Learning Resources

Text Books:

Horowitz, Sahani, Dinesh Mehta, “Fundamentals of Data Structures in C++”, Galgotia Publisher, ISBN: 8175152788, 9788175152786.

Peter Brass, “Advanced Data Structures”, Cambridge University Press, ISBN: 978-1-107-43982.

Reference Books:

Sartaj Sahani, “Data Structures, Algorithms and Applications in C++”, Second Edition, University Press, ISBN: 81-7371522 X.

Augenstein ,Tenenbaum & Langsam,“Data Structure Using C & C++”,PHI Publication.

Supplementary Reading:

Yashwant Kanitkar,” Data Structures through C++”, BPB Publication.

Learning Resources contd…

Web Resources:

https://www.khanacademy.org/computing/computer-science/algorithms

https://www.hackerrank.com/contests/basic-ds-quiz-1/

Web links:

https://www.tutorialspoint.com/data_structures_algorithms/

MOOCs:

http://nptel.ac.in/courses/106102064/1

https://nptel.ac.in/courses/106103069/

Symbol Table

- Introduction to Symbol Tables
- Static tree table- Optimal Binary Search Tree (OBST)
- Dynamic tree table-AVL tree
- Multiway search tree- B-Tree

Symbol Table

- Symbol table is defined as a name-value pair

- Symbol tables are data structures that are used by compilers to hold information about source-program constructs.

Symbol Tables

Associates **attributes with identifiers used in a program**

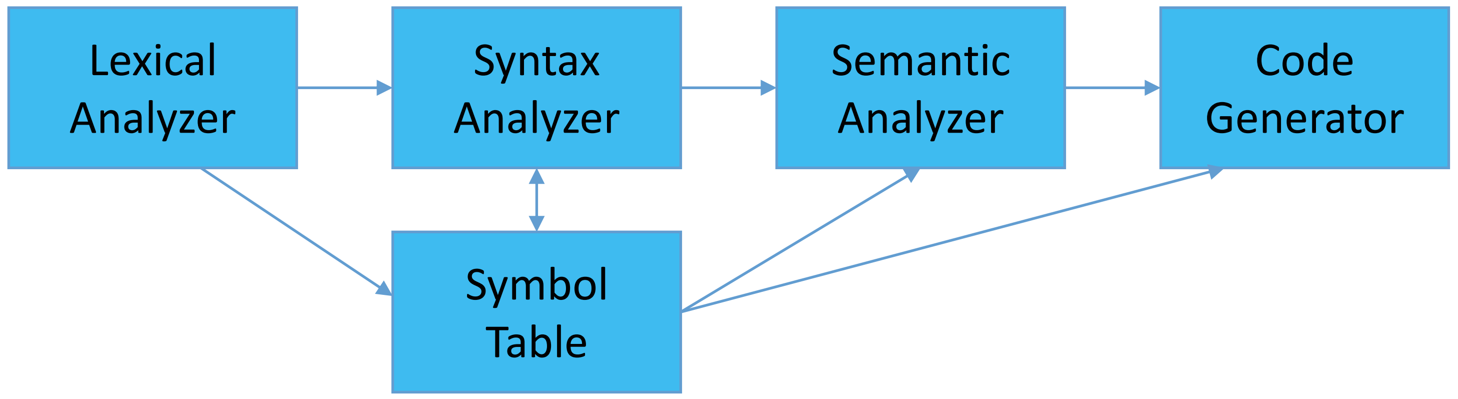
- For instance, a **type attribute is usually associated with each identifier**
- A symbol table is a necessary component
  - Definition (declaration) of identifiers appears once in a program
  - Use of identifiers may appear in many places of the program text
- Identifiers and attributes are entered by the analysis phases
  - When processing a definition (declaration) of an identifier
  - In simple languages with only global variables and implicit declarations: **The scanner can enter an identifier into a symbol table if it is not already there**
  - In block-structured languages with scopes and explicit declarations: **The parser and/or semantic analyzer enter identifiers and corresponding attributes**

Symbol Tables

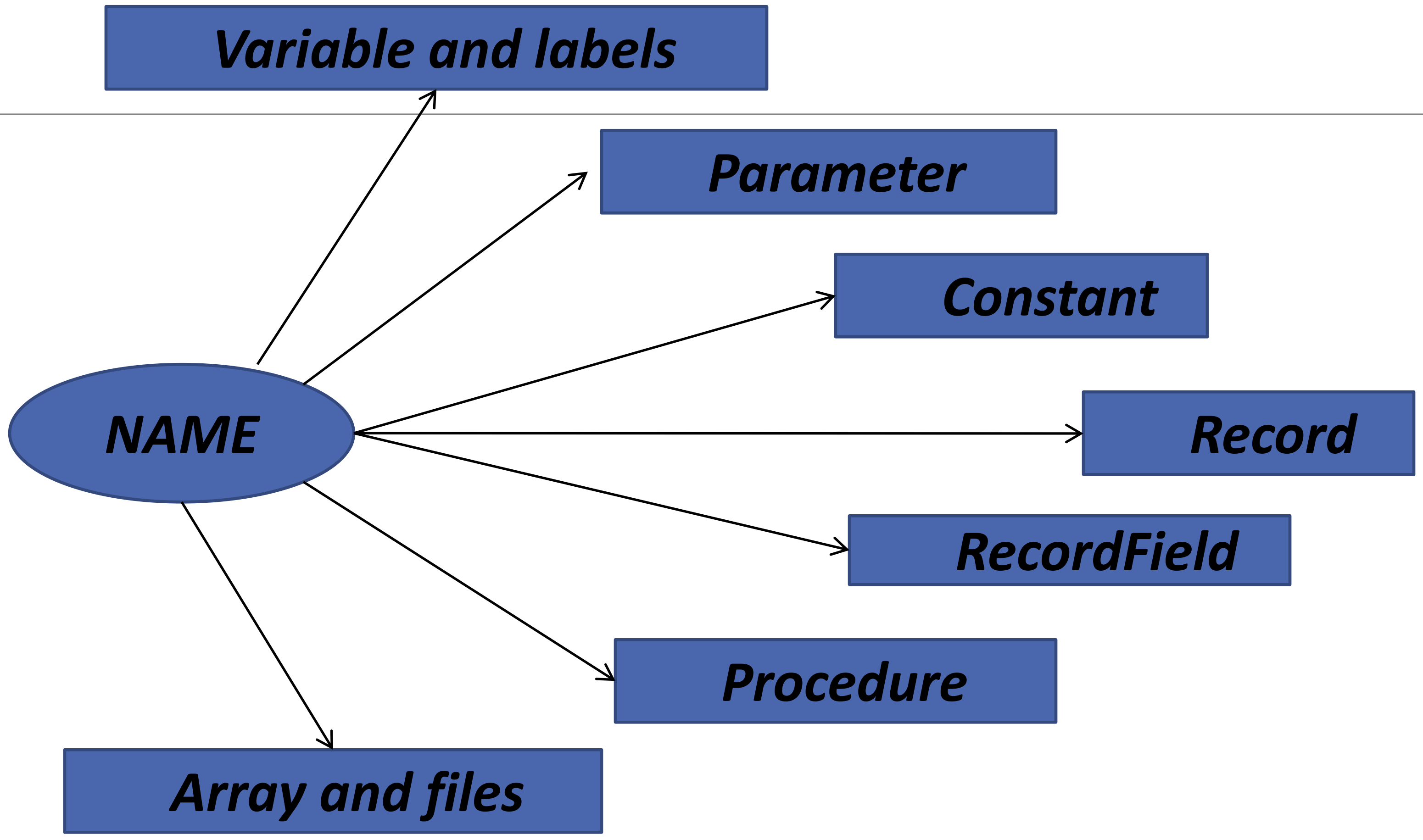
- Symbol table information is used by the analysis and synthesis phases
  - To verify that used identifiers have been defined (declared)
  - To verify that expressions and assignments are semantically correct – **type checking**
  - To generate intermediate or target code

The Symbol Table

- When identifiers are found, they will be entered into a symbol table, which will hold all relevant information about identifiers.
- This information will be used later by the semantic analyzer and the code generator.



SYMBOL TABLE - NAMES



SYMBOL TABLE-ATTRIBUTES

- Each piece of information associated with a name is called an ***attribute.***
- Attributes are language dependent.
- Different classes of Symbols have different Attributes

Variable, Constants	Procedure or function	Array
• Type , Line number where declared , Lines where referenced , Scope	• Number of parameters, parameters themselves, result type.	• # of Dimensions, Array bounds.

Symbol Table

- While compilers and assemblers are scanning a program, each identifier must be examined to determine if it is a keyword
- This information concerning the keywords in a programming language is stored in a symbol table
- Symbol table is a kind of ‘keyed table’
- The keyed table stores <key, information> pairs with no additional logical structure

Symbol Table(Contd..)

- The operations performed on symbol tables are the following:
- Insert the pairs <key, information> into the collection
- Remove the pairs <key, information> by specifying the key
- Search for a particular key
- Retrieve the information associated with a key

Possible implementations:

- Unordered list: for a very small set of variables.
  - Simplest to implement
  - Implemented as an array or a linked list
  - Linked list can grow dynamically – alleviates problem of a fixed size array
  - Insertion is fast O(1), but lookup is slow for large tables – O(n) on average.
- Ordered linear list:
  - If an array is sorted, it can be searched using binary search – *O(log2 n)*
  - Insertion into a sorted array is expensive – *O(n) on average*
  - Useful when set of names is known in advance – table of reserved words

Possible implementations:

- Binary search tree:
  - Can grow dynamically
  - Insertion and lookup are *O(log2 n) on average*

Representation of Symbol Table

There are two different techniques for implementing a keyed table:

- Static Tree Tables

- Dynamic Tree Tables

Static Tree Tables

- When symbols are known in advance and no insertion and deletion is allowed, it is called a static tree table

- An example of this type of table is a reserved word table in a compiler

Dynamic Tree Table

- A dynamic tree table is used when symbols are not known in advance but are inserted as they come and deleted if not required
- Dynamic keyed tables are those that are built on-the-fly
- The keys have no history associated with their use

Optimal Binary Search Trees

- To optimize a table knowing what keys are in the table and what the probable distribution is of those that are not in the table, we build an optimal binary search tree (OBST)
- Optimal binary search tree is a binary search tree having an average search time of all keys optimal
- An OBST is a BST with the minimum cost

Optimal binary search trees

A full binary tree may not be an optimal binary search tree if the identifiers are searched for with different frequency

Consider these two search trees,

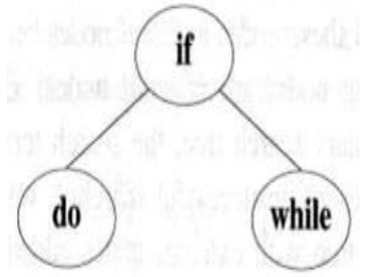
If we search for each identifier with equal Probability

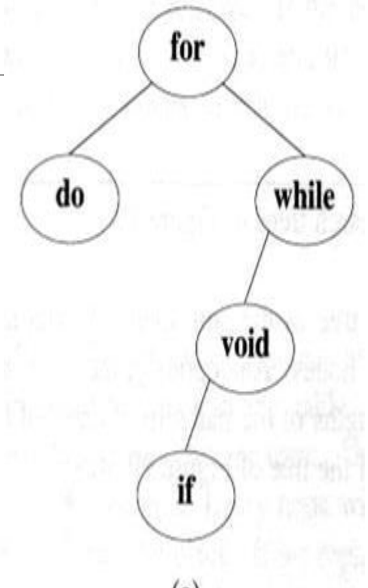
In first tree (a)

- the average number of comparisons for successful search is 2.4.
- Comparisons for second tree is 2.2.

The second tree (b) has

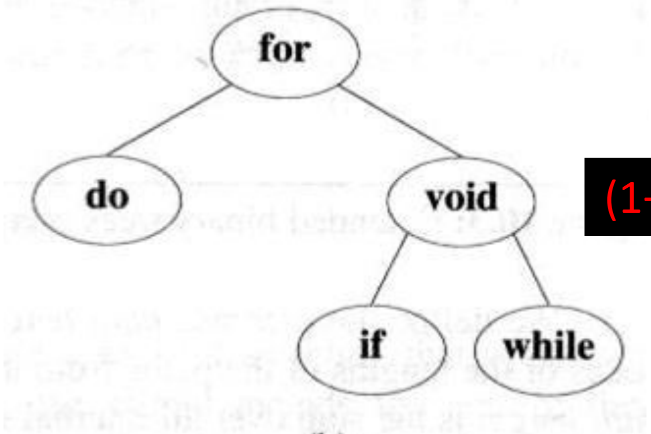
- a better worst case search time than the first tree.
- a better average behavior.





(a)

(1+2+2+3+4)/5 = 2.4



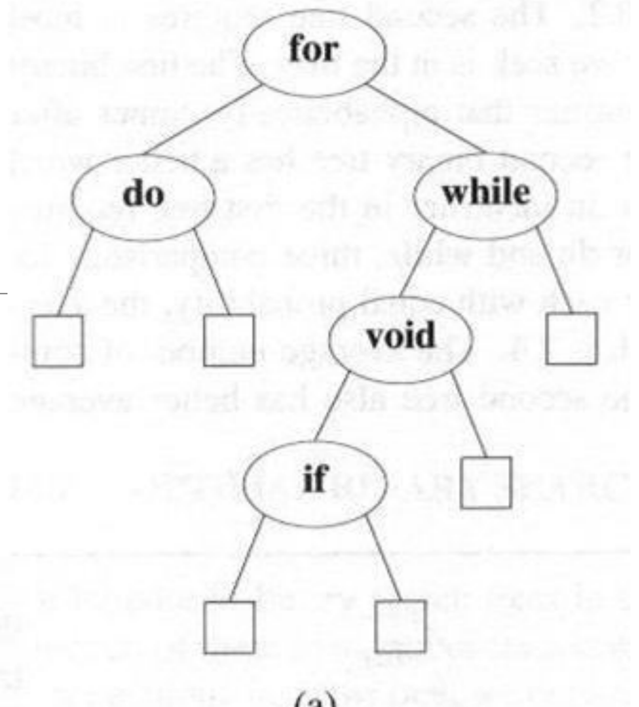
(b)

(1+2+2+3+3)/5 = 2.2

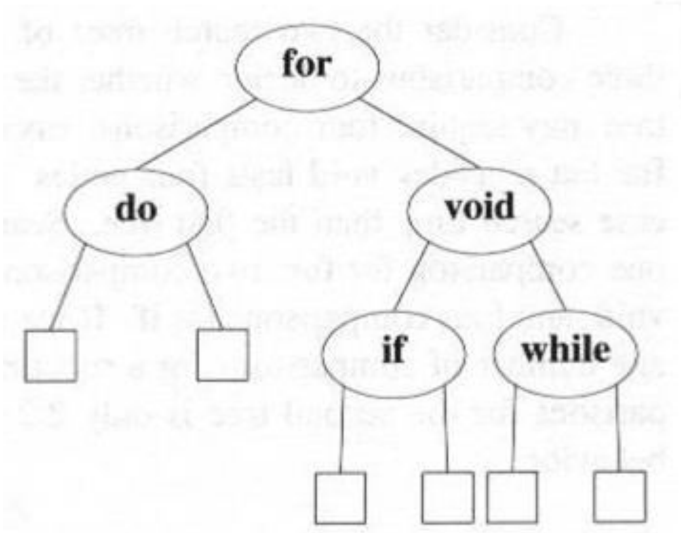
Optimal binary search trees

In evaluating binary search trees, it is useful to add a special square node at every place there is a null links.

- We call these nodes *external nodes*.
- We also refer to the external nodes as *failure* nodes.
- The remaining nodes are *internal nodes*.
- A binary tree with external nodes added is an *extended binary tree*



(a)



(b)

Optimal binary search trees

External / internal path length

- The sum of all external / internal nodes' levels.

For example

- Internal path length, I, is:

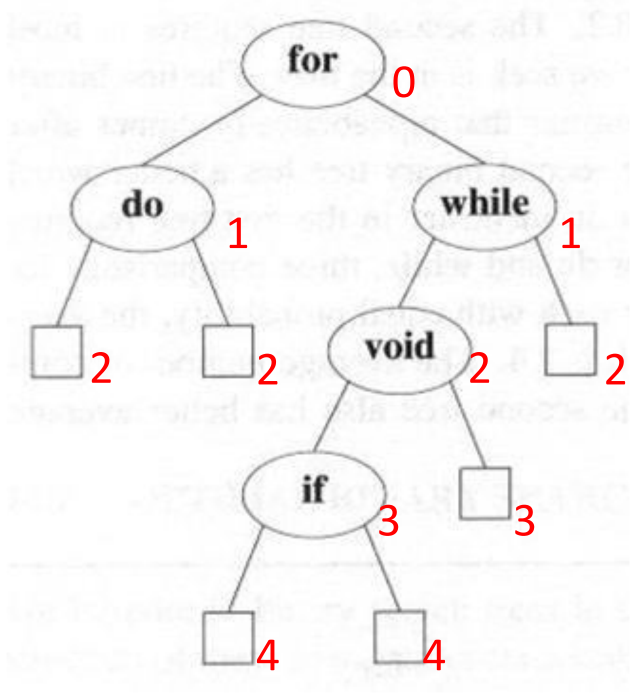
$$I = 0 + 1 + 1 + 2 + 3 = 7$$

- External path length, E, is :

$$E = 2 + 2 + 4 + 4 + 3 + 2 = 17$$

A binary tree with n internal nodes are related

by the formula : $E = I + 2n$



Optimal binary search trees

In the binary search tree:

- The identifiers $a_1, a_2, \ldots, a_n$ with $a_1 < a_2 < \ldots < a_n$
- The probability of searching for each a_i is p_i
- The total cost (when only successful searches are made) is:

$$\sum_{i=1}^{n} p_i \cdot \text{level}(a_i)$$

Optimal binary search trees

If we replace the null subtree by a failure node, we may partition the identifiers that are not in the binary search tree into n+1 classes E_i, $0 \leq i \leq n$

- E_i contains all identifiers x such that $a_i < x < a_{i+1}$
- For all identifiers in a particular class, E_i, the search terminates at the same failure node

Optimal binary search trees

We number the failure nodes form 0 to *n* with *i* being for class E_i, $0 \le i \le n$.

- If q_i is the probability that the identifier we are searching for is in E_i, then the cost of the failure node is:

$$\sum_{i=0}^{n} q_i \cdot (\text{level}(\text{failure node } i) - 1)$$

Therefore, the total cost of a binary search tree is:

$$\sum_{i=1}^{n} p_i \cdot \text{level}(a_i) + \sum_{i=0}^{n} q_i \cdot (\text{level (failure node } i) - 1)$$

- An optimal binary search tree for the identifier set $a_1, \ldots, a_n$ is one that minimizes
  - Since all searches must terminate either successfully or unsuccessfully, we have

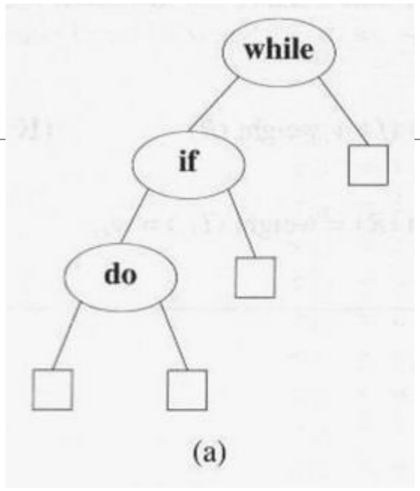
$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Optimal binary search trees

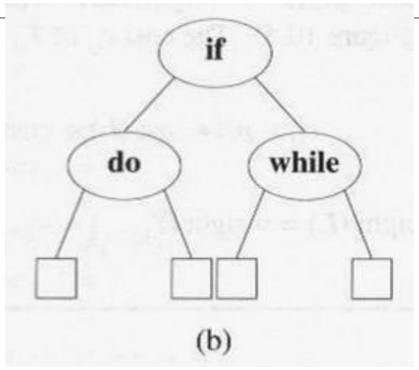
The possible binary search trees for the identifier set (a_1, a_2, a_3) = (**do, if, while**)

- The identifiers with equal probabilities, $p_i = a_j = 1/7$ for all i, j,
  - cost(tree *a*) = 15/7;
  - cost(tree *b*) = 13/7; (optimal)
    cost(tree *c*) = 15/7;
  - cost(tree *d*) = 15/7;
    cost(tree *e*) = 15/7;

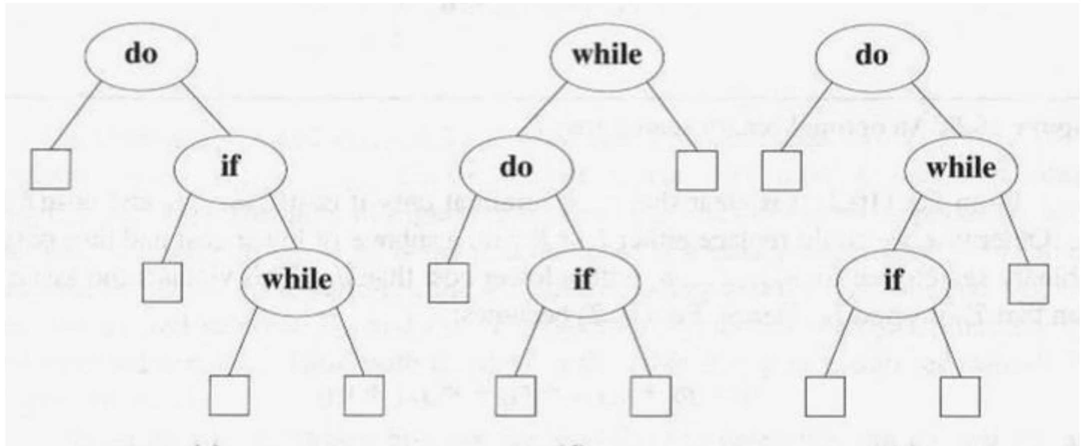
- $p_1 = 0.5$, $p_2 = 0.1$, $p_3 = 0.05$,
  $q_0 = 0.15$, $q_1 = 0.1$, $q_2 = 0.05$, $q_3 = 0.05$
  - cost(tree *a*) = 2.65;
    cost(tree *b*) = 1.9;
    cost(tree *c*) = 1.5; (optimal)
    cost(tree *d*) = 2.05;
    cost(tree *e*) = 1.6;



(a)



(b)



(c) (d) (e)

OPTIMAL BINARY SEARCH TREES (Contd..)

With equal probability P(i)=Q(i)=1/7.

Let us find an OBST out of these.

Cost(tree a)=∑P(i)*level a(i) +∑Q(i)*level (Ei) -1

1≤i≤n 0≤i≤n

(2-1) (3-1) (4-1) (4-1)

=1/7[1+2+3 + 1 + 2 + 3 + 3] = 15/7

Cost (tree b) =17[1+2+2+2+2+2+2] =13/7

Cost (tree c) =cost (tree d) =cost (tree e) =15/7

∴ tree b is optimal.

OPTIMAL BINARY SEARCH TREES (Contd..)

If P(1) =0.5 ,P(2) =0.1, P(3) =0.005 , Q(0) =.15 , Q(1) =.1, Q(2) =.05 and Q(3) =.05 find the OBST.

Cost (tree a) = .5 x 3 +.1 x 2 +.05 x 3 +.15x3 +.1x3 +.05x2 +.05x1 = 2.65

Cost (tree b) =1.9 , Cost (tree c) =1.5 ,Cost (tree d) =2.05 ,

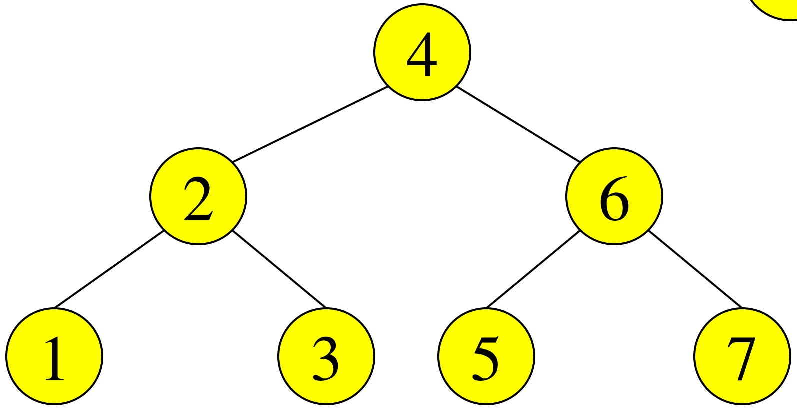
Cost (tree e) =1.6 Hence tree C is optimal.

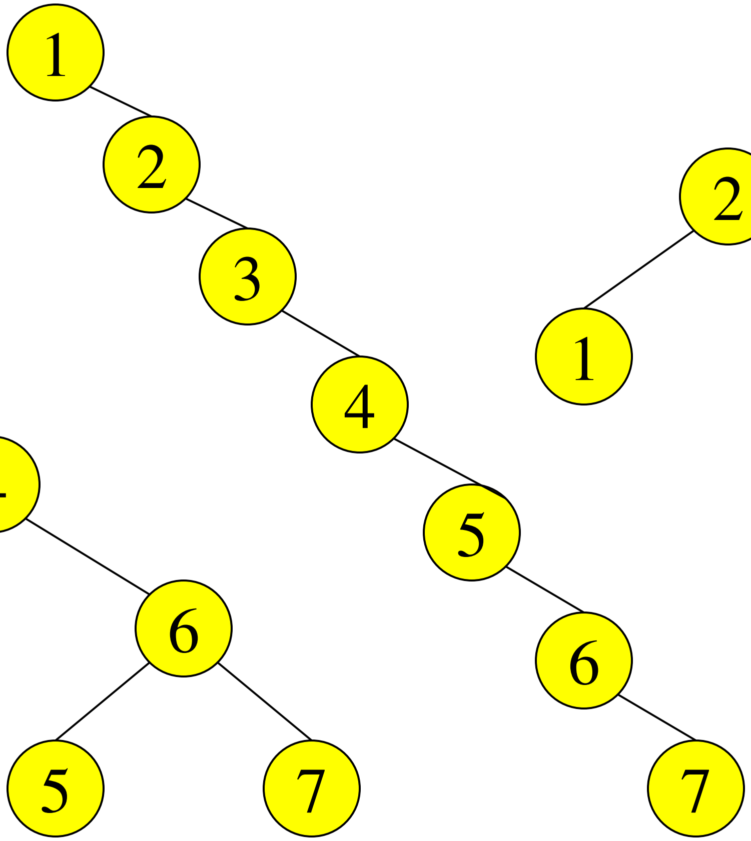
Binary Search Tree - Worst Time

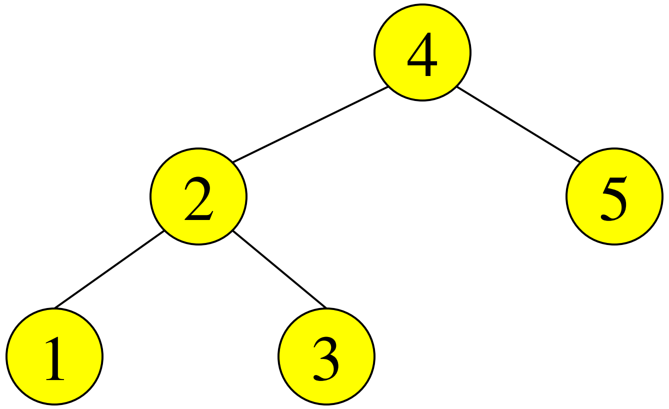
Worst case running time is O(N)

- What happens when you Insert elements in ascending order?
  - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
- Problem: Lack of "balance":
  - compare depths of left and right subtree
- Unbalanced degenerate tree

Balanced and unbalanced BST







Is this “balanced”?

Balancing Binary Search Trees

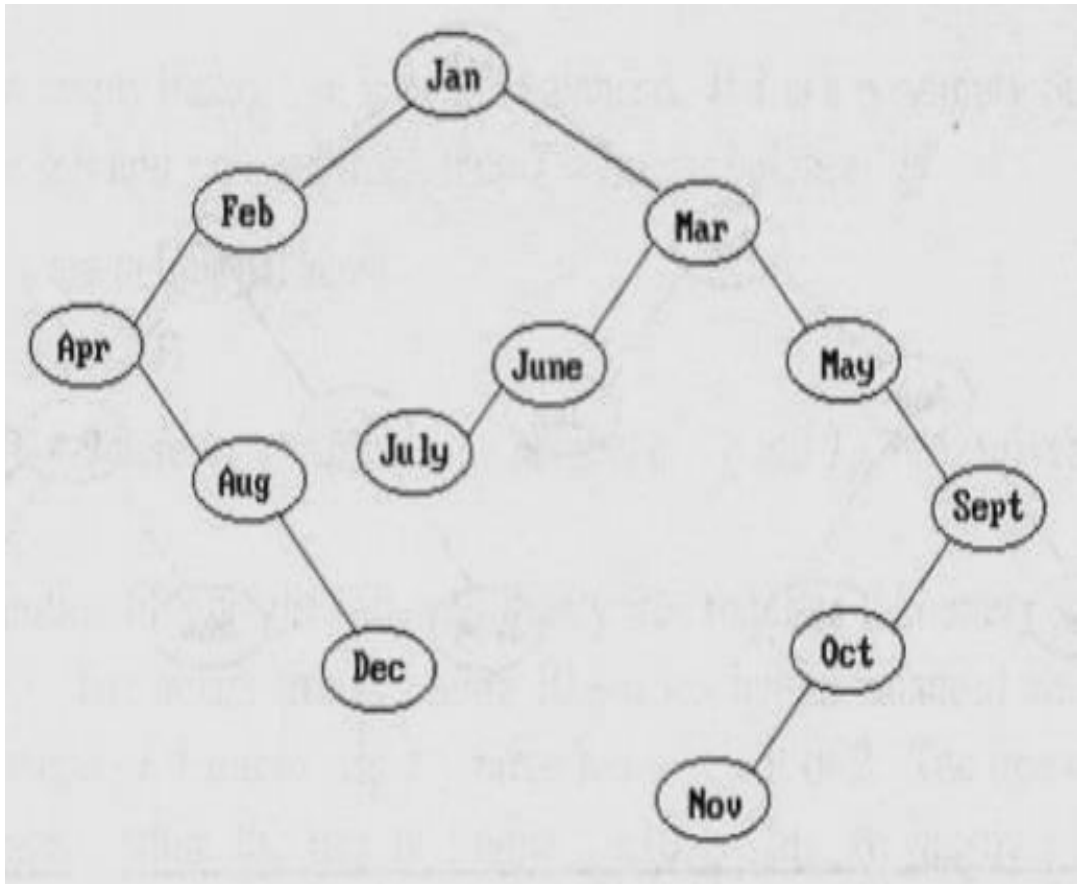
Many algorithms exist for keeping binary search trees balanced

- Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
- Splay trees and other self-adjusting trees
- B-trees and other Multiway search trees

AVL Trees

We also may maintain dynamic tables as binary search trees.

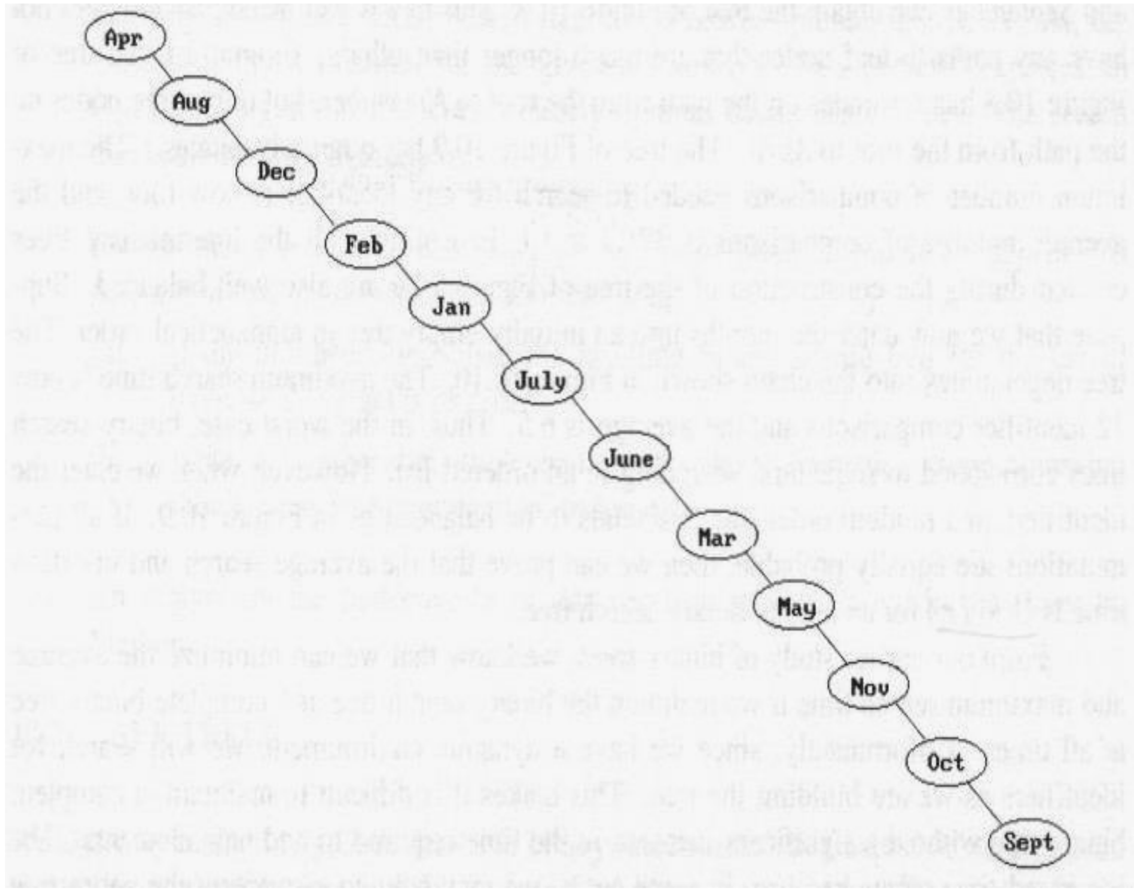
- The binary search tree obtained by entering the months *January* to *December*, in that order, into an initially empty binary search tree
- The maximum number of comparisons needed to search for any identifier in the tree (for *November*).
- Average number of comparisons is 42/12 = 3.5



AVL Trees

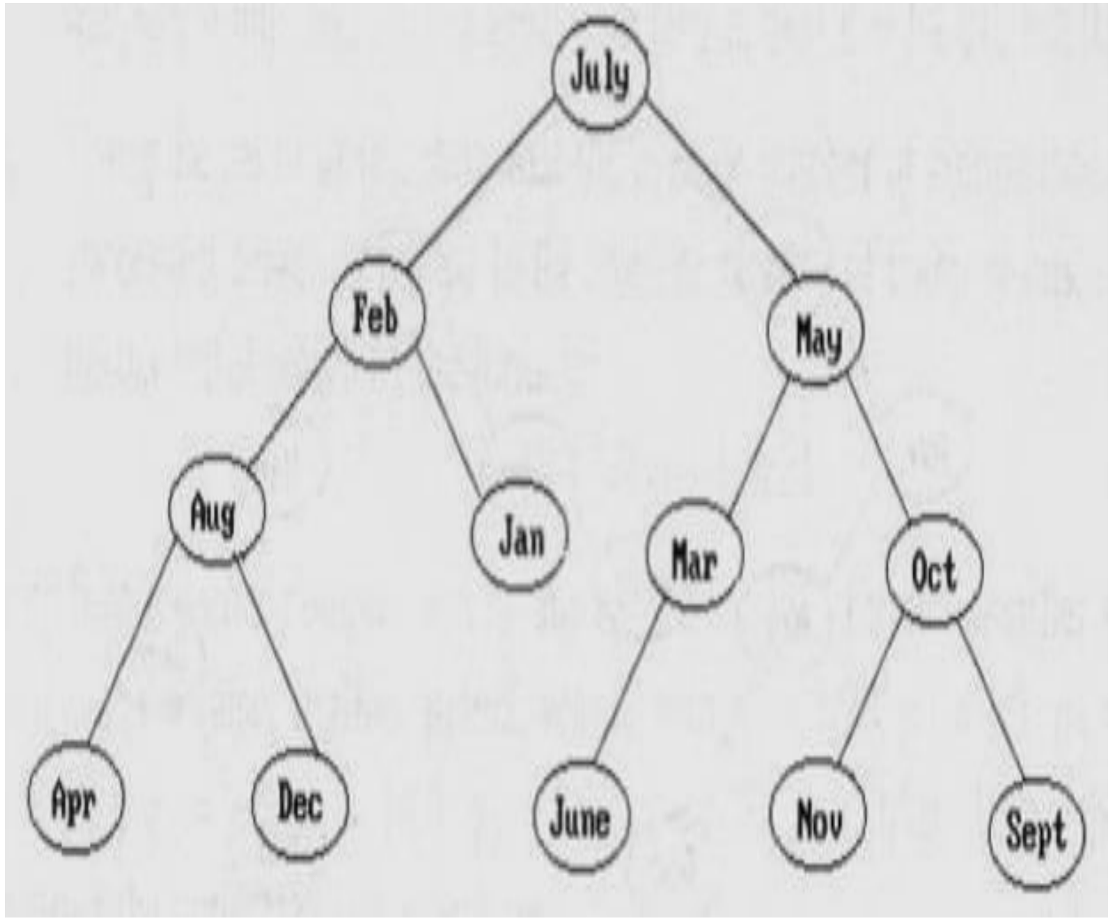
Suppose that we now enter the months into an initially empty tree in alphabetical order

- The tree degenerates into the chain number of comparisons:
  maximum: 12,   and average: 6.5
- in the worst case, binary search trees correspond to sequential searching in an ordered list



Another insert sequence

- In the order *Jul, Feb, May, Aug, Jan, Mar, Oct, Apr, Dec, Jun, Nov,* and *Sep*
- Well balanced and does not have any paths to leaf nodes that are much longer than others.
- Number of comparisons: maximum: 4, and average: 37/12 ≈ 3.1.
- All intermediate trees created during the construction of Figure are also well balanced



AVL Trees

Adelson-Velskii and Landis introduced a binary tree structure (*AVL trees*):

- Balanced with respect to the heights of the subtrees.
- We can perform dynamic retrievals in O(log*n*) time for a tree with n nodes.
- We can enter an element into the tree, or delete an element form it, in O(log*n*) time. The resulting tree remain height balanced.
- As with binary trees, we may define AVL tree recursively

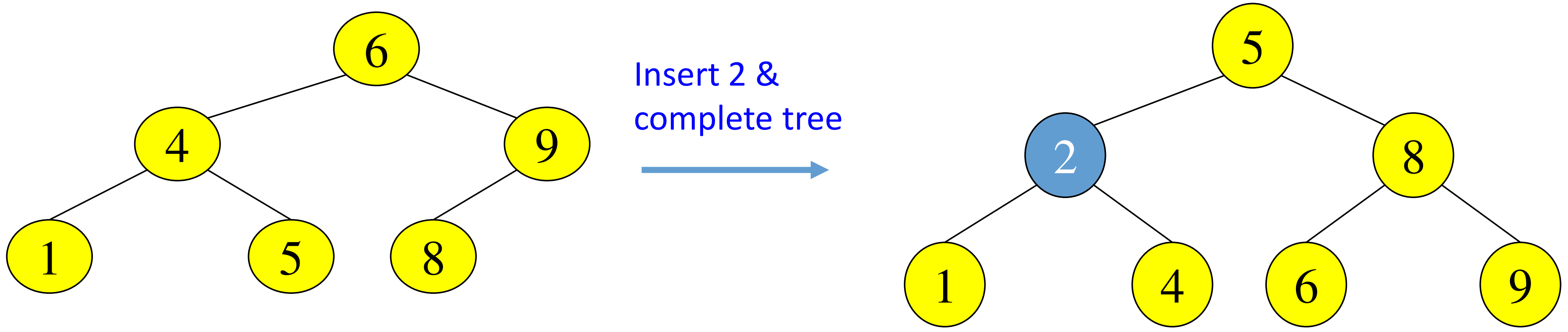
Perfect Balance

Want a complete tree after every operation

- tree is full except possibly in the lower right

This is expensive

- For example, insert 2 in the tree on the left and then rebuild as a complete tree



AVL - Good but not Perfect Balance

AVL trees are height-balanced binary search trees

Balance factor of a node

- height(left subtree) - height(right subtree)

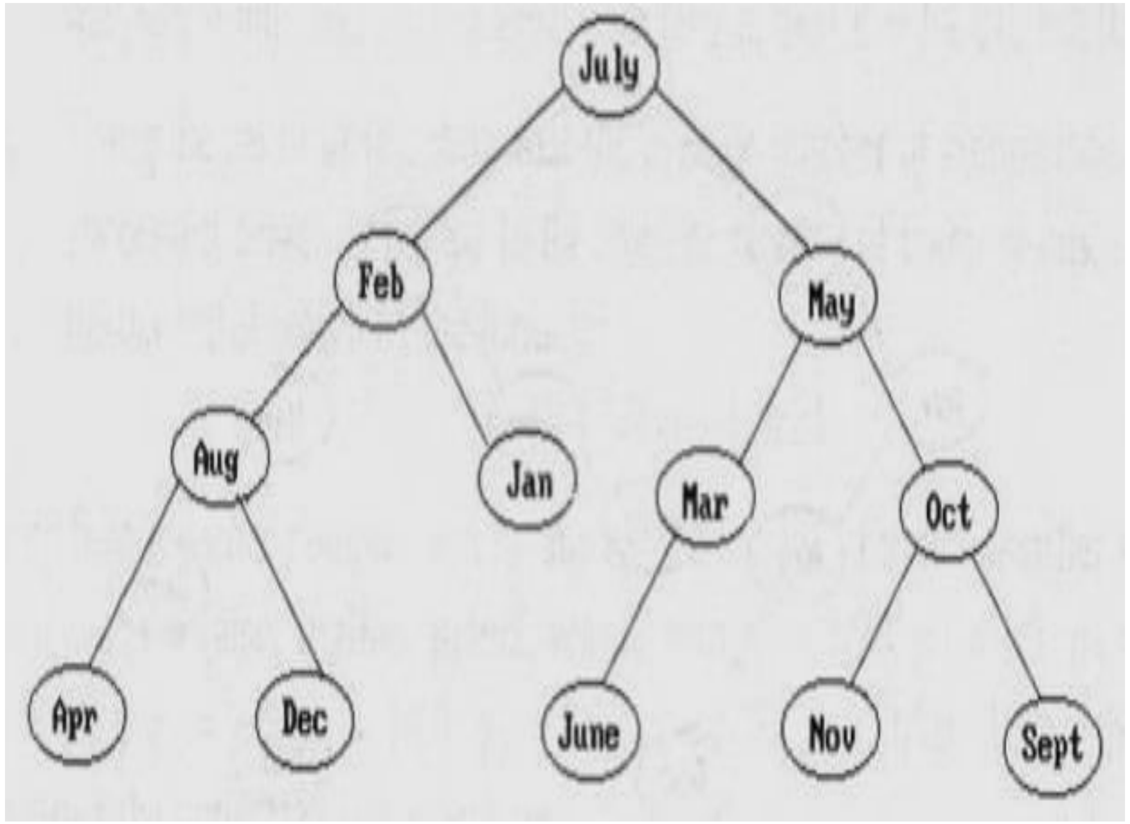
An AVL tree has balance factor calculated at every node

- For every node, heights of left and right subtree can differ by no more than 1
- Store current heights in each node

AVL Trees Definition:

- An empty binary tree is height balanced.
- If *T* is a nonempty binary tree with T_L and T_R as its left and right subtrees, then *T* is *height balanced iff*
  - T_L and T_R are height balanced, and
  - $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R, respectively.

The definition of a height balanced binary tree requires that every subtree also be height balanced



AVL Trees

The numbers by each node represent the difference in heights between the left and right subtrees of that node

We refer to this as the balance factor of the node

Definition:

- The balance factor, *BF(T)*, of a node, *T*, in a binary tree is defined as $h_L - h_R$, where h_L/h_R are the heights of the left/right subtrees of *T*.

- For any node *T* in an AVL tree *BF(T)* = -1, 0, or 1.

AVL Trees

We carried out the rebalancing using four different kinds of rotations: *LL*, *RR*, *LR*, and *RL*

- *LL* and *RR* are symmetric as are *LR* and *RL*
- These rotations are characterized by the nearest ancestor, *A*, of the inserted node, *Y*, whose balance factor becomes ± 2.
  - *LL*: *Y* is inserted in the left subtree of the left subtree of *A*.
  - *LR*: *Y* is inserted in the right subtree of the left subtree of *A*
  - *RR*: *Y* is inserted in the right subtree of the right subtree of *A*
  - *RL*: *Y* is inserted in the left subtree of the right subtree of *A*

Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :

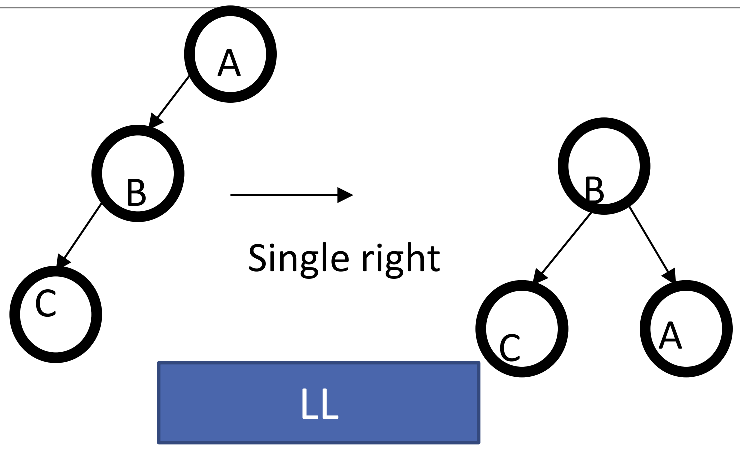
1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.

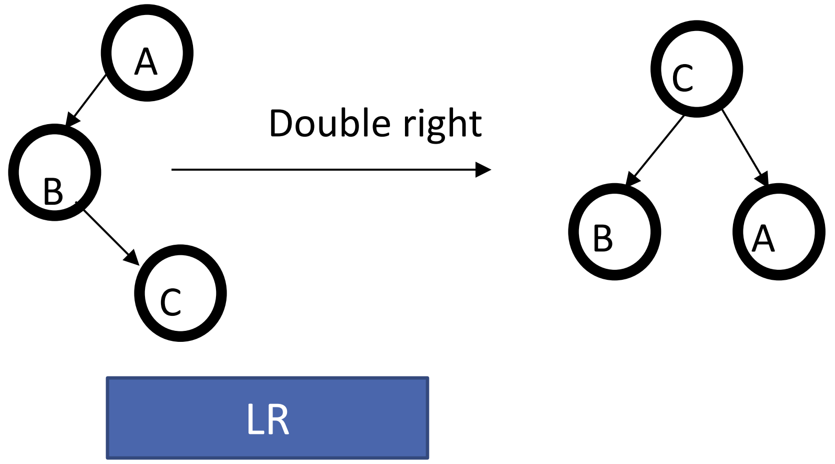
Inside Cases (require double rotation) :

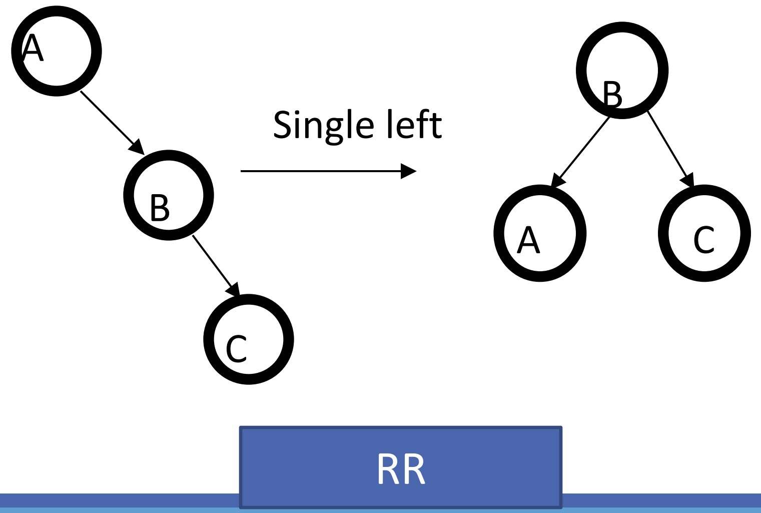
3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

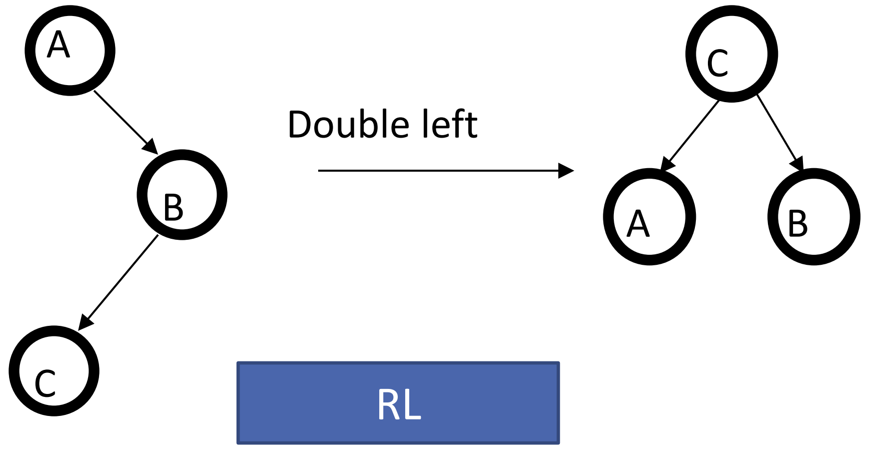
The rebalancing is performed through four separate rotation algorithms.

AVL Balancing : Four Rotations

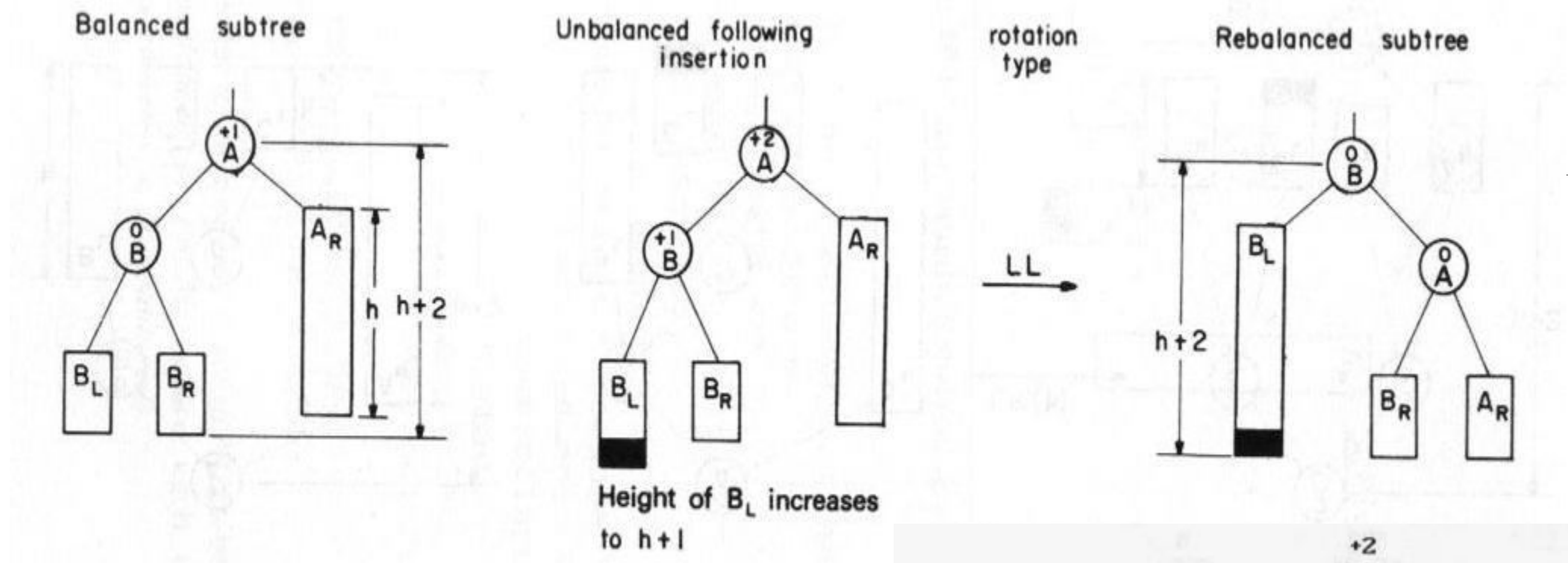


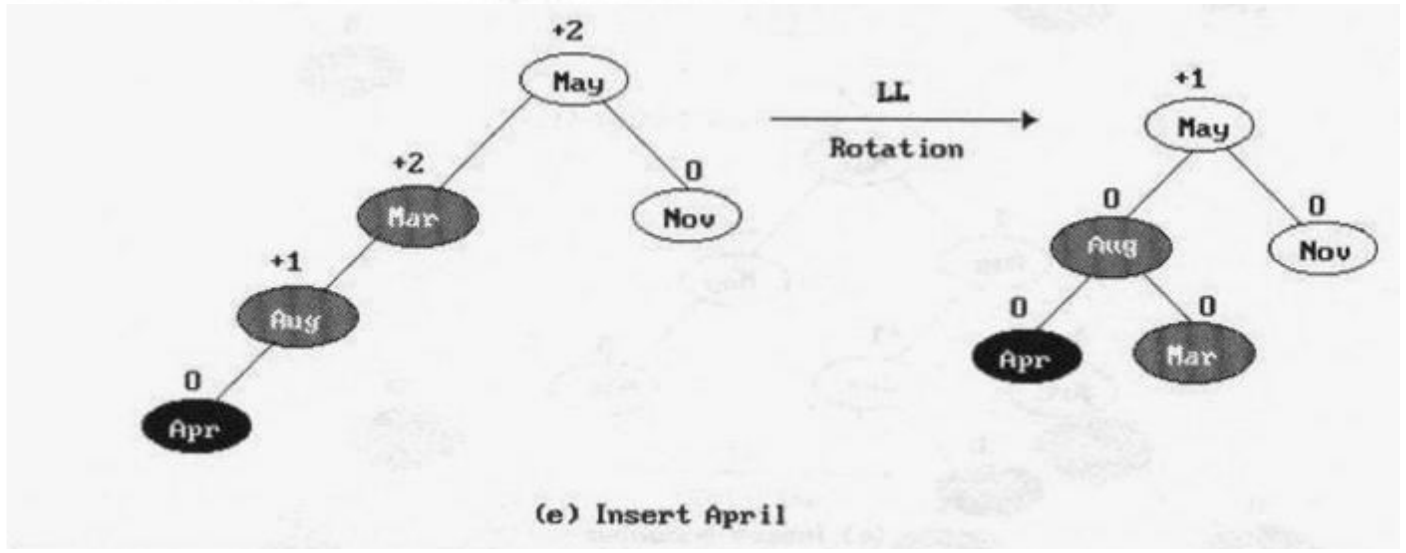






Rebalancing rotations

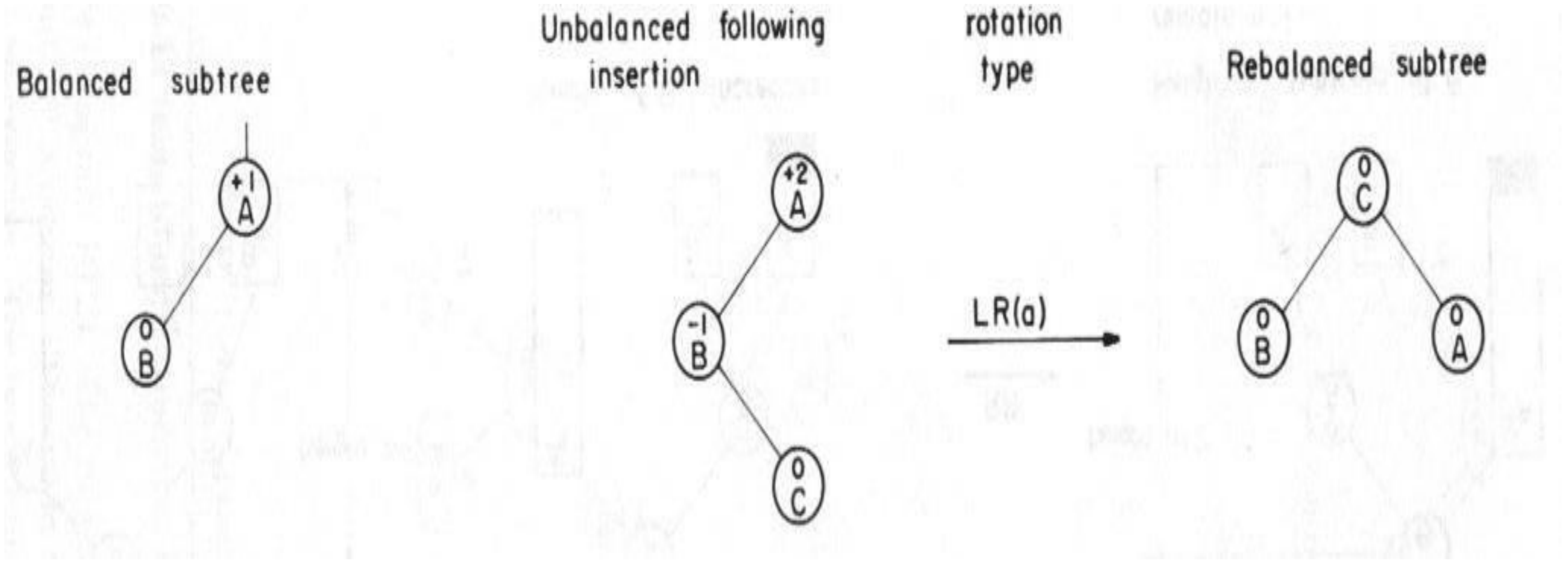




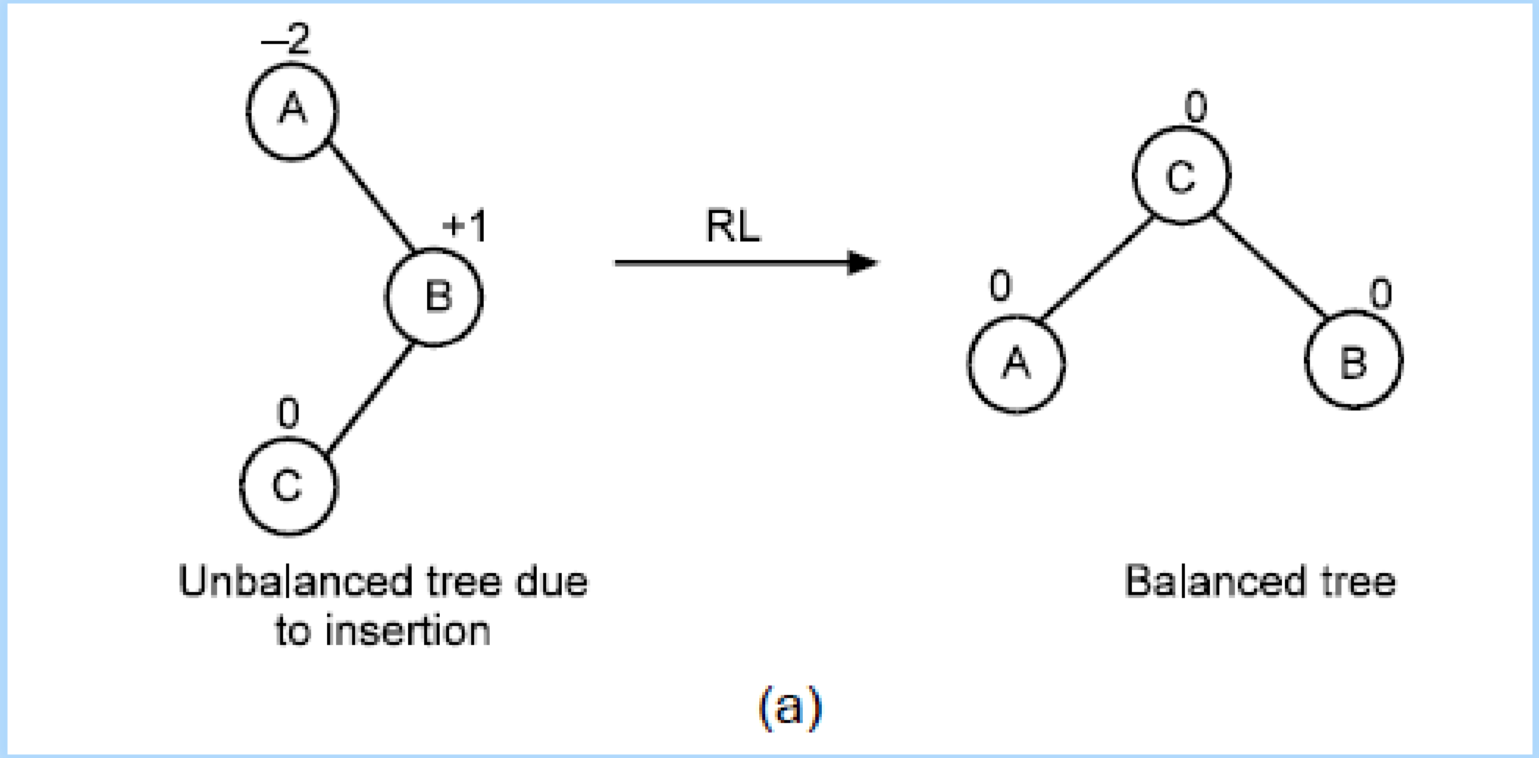
(e) Insert April

AVL Trees

Rebalancing rotations (cont'd)

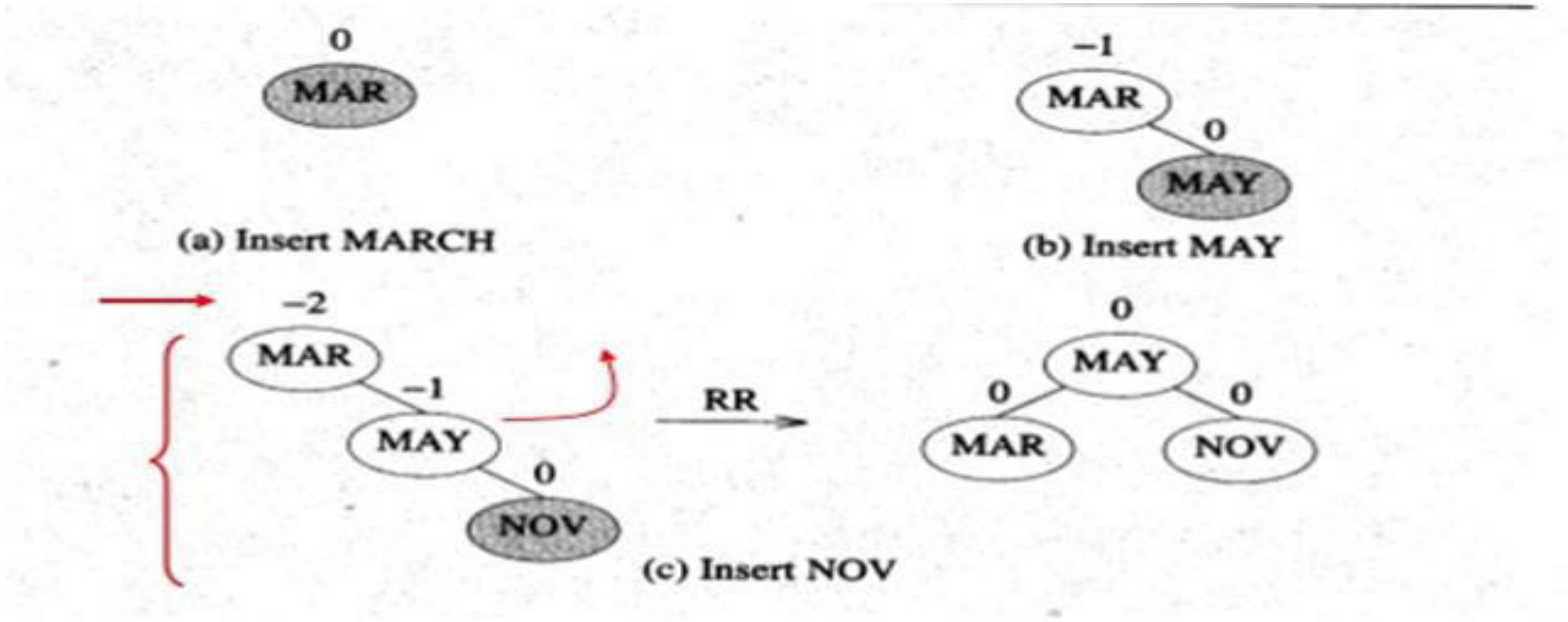


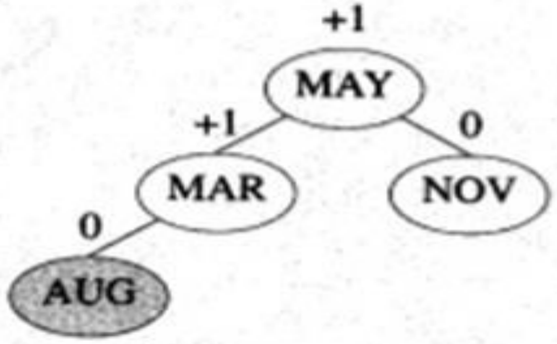
Case 4: RL (Left of Right)



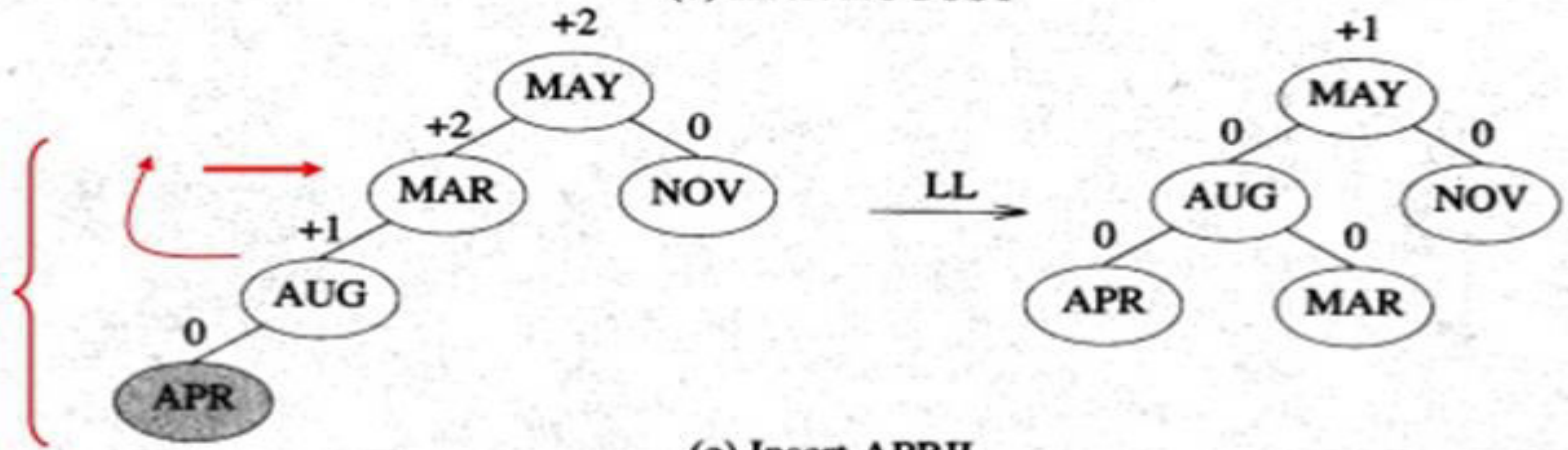
- This time we will insert the months into the tree in the order
    - *Mar, May, Nov, Aug, Apr, Jan, Dec, Jul, Feb, Jun, Oct, Sep*

- It shows the tree as it grows, and the restructuring involved in keeping it balanced.



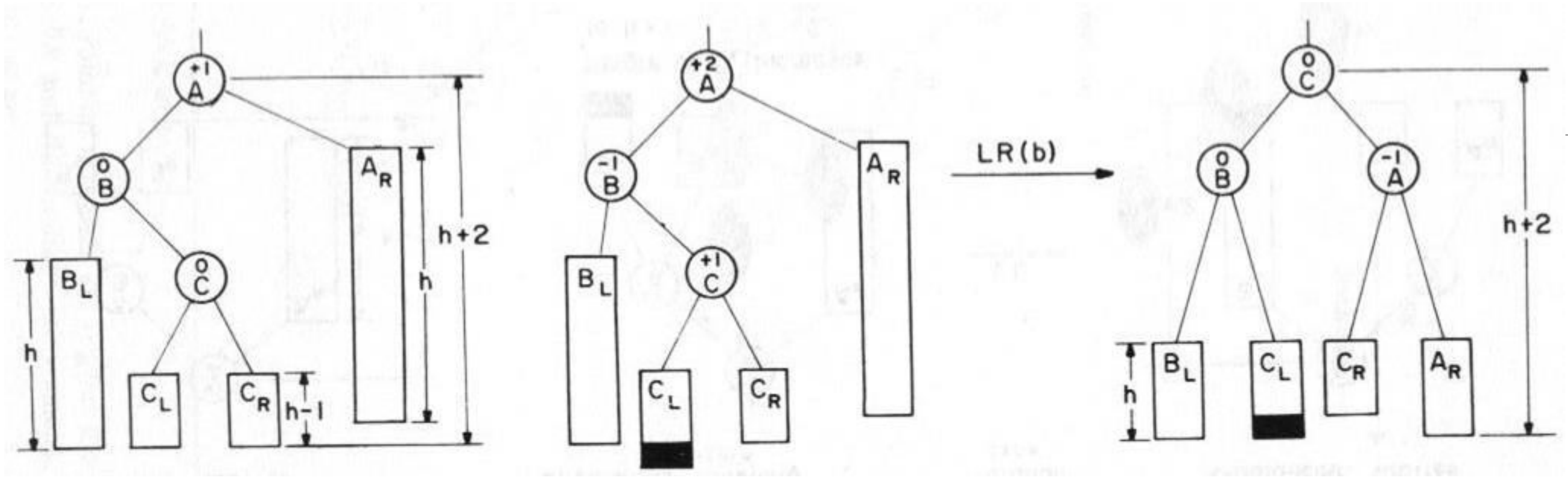


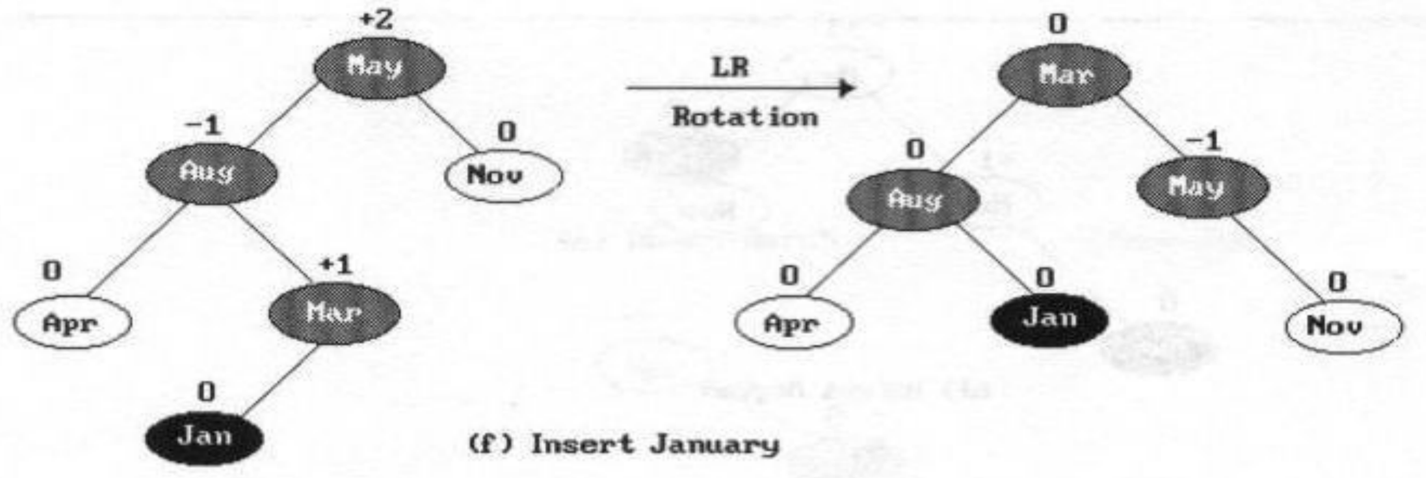
(d) Insert AUGUST



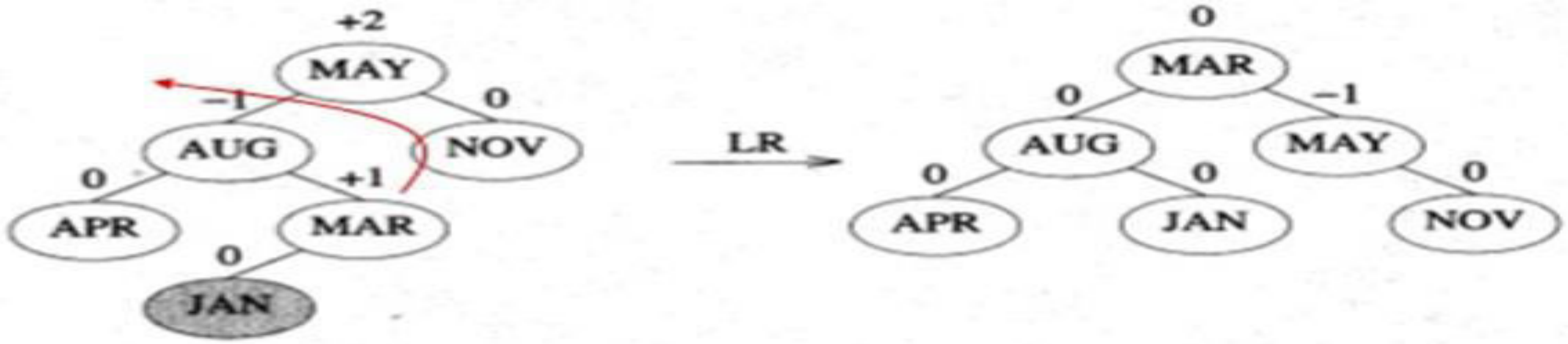
(e) Insert APRIL

Rebalancing rotations (cont'd)

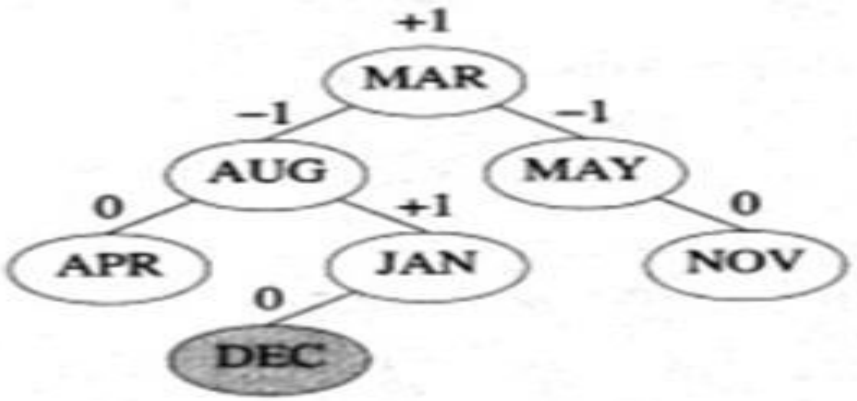




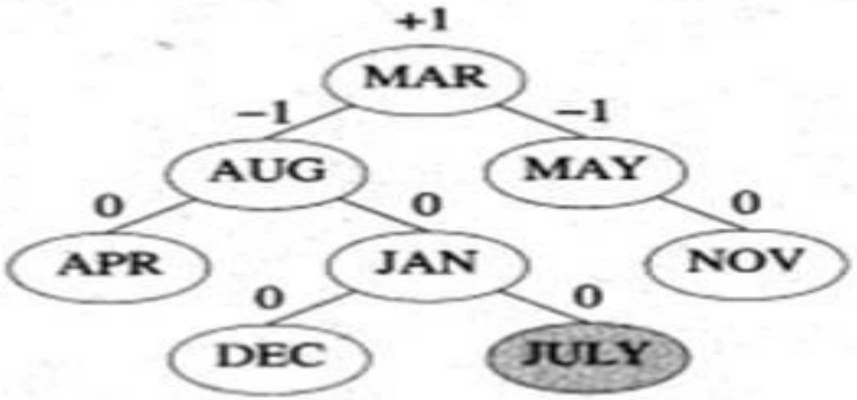
(f) Insert January



(f) Insert JANUARY

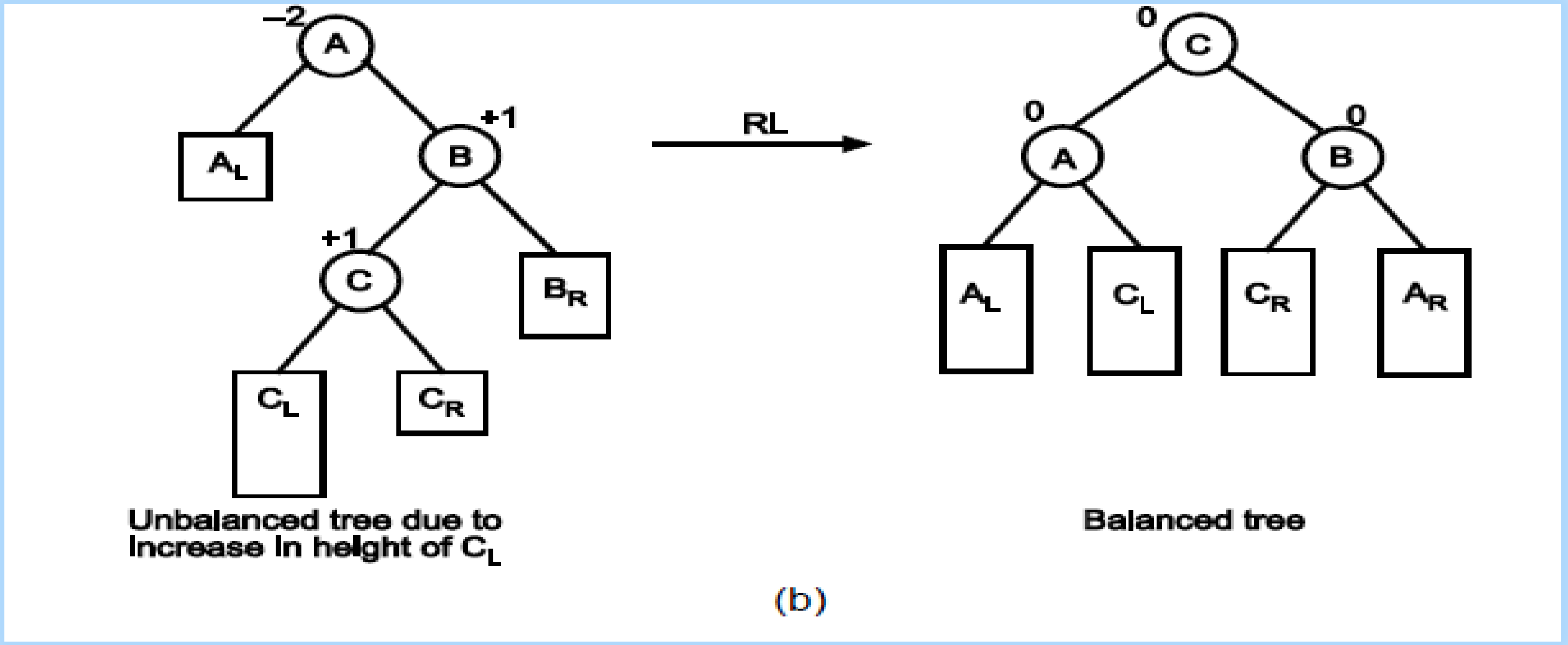


(g) Insert DECEMBER

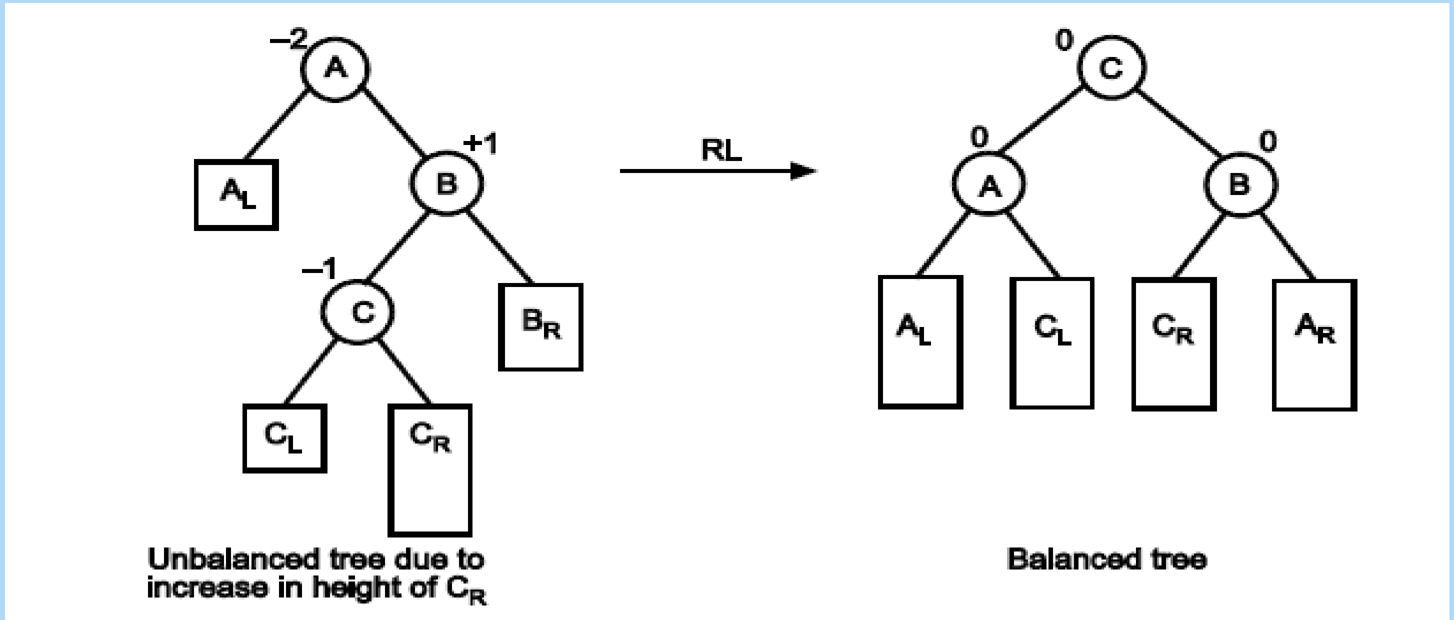


(h) Insert JULY

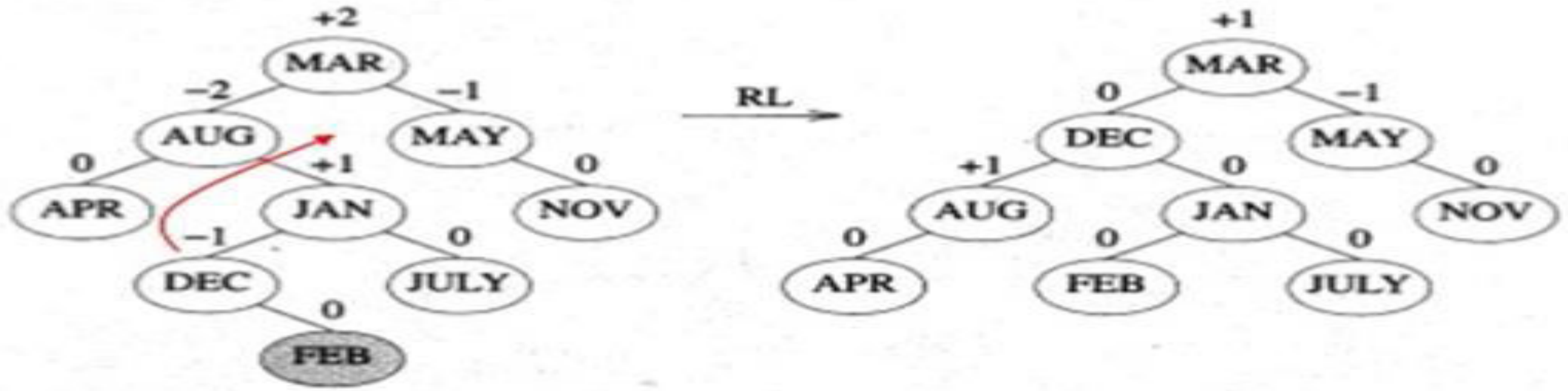
Case 4: RL(Left of Right)



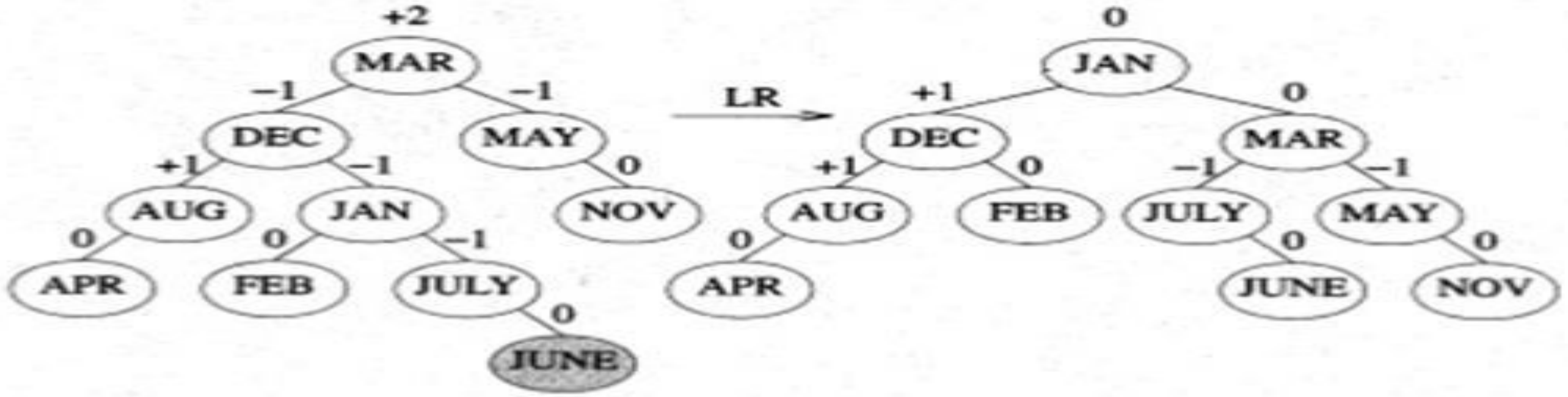
Case 4: RL (Left of Right)



(c)

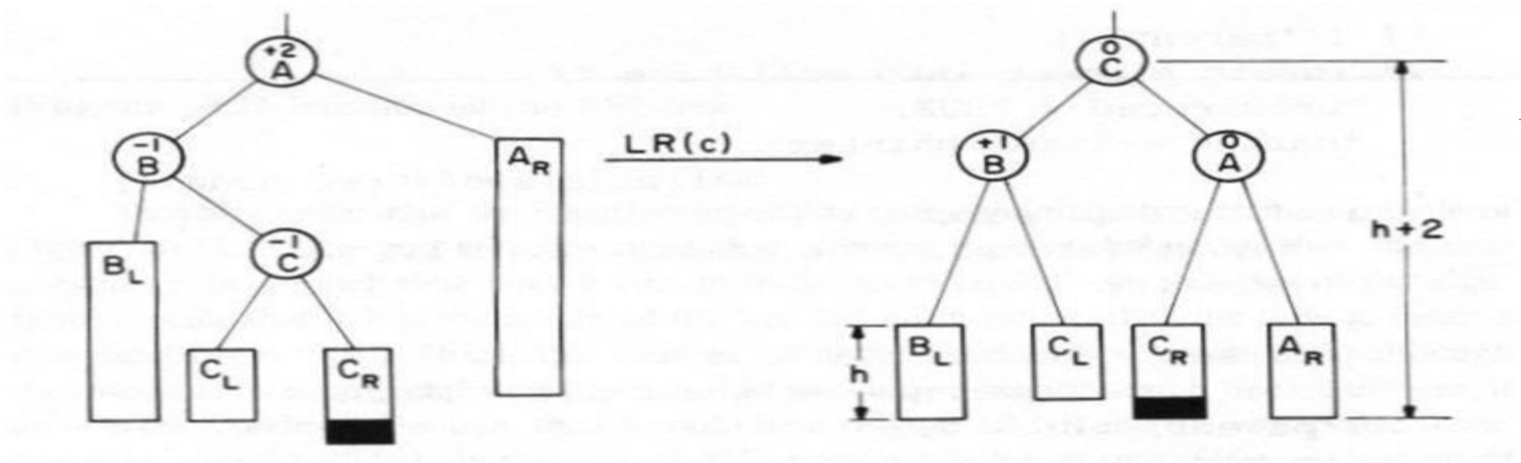


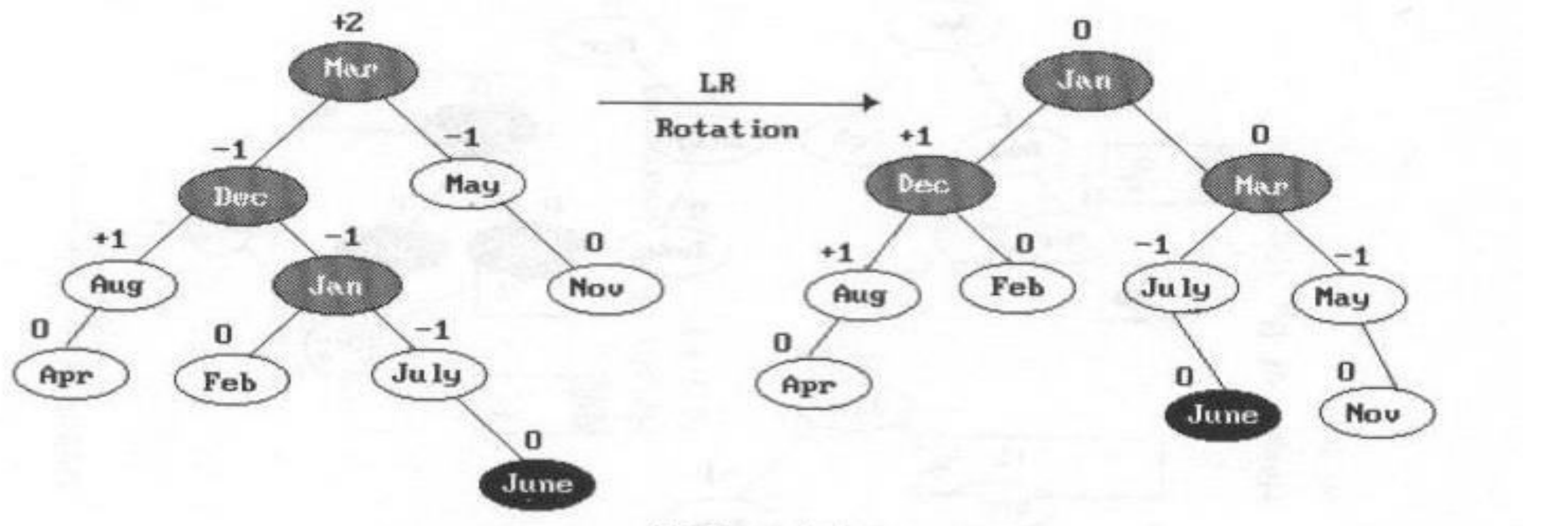
(i) Insert FEBRUARY



(j) Insert JUNE

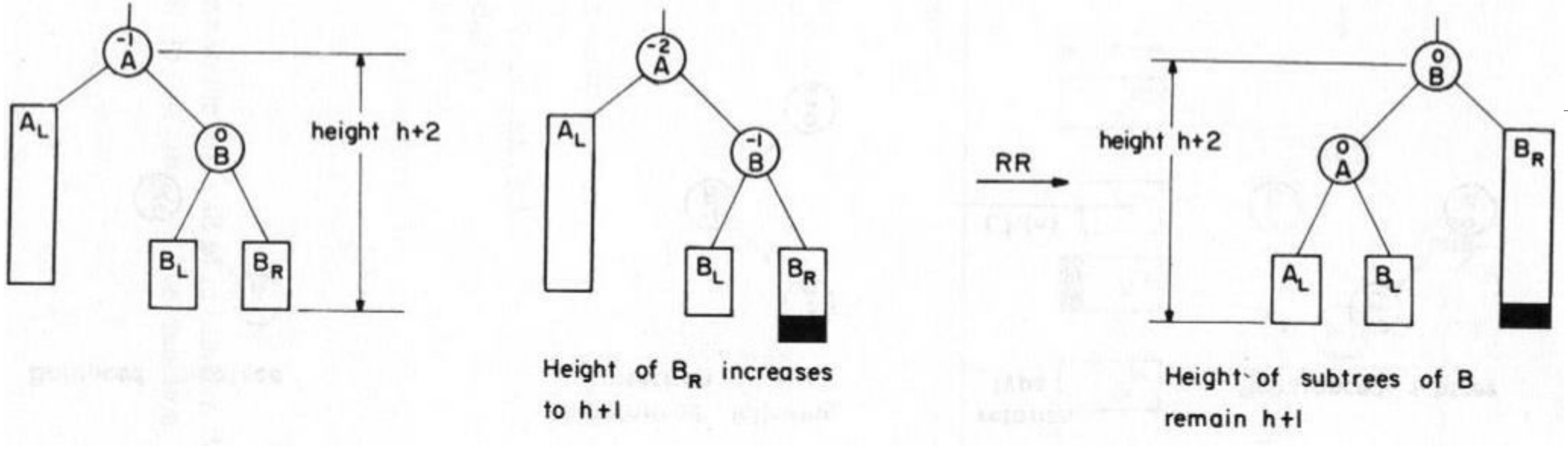
Case 3: LR (Left of Right)



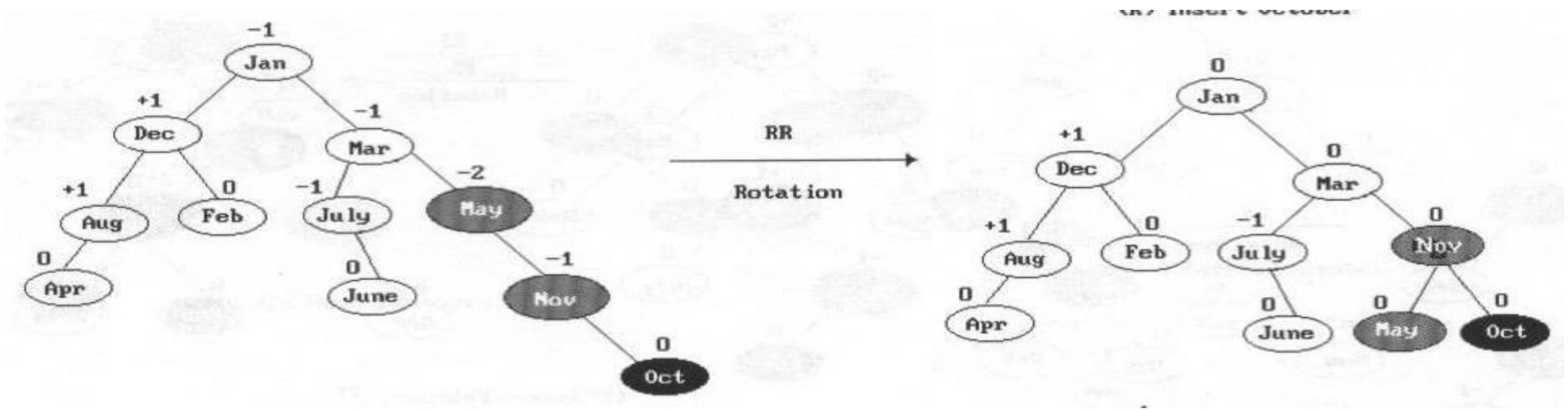


(j) Insert June

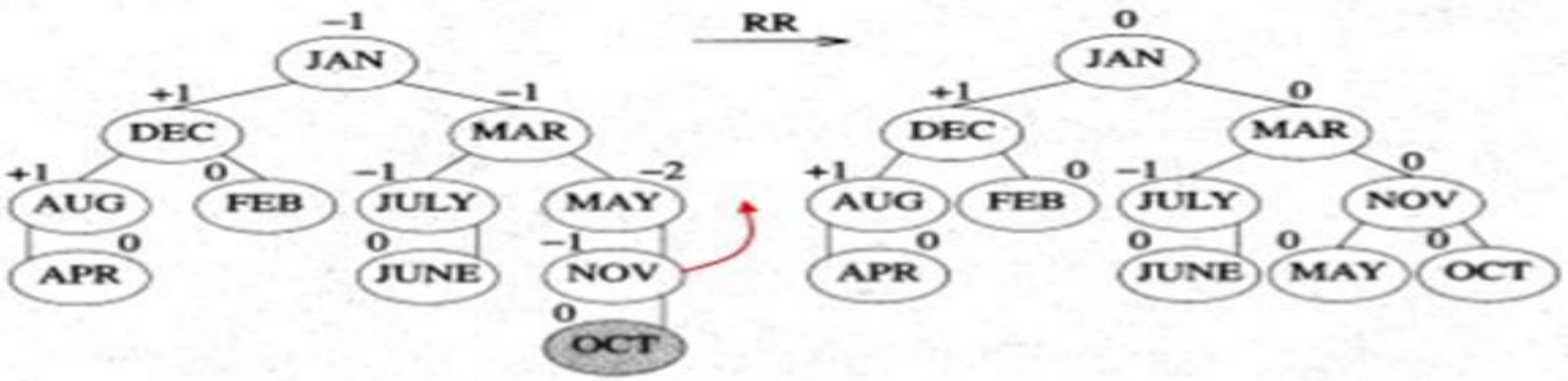
AVL Trees



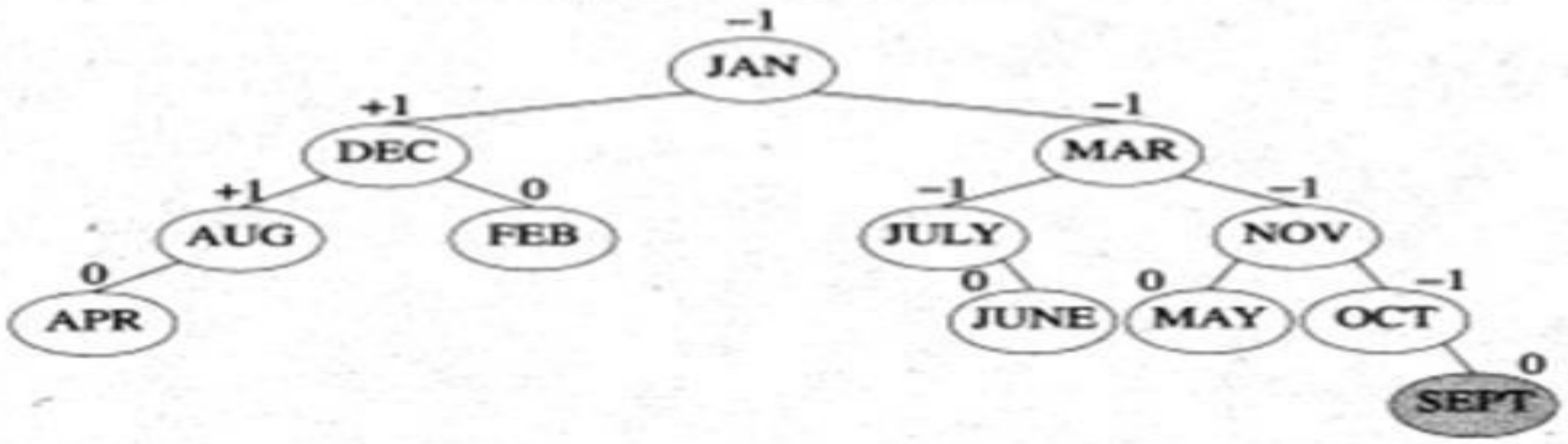
-1
A
A_L
0
B
B_L
B_R
height h+2
-2
A
A_L
-1
B
B_L
B_R
Height of B_R increases to h+1
RR
height h+2
0
B
0
A
A_L
B_L
B_R
Height of subtrees of B remain h+1



Jan
Dec
Mar
Aug
Feb
July
May
Apr
June
Nov
Oct
RR
Rotation
Jan
Dec
Mar
Aug
Feb
July
Nov
Apr
June
May
Oct



(k) Insert OCTOBER



(l) Insert SEPTEMBER

```
class node
{
  char v[10];
  int height;
  node *left,*right;
  public :
  friend class avl;
};
```

```
class avl
{  node *root;
  node* ll(node*);
  node* lr(node*);
  node* rl(node*);
  node* rr(node*);
  int height(node*);
  node*insert(node*,char[],node*);
  void inorder(node*);
};
```

```
height (node *root)          //Function for height
{
          int t1=-1,t2=-1;
          if(root!=NULL)
          {
           t1=height(root->left);
           t2=height(root->right);
          }
          if(t1<t2)
               h=t2;
          else
              h=t1;
     return h+1;
}
```

```
insert() //workhorse to insert
{
  do
  {
     Accept   key to be inserted :";
     root1=insert(root1,key,NULL);
      accept choice for next node
   }    while    (choice  is Y);
}
```

```
node *insert(root, key,p)     //Driver to insert
{
    if root=NULL
    {
                    allocate memory for root;
                    copy key to root->data;
                     root->height=0;
    }
 if(strcmp(root->data,key)>0)         //Left Subtree
   {
          root->left=insert(root->left,key,root);
          BF=height(root->left)-height(root->right);
          print  Balance factor:;
```

```
if(BF==2 ||BF==-2)
{
 if(strcmp(key , (root->left)->data)<0)
    {      print LL Rotation;
           if(p==NULL)
               root=LL(root);
           else
           {
              if(root==p->left)
                {  root=LL(root);  p->left=root;  }
              else
                  {     root=LL(root);  p->right=root;  }
            }   //else close
       }     //if strcmp close
```

```
  if(strcmp(key , (root->left)->data)>0)
   {
      print LR Rotation;
       if(p==NULL)
            root=LR(root);
       else
      {
          if(root==p->left)
               {  root=LR(root);p->left=root;  }
          else
               {   root=LR(root ); p->right=root;   }
       }  //else close
     }  //if strcmp close
   }     //if BF close
}    //main if close
```

```
else                                    //right subtree
   {
          if(strcmp(root->data , key)<0)
    }  //else close
return root;
}    //insert close
```

```
  node * LL(node *a)        //LL rotation
{
     node *b;
          b=a->left;
          a->left=b->right;
          b->right=a;
          a->ht=height(a);
          b->ht=height(b);
          return b;
}
```

```
node *avl::LR(node *a)                //LR rotation
{
          node *b,*c;
          b=a->left;
          c=b->right;
          a->left=c->right;
          b->right=c->left;
          c->left=b;
          c->right=a;
          a->ht=height(a);
          b->ht=height(b);
          c->ht=height(c);
          return c;
}
```

```
   node *avl::RR(node *a)//RR rotation
{
        node *b;
        b=a->right;
        a->right=b->left;
        b->left=a;
        a->ht=height(a);
        b->ht=height(b);
        return b;
}
```

```
node *avl::RL(node *a)     //RL rotation
{
        node *b,*c;
        b=a->right;
        c=b->left;
        a->right=c->left;
        b->left=c->right;
        c->left=a;
        c->right=b;
        a->ht=height(a);
        b->ht=height(b);
        c->ht=height(c);
        return c;
}
```

Assignment no 8

A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword

Binary Search Tree - Worst Time

Worst case running time is O(N)

- What happens when you Insert elements in ascending order?
  - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
- Problem: Lack of "balance":
  - compare depths of left and right subtree
- Unbalanced degenerate tree

B-Tree

A B-tree is a balanced multiway tree. A node of the tree contains many records or keys of records and pointers to children.

To reduce disk access, the following points are applicable:

1. Height is kept minimum.

2. All leaves are kept at the same level.

3. All nodes other than leaves must have at least minimum number of children.

B-tree Definition

A B-tree of order *m is a multiway tree with the following properties:*

1. The number of keys in each internal node is one less than the number of its non-empty children, and these keys partition the keys in the children in the fashion of the search tree.

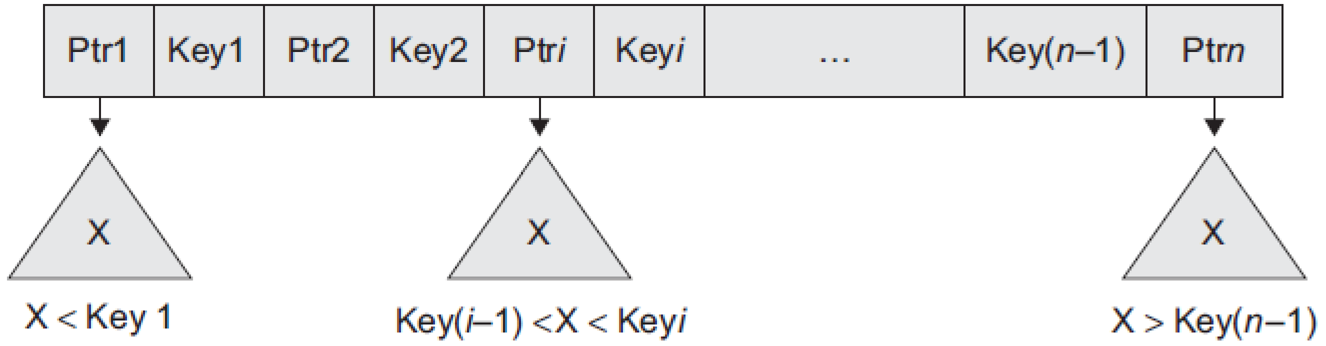
2. All leaves are on the same level.

3. All internal nodes except the root have utmost *m non-empty children and at least* $\lceil m/2 \rceil$ non-empty children.

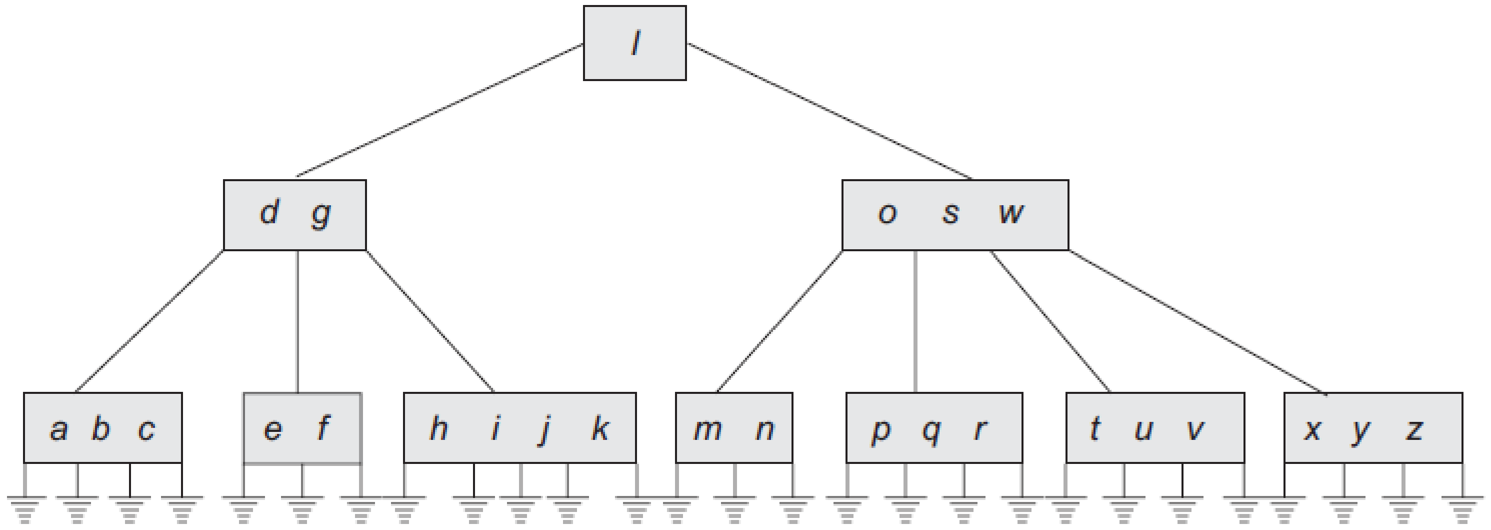
4. The root is either a leaf node, or it has from two to *m children.*

5. A leaf node contains no more than *m - 1 keys.*

Node structure for B-tree



B-tree of order 5



Reasons for using B-trees B-trees are widely used for the following reasons:

1. The cost of each disk transfer is high when the searching tables are held on disk and do not depend much on the amount of data transferred, especially if the consecutive items are transferred. Consider a condition of the B-tree of order 101. We can transfer each node in one disk read operation.

2. A B-tree of order 101 and height 3 can hold 1014 - 1 items (approximately 100 million), and any item can be accessed with three disk reads (assuming we hold the root in memory).

3. When a balanced tree is required and if we take *m = 3, we get a '2–3 tree', where the* non-leaf nodes have two or three children (i.e., one or two keys).

4. B-trees are always balanced (since the leaves are all at the same level) .

Inserting a key into a B-tree

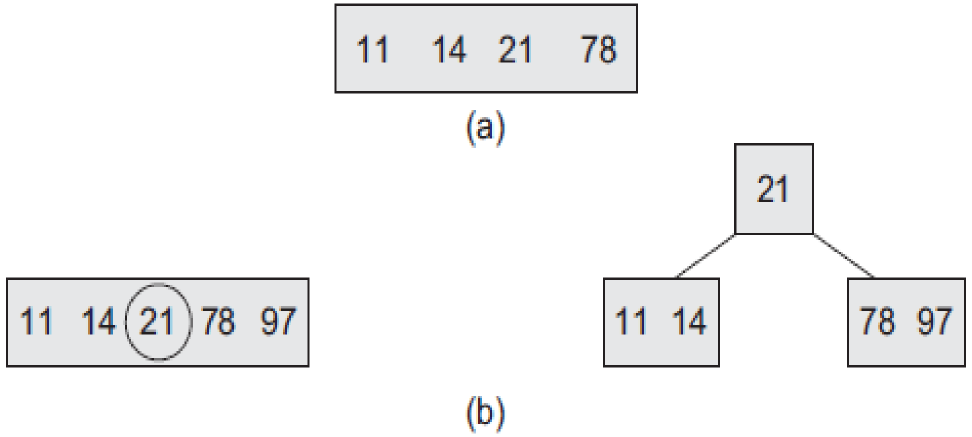
Binary search trees grow at their leaves, but the B-trees grow at the root.

The general method of insertion is as follows:

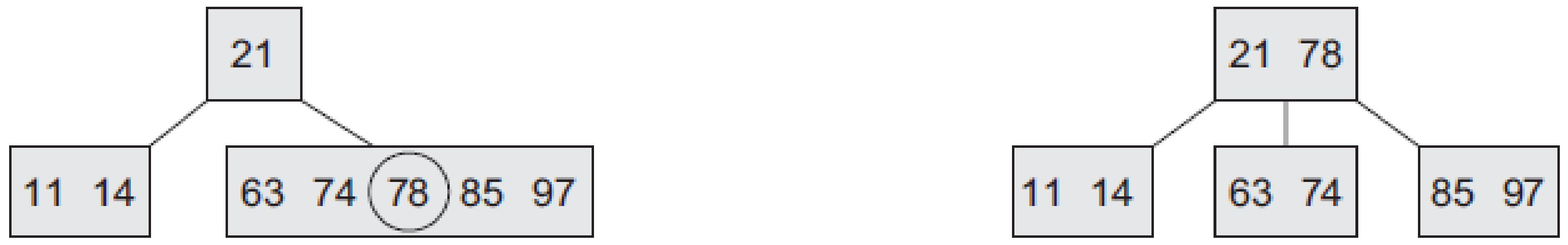
1. First, the new key is searched in the tree. If the new key is not found, then the search terminates at a leaf.
2. Attempt to insert the new key into a leaf.
3. If the leaf node is not full, then the new key is added to it and the insertion is finished.
4. If the leaf node is full, then it splits into two nodes on the same level, except that the median key is sent up the tree to be inserted into the parent node.
5. If this would result in the parent becoming too big, split the parent into two, promoting the middle key.
6. This strategy might have to be repeated all the way to the top.
7. If necessary, the root is split into two and the middle key is promoted to a new root, making the tree one level higher.

Let us see one example to build a B-tree of order 5 for the following data:

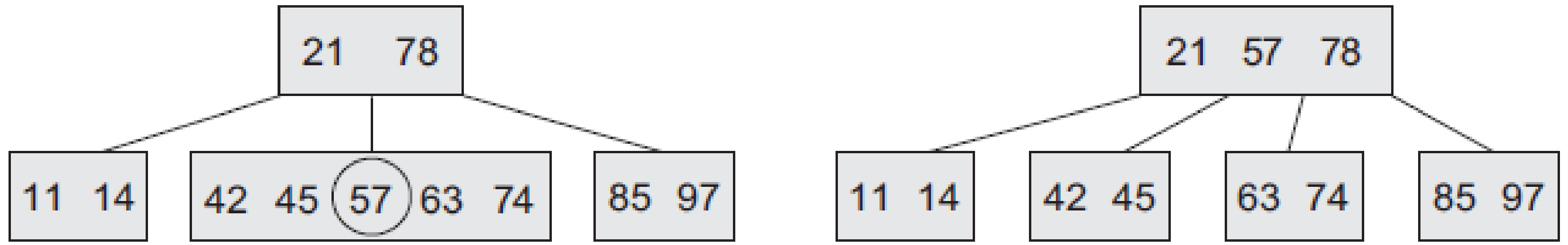
78, 21, 14, 11, 97, 85, 74, 63, 45, 42, 57, 20, 16, 19, 52, 30, 21.



78, 21, 14, 11, 97, 85, 74, 63, 45, 42, 57, 20, 16, 19, 52, 30, 21

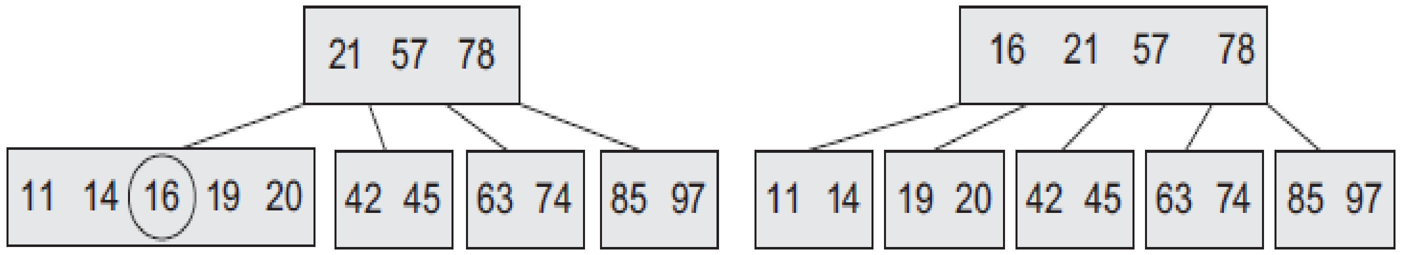


(c)

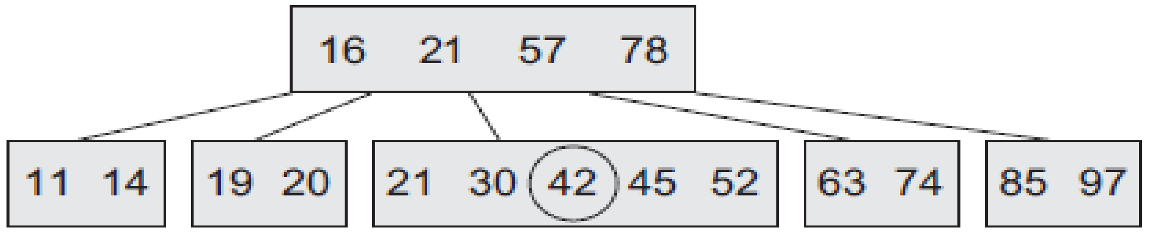


(d)

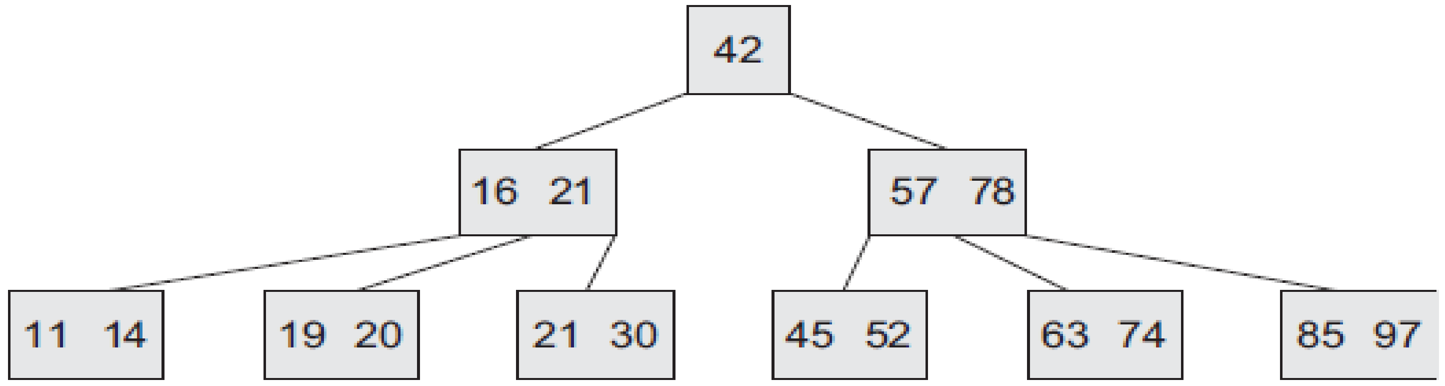
78, 21, 14, 11, 97, 85, 74, 63, 45, 42, 57, 20, 16, 19, 52, 30, 21



(e)



(f)



(g)

Deleting from a B-tree

During insertion, the key always goes *into a leaf*.

For deletion, if we wish to remove *from a leaf, there are three possible ways mentioned as follows:*

1. If the key is already in a leaf node and removing it does not cause that leaf node to have too few keys, then simply remove the key to be deleted.

2. If the key is *not in a leaf, then it is guaranteed (by the nature of a B-tree) that its* predecessor or successor will be in a leaf—in this case, we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

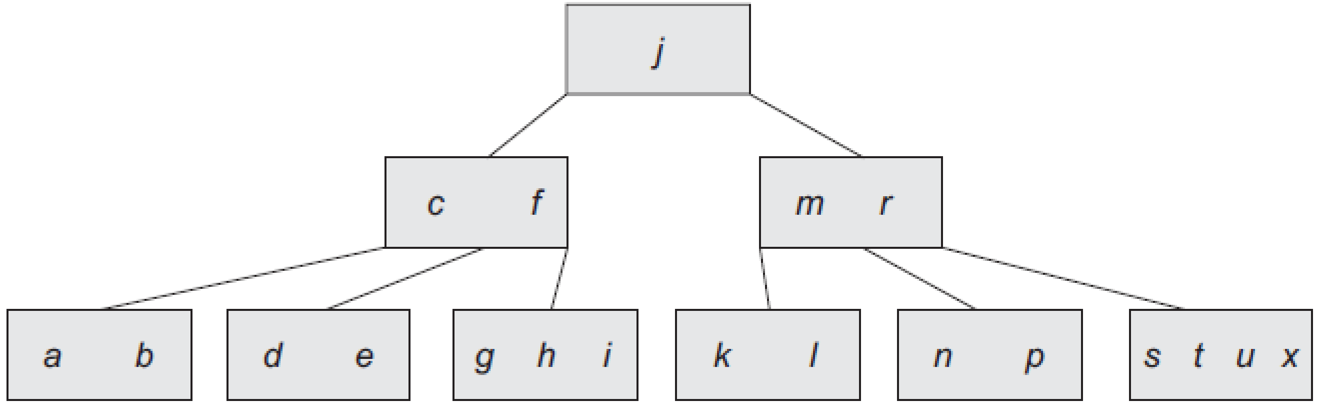
3. If these two conditions lead to a leaf node containing less than the minimum number of keys, then we have to look at the siblings immediately adjacent to the leaf in questions listed as follows:

(a) If one of them has more than the minimum number of keys, then we can promote one of its keys to the parent and take the parent key into our lacking leaf.

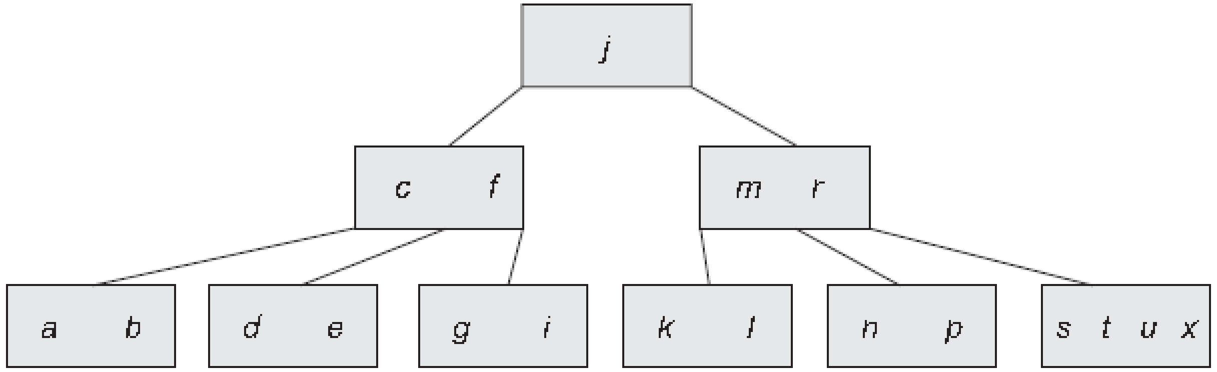
(b) If neither of them has more than the minimum number of keys, then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key), and the new leaf will have the correct number of keys; if this step leaves the parent with very few keys, then we repeat the process up to the root itself, if required.

4. If the leaf contains more than the minimum number of entries, then the data can be deleted with no further action.

Delete h ,r ,p & d

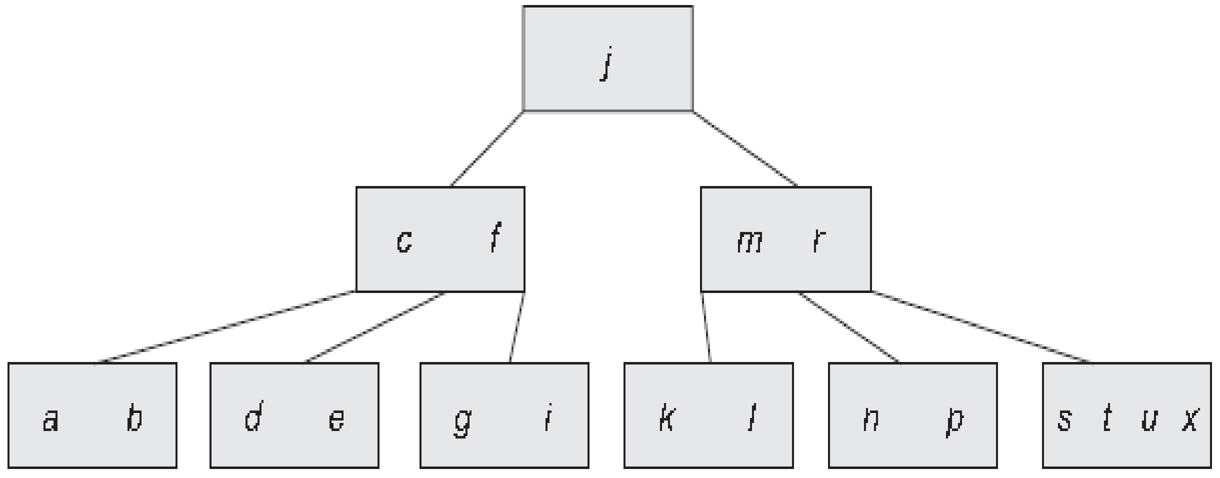


(a)

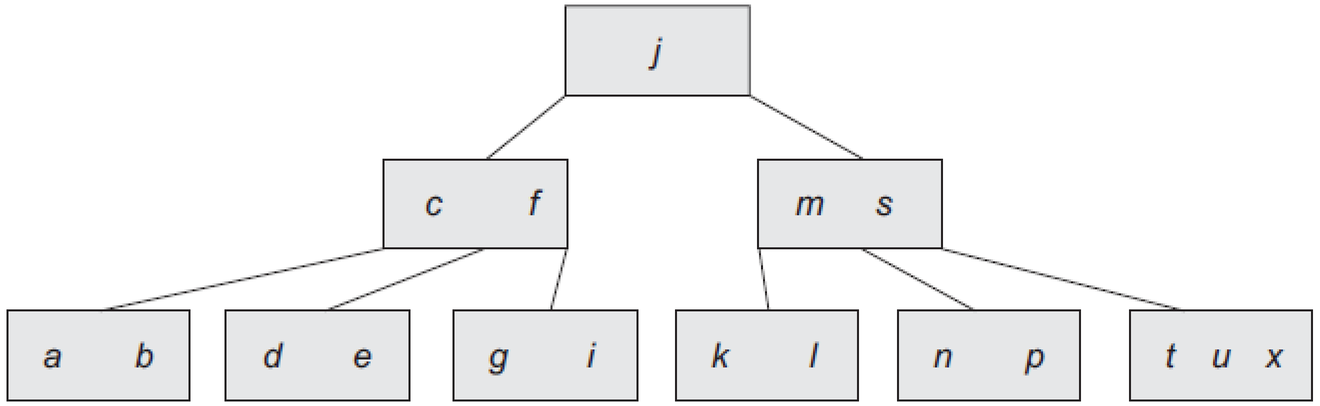


(b)

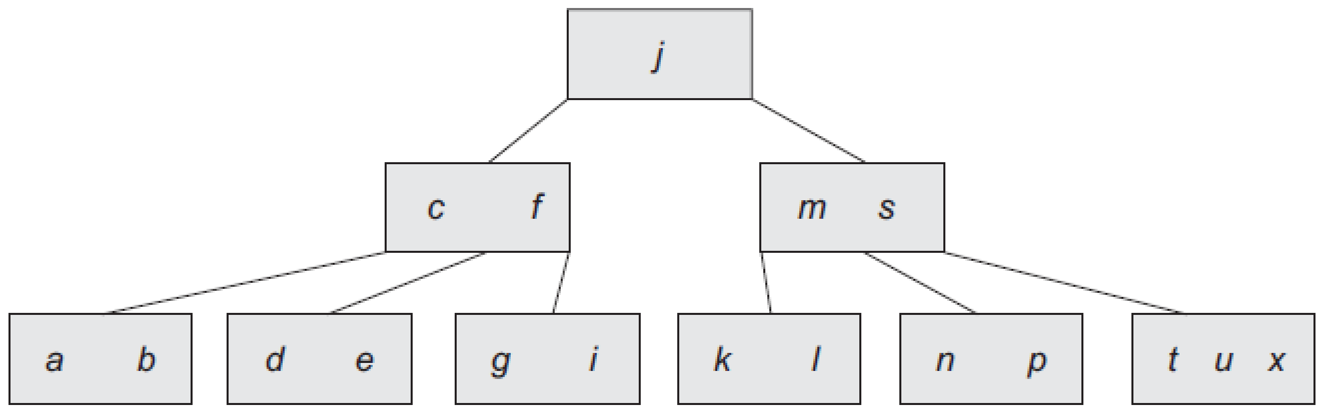
delete r

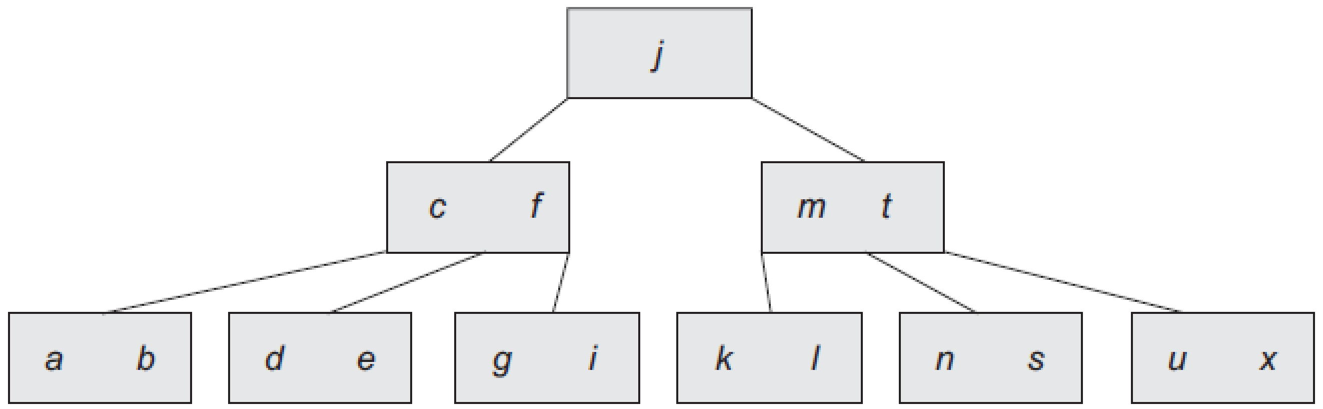


(b)

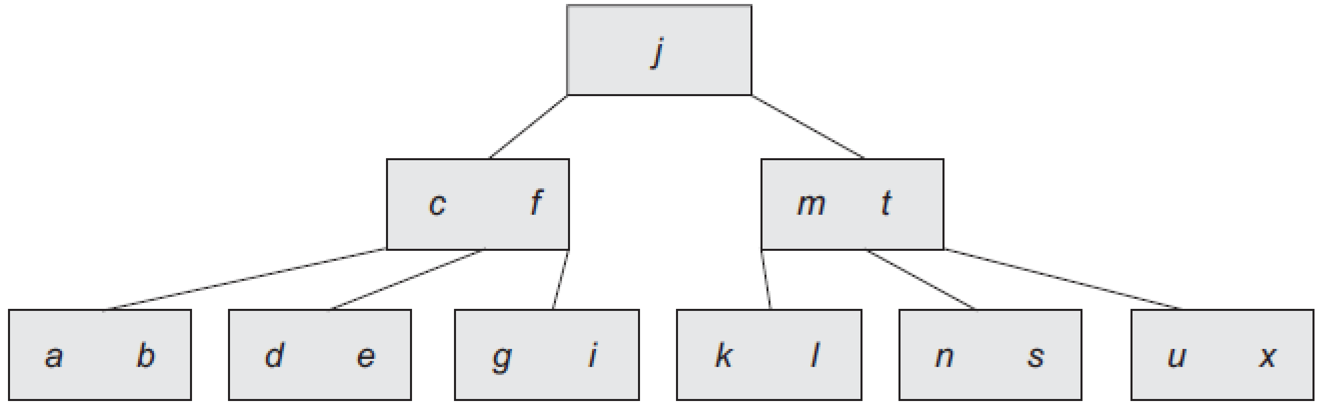


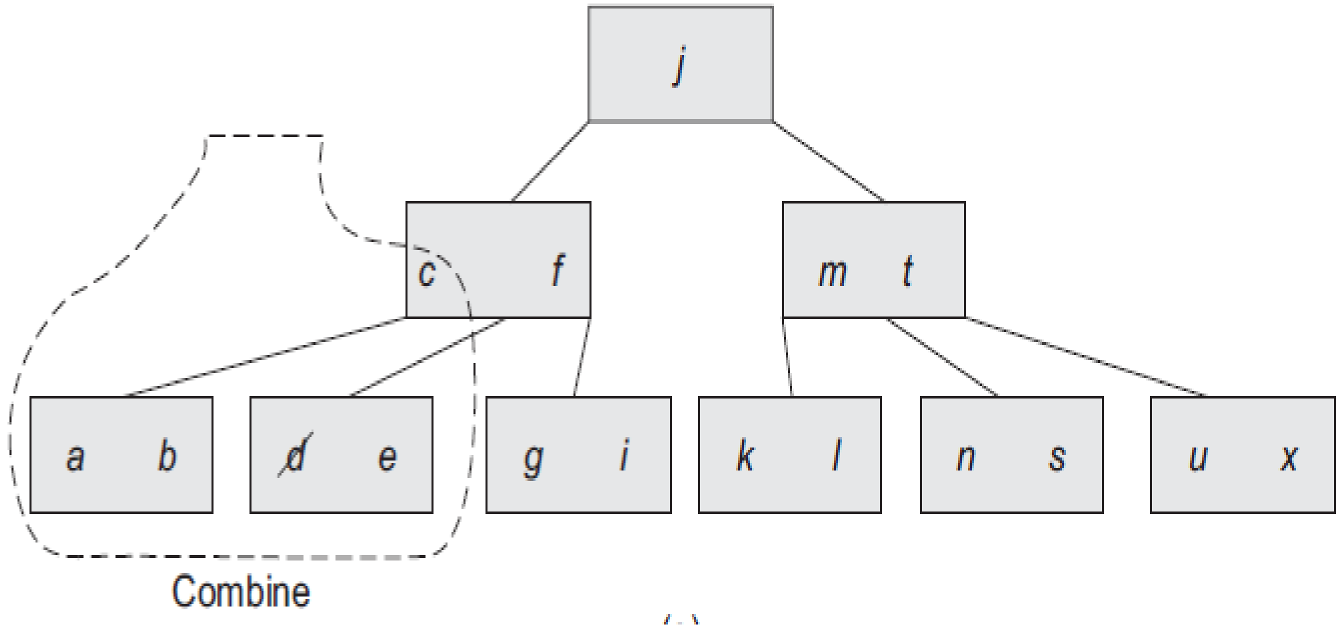
Delete p

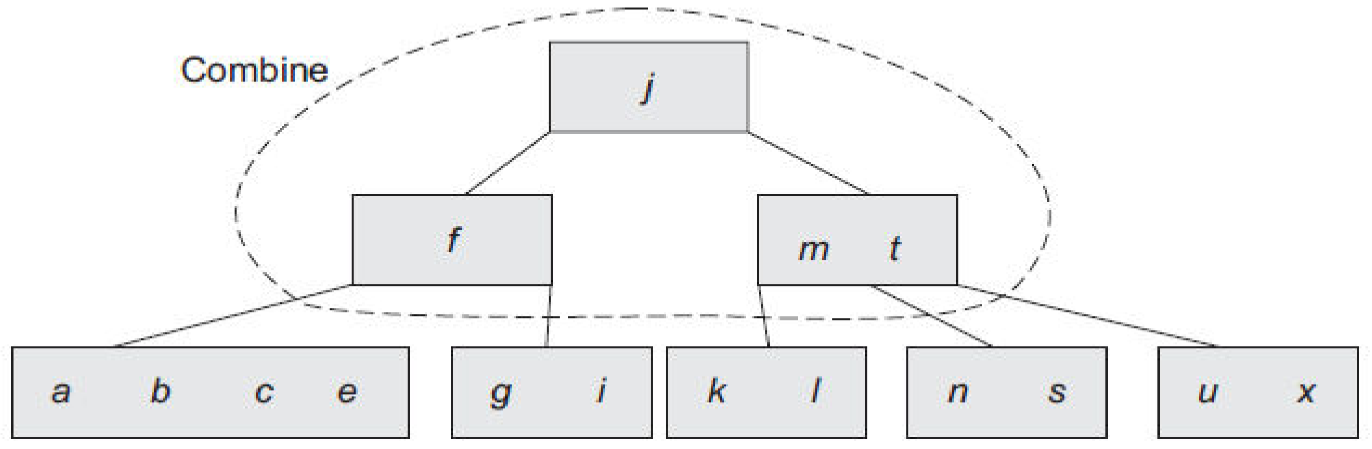




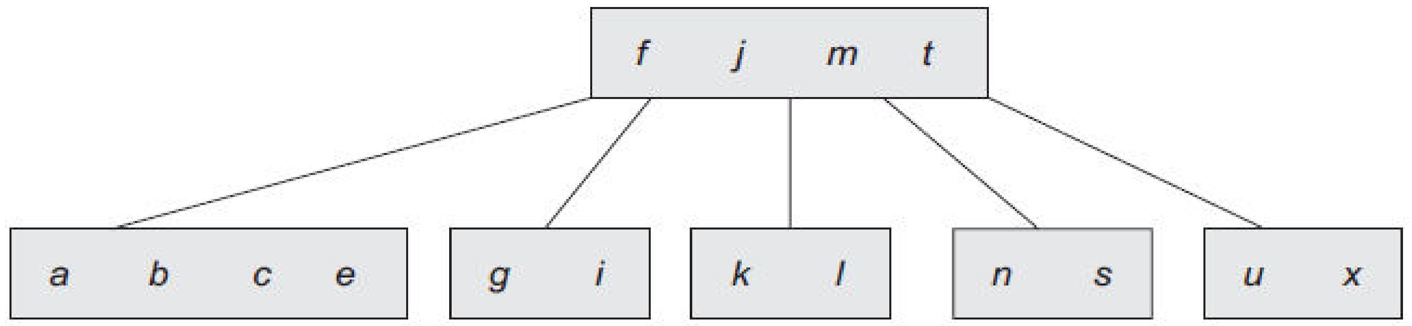
Delete d







(b)



(c)