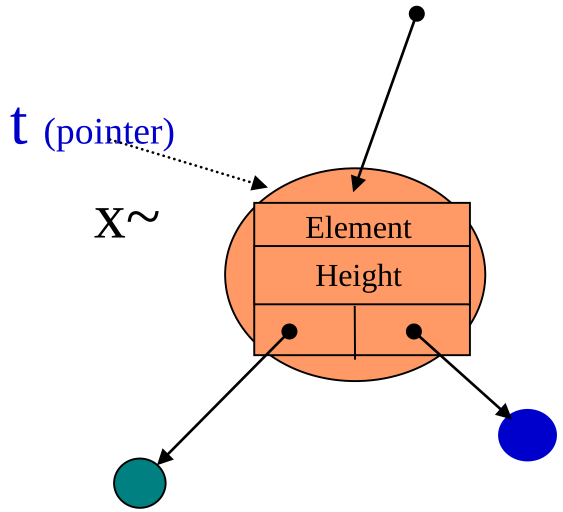
```
#ifndef _AVL_TREE_H_
#define _AVL_TREE_H_
  // Node and forward declaration because g++ does
  // not understand nested classes.
      template <class Comparable>
      class AvlTree;
      template <class Comparable>
      class AvlNode
      {
          Comparable element;
          AvlNode   *left;
          AvlNode   *right;
          int       height;
AvlNode( const Comparable & theElement, AvlNode *lt,
AvlNode *rt, int h = 0 )
: element( theElement ), left( lt ), right( rt ), height( h ) { }
          friend class AvlTree<Comparable>;
      };
```

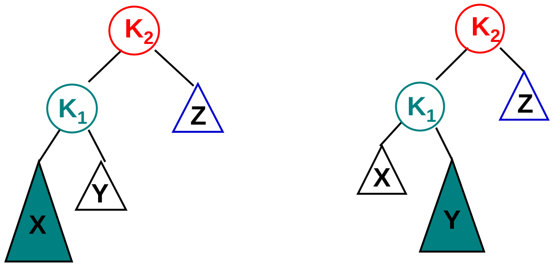


```
template <class Comparable>
class AvlTree {
    public:
      explicit AvlTree( const Comparable &   notFound );
      AvlTree( const AvlTree & rhs );
      ~AvlTree( );
      const Comparable & findMin( ) const;
      const Comparable & findMax( ) const;
      const Comparable & find( const Comparable & x ) const;
      bool isEmpty( ) const;
      void printTree( ) const;
      void makeEmpty( );
      void insert( const Comparable & x );
      void remove( const Comparable & x );
      const AvlTree & operator=( const AvlTree & rhs );
```

```
private:
        AvlNode<Comparable> *root;
        const Comparable ITEM_NOT_FOUND;
        const Comparable & elementAt( AvlNode<Comparable> *t ) const;
        void insert( const Comparable & x, AvlNode<Comparable> * & t ) const;
        AvlNode<Comparable> * findMin( AvlNode<Comparable> *t ) const;
        AvlNode<Comparable> * findMax( AvlNode<Comparable> *t ) const;
        AvlNode<Comparable> * find( const Comparable & x, AvlNode<Comparable> *t ) const;
        void makeEmpty( AvlNode<Comparable> * & t ) const;
        void printTree( AvlNode<Comparable> *t ) const;
        AvlNode<Comparable> * clone( AvlNode<Comparable> *t ) const;
           // Avl manipulations
        int height( AvlNode<Comparable> *t ) const;
        int max( int lhs, int rhs ) const;
        void rotateWithLeftChild( AvlNode<Comparable> * & k2 ) const;
        void rotateWithRightChild( AvlNode<Comparable> * & k1 ) const;
        void doubleWithLeftChild( AvlNode<Comparable> * & k3 ) const;
        void doubleWithRightChild( AvlNode<Comparable> * & k1 ) const;
    };
```

```
/**Internal method to insert into a subtree. x is the item to insert. t is the node that roots the tree. */
    template <class Comparable>
    void AvlTree<Comparable>::insert( const Comparable & x, AvlNode<Comparable> * & t ) const
    {
      if( t == NULL )
        t = new AvlNode<Comparable>( x, NULL, NULL );
      else if( x < t->element )
      {
        insert( x, t->left );
        if( height( t->left ) - height( t->right ) == 2 )
          if( x < t->left->element ) rotateWithLeftChild( t );
          else  doubleWithLeftChild( t );
      } else
      if( t->element < x ) {
          insert( x, t->right );
          if( height( t->right ) - height( t->left ) == 2 )
              if( t->right->element < x ) rotateWithRightChild( t );
          else  doubleWithRightChild( t );
      }
      else;  // Duplicate; do nothing
      t->height = max( height( t->left ), height( t->right ) ) + 1;
    }
```



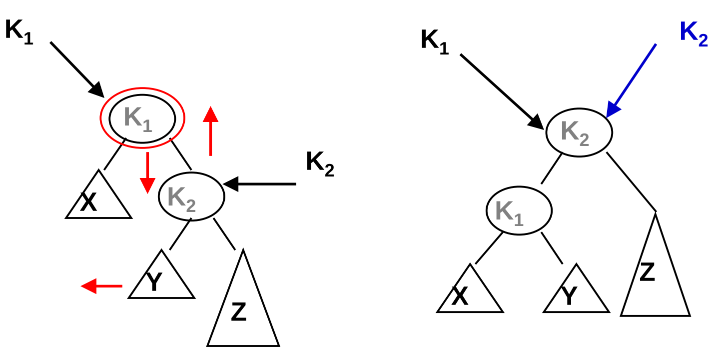


Single rotation

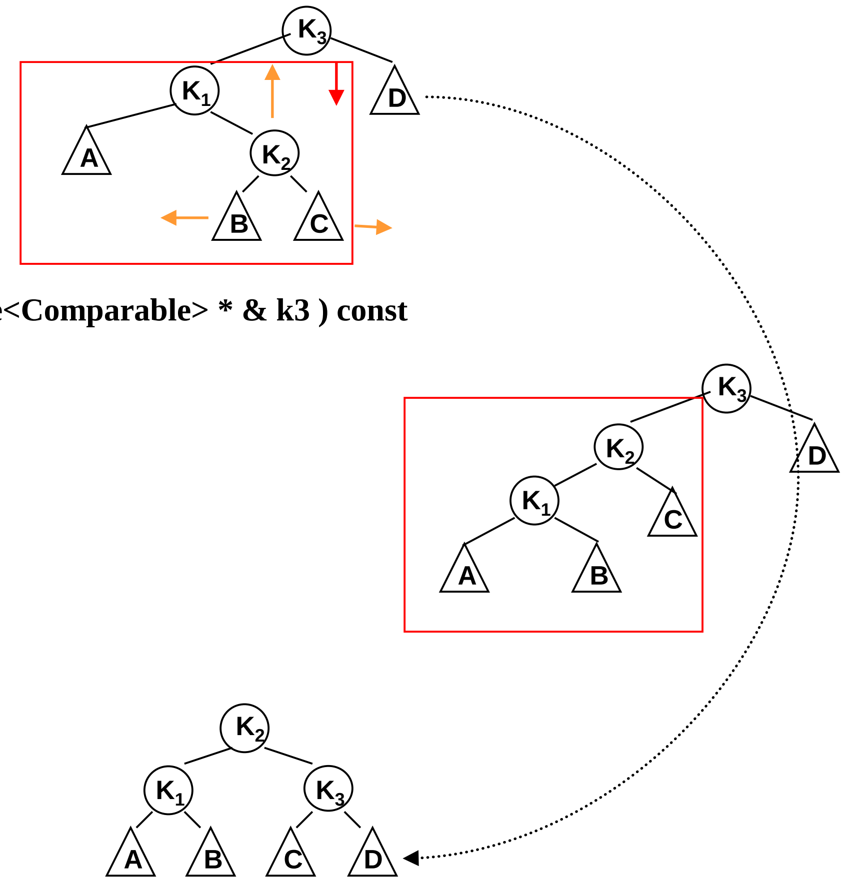
Double rotation

```
/* Rotate binary tree node with left child.
For AVL trees, this is a single rotation for case 1.
    Update heights, then set new root.
  */
   template <class Comparable>
   void
AvlTree<Comparable>::rotateWithLeftChild(
AvlNode<Comparable> * & k2 ) const
   {
     AvlNode<Comparable> *k1 = k2->left;
     k2->left = k1->right;
     k1->right = k2;
     k2->height = max( height( k2->left ),
                       height( k2->right ) ) + 1;
     k1->height = max( height( k1->left ),
                       k2->height ) + 1;
     k2 = k1;
   }
```

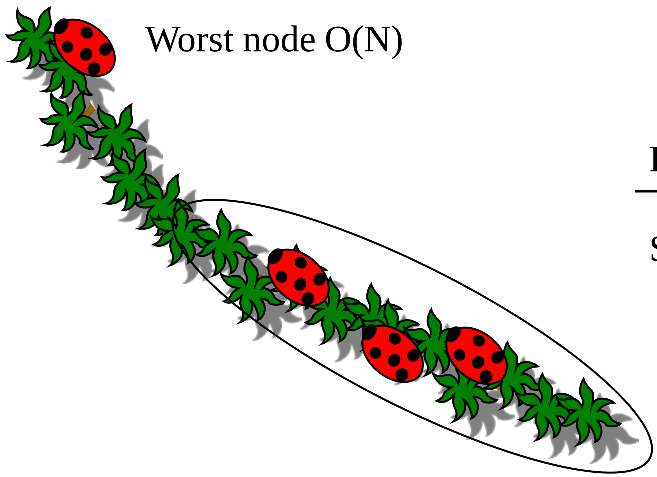
```
/** Rotate binary tree node with right child.
* For AVL trees, this is a single rotation for case 4.
 * Update heights, then set new root. */
   template <class Comparable>
   void AvlTree<Comparable>::
rotateWithRightChild( AvlNode<Comparable> * & k1 ) const
 {
     AvlNode<Comparable> *k2 = k1->right;
     k1->right = k2->left;
     k2->left = k1;
     k1->height = max( height( k1->left ), height( k1->right ) ) + 1;
     k2->height = max( height( k2->right ), k1->height ) + 1;
     k1 = k2;
}
```



```
/* Double rotate binary tree node: first left child.
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.        */
template <class Comparable>
void AvlTree<Comparable>::doubleWithLeftChild( AvlNode<Comparable> * & k3 ) const
{
   rotateWithRightChild( k3->left );
   rotateWithLeftChild( k3 );
}

/* Double rotate binary tree node: first right child.
 * with its left child; then node k1 with new right child.
 * For AVL trees, this is a double rotation for case 3.
 * Update heights, then set new root.        */
template <class Comparable>
void AvlTree<Comparable>::doubleWithRightChild( AvlNode<Comparable> * & k1 ) const
{
   rotateWithLeftChild( k1->right );
   rotateWithRightChild( k1 );
}
```

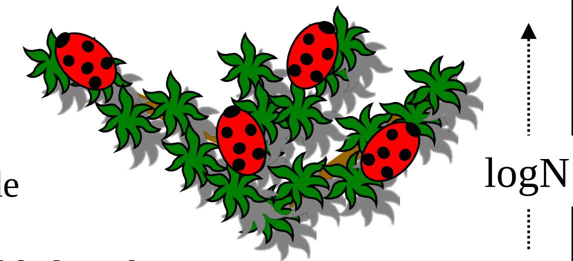


4.5 Splay trees



Reconstruction

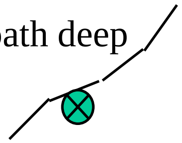
Splaying: move the worst node



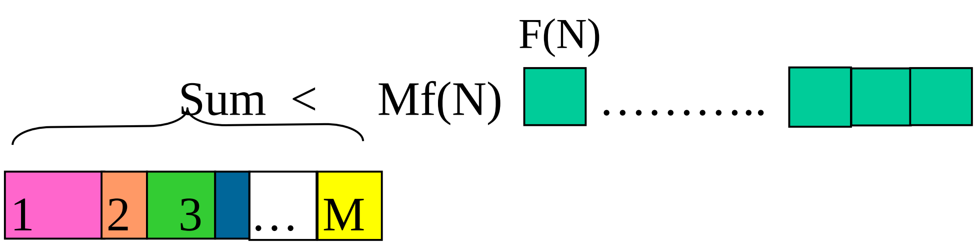
→ Accessed node is likely to be accessed again in the near future

→ Move the node to the root

→ Simply rotation push another node on the path deep
SUM = N+(N-1)+(N-2)+…. ~$O(N^2)$



→ New rotation method is needed
(each rotation reduce height to N/2, then
N + N/2 + N/4 + …~ MLog(N))



4.5 Splay trees

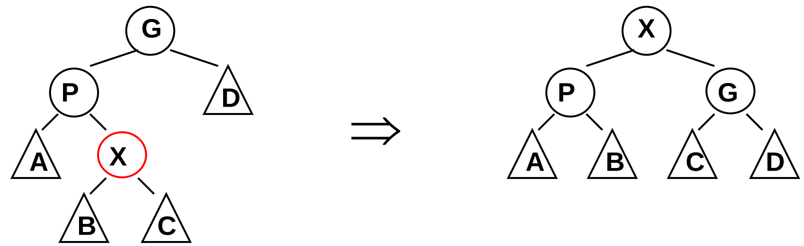
– Basics
  - If a sequence of M operations has total worst case running time $O(Mf(n))$, we say the <u>amortized</u> running time is $O(f(n))$.
  - A splay tree has an $O(logN)$ amortized cost per operation. The worst case single operation can still be $O(N)$, but the tree will self-adjust, through a sequence of M operations, to guarantee that the total cost is $O(MlogN)$
  - Easy to maintain (no height restrictions)

– Splaying
  - Basic idea: when a node is accessed, it will be pushed to the root by a sequence of rotations along the access path.
  - Let X be a (non-root) node on the access path at which we are rotating.

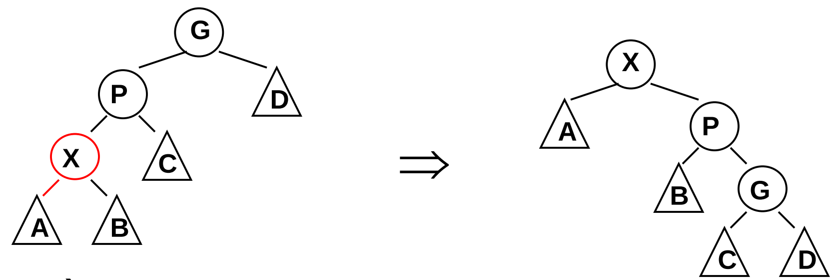
    If the parent of X is root, rotate X and the root by a AVL single rotation (this is the last rotation along the path).

    Otherwise:

case 1 (zig-zag case):



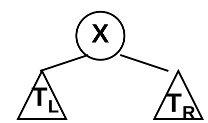
case 2 (zig-zig case):



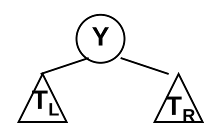
– Examples: `(pp. 158-162)`

– Deletion

a.) access the node to be deleted, which will push the node to the root



b.) access the largest node, Y, in T_L, which will push Y to the root of T_L, and then link T_R to the right child link of Y.



4.7 B-trees

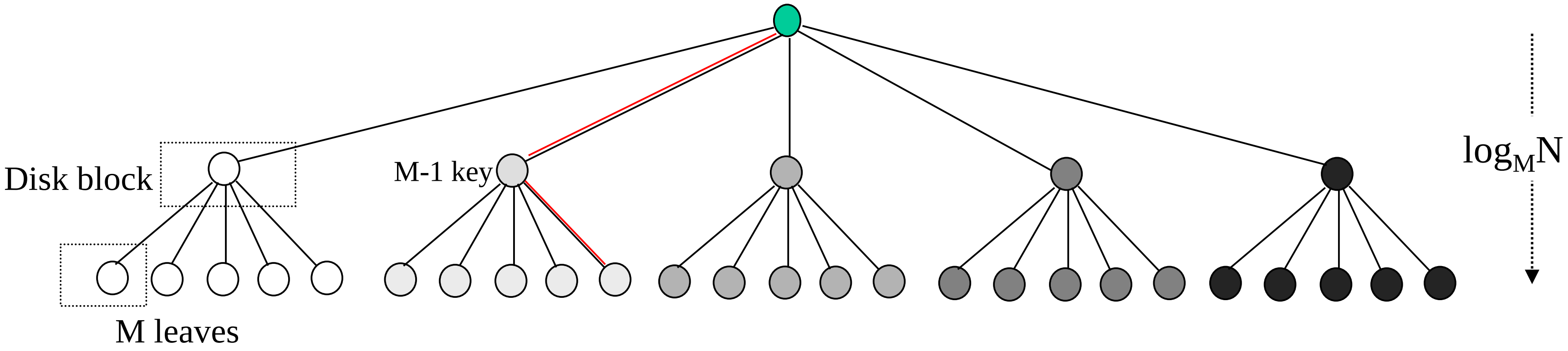
- Disk-access *vs*. memory access
  - each disk access worth *~200,000* machine instructions.
  - reduce the number of disk access to a very small constant, using complicated data structures & algorithms (since machine instructions are virtually free compared to disk-access)
- B-tree
  - A B-tree of order M is an M-*ary* tree with conditions:

  1.) data items are stored at leaves

  2.) non-leaf nodes store up to M-*1* keys to guide the searching;

  key i represents the smallest key in subtree $i+1$

  3.) the root is either a leaf or has between two and M children



4.) all non-leaf nodes (except root) have between $\lceil M/2 \rceil$ and M children.

5.) all leaf nodes are at the same depth and have between $\lceil L/2 \rceil$ and L children for some L.

- Each node represents a dish block, and is guaranteed to be at least half full, which presents it from degenerating into a simple binary tree.
- M & L are chosen according to data size, key size, and disk block size.

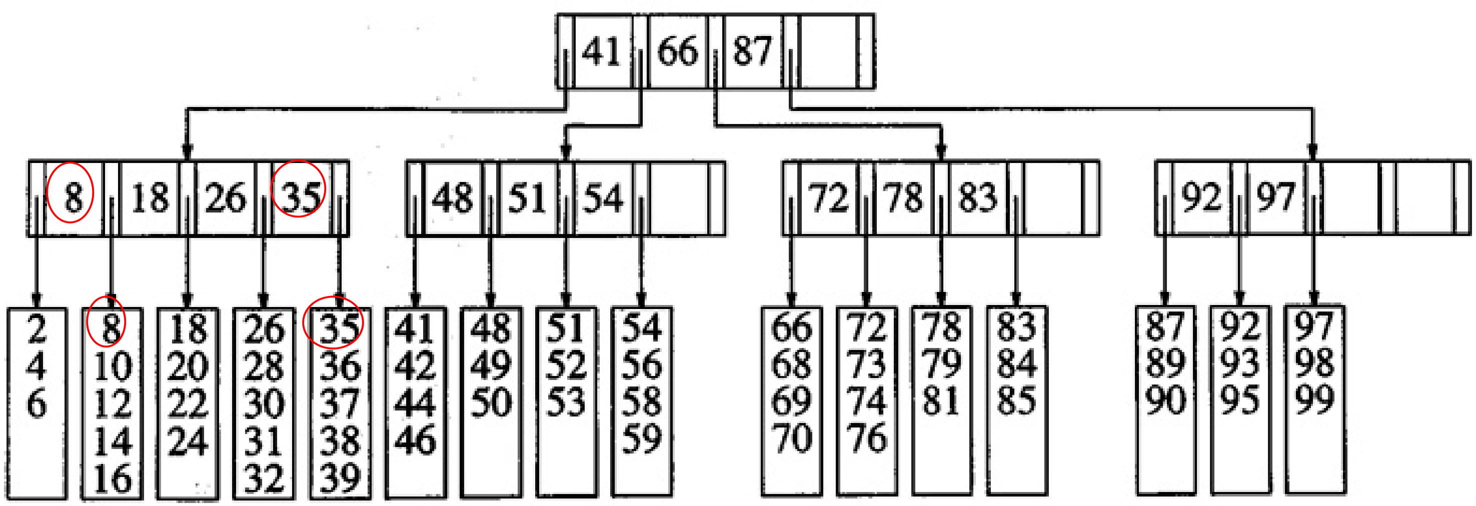


Figure 4.62 B-tree of order 5

- Example:

  block-size = *8192* bytes,

  key-size = *32* bytes,

  data-record-size = *256* bytes

  each non-leaf node needs $32(M-1)+4M$ bytes.

  $\Rightarrow$ the largest M is *228*

  Each leaf node needs *256* bytes

  $\Rightarrow$ the largest L is *32* (*8192*/*256*)

  If we have up to *10,000,000* data records, at most *625,000* leaves can be used.

  $\Rightarrow$ leaf nodes will have depth at most *4* (or, in general, $\log_{\frac{M}{2}} N$)

  If we store the root the *1st* level in main memory/cache, only *2* disk accesses are needed.

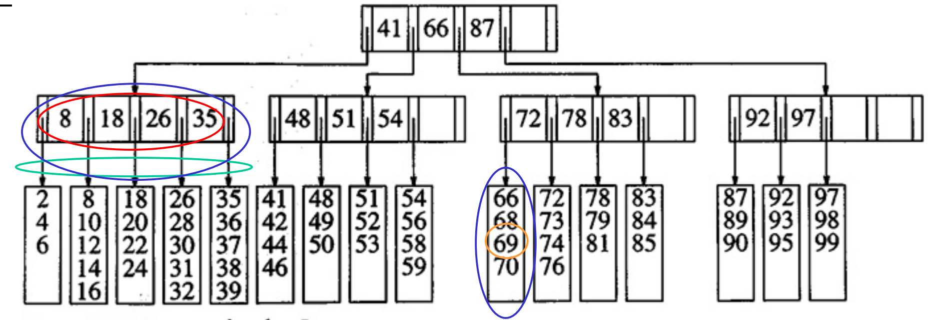
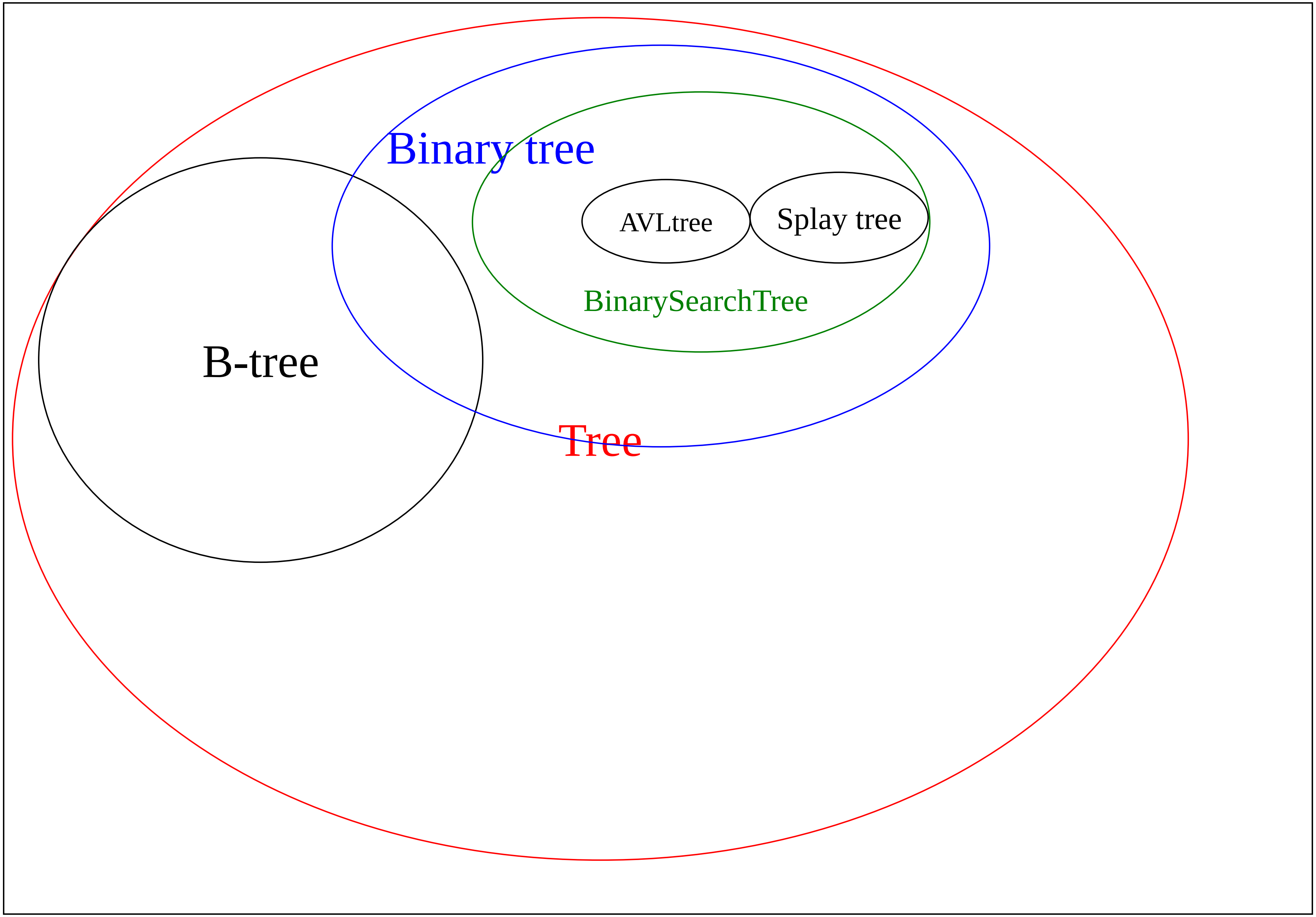


Figure 4.62 B-tree of order 5

- B-trees will be revisited in file structures.


Binary tree
AVLtree
Splay tree
BinarySearchTree
B-tree
Tree