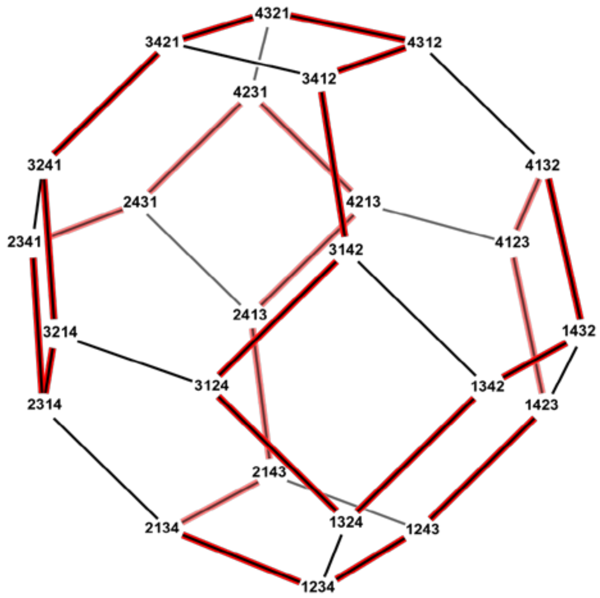
CS3334 Data Structures

Lecture 10: Balanced Binary Search Trees (AVL Trees)



Chee Wei Tan

Story Thus Far

Draw the worst-case shape of a Binary Search Tree for a search query

Story Thus Far

Draw the worst-case shape of a Binary Search Tree for a search query

How many Binary Search Trees are possible with *n* nodes?

Story Thus Far

Draw the worst-case shape of a Binary Search Tree for a search query

How many Binary Search Trees are possible with *n* nodes?

What are ideal Binary Search Trees that have O(log(*n*)) complexity for a search query?

Story Thus Far

Draw the worst-case shape of a Binary Search Tree for a search query

How many Binary Search Trees are possible with *n* nodes?

What are ideal Binary Search Trees that have O(log(*n*)) complexity for a search query?

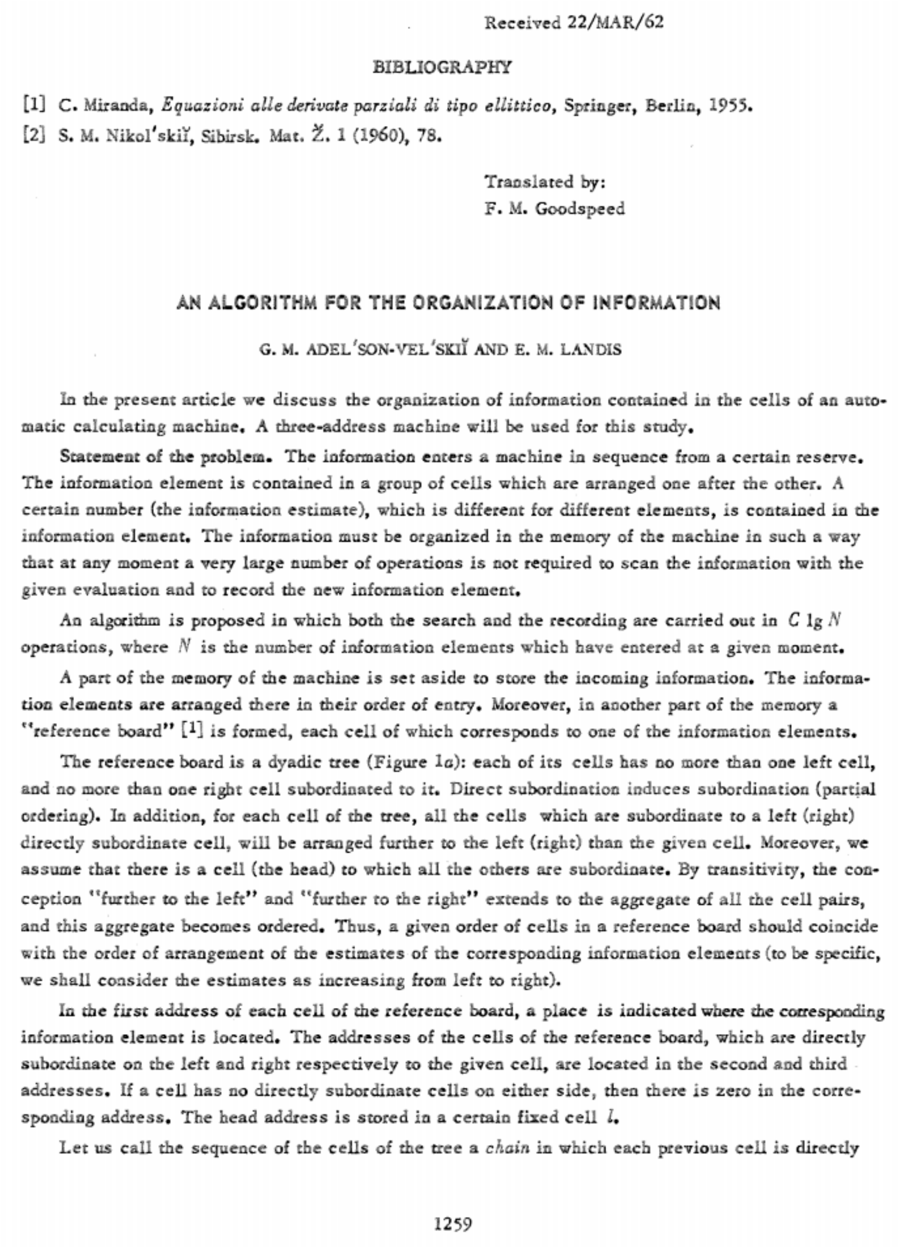
Goldren rule: For any node, the heights of its two subtrees differ by no more than one. (Balanced condition)
The level of the root of the binary search tree thus differs from its height by at most one.

AVL Trees in Computer Science

Georgy **A**delson-**V**elsky

Evgenii **L**andis

1962 paper "An algorithm for the organization of information" originally in Russian



Received 22/MAR/62

BIBLIOGRAPHY

[1] C. Miranda, *Equazioni alle derivate parziali di tipo ellittico*, Springer, Berlin, 1955.

[2] S. M. Nikol'skiĭ, Sibirsk. Mat. Ž. 1 (1960), 78.

Translated by:
F. M. Goodspeed

AN ALGORITHM FOR THE ORGANIZATION OF INFORMATION

G. M. ADEL'SON-VEL'SKIĬ AND E. M. LANDIS

In the present article we discuss the organization of information contained in the cells of an automatic calculating machine. A three-address machine will be used for this study.

Statement of the problem. The information enters a machine in sequence from a certain reserve. The information element is contained in a group of cells which are arranged one after the other. A certain number (the information estimate), which is different for different elements, is contained in the information element. The information must be organized in the memory of the machine in such a way that at any moment a very large number of operations is not required to scan the information with the given evaluation and to record the new information element.

An algorithm is proposed in which both the search and the recording are carried out in $C \lg N$ operations, where N is the number of information elements which have entered at a given moment.

A part of the memory of the machine is set aside to store the incoming information. The information elements are arranged there in their order of entry. Moreover, in another part of the memory a "reference board" [1] is formed, each cell of which corresponds to one of the information elements.

The reference board is a dyadic tree (Figure 1*a*): each of its cells has no more than one left cell, and no more than one right cell subordinated to it. Direct subordination induces subordination (partial ordering). In addition, for each cell of the tree, all the cells which are subordinate to a left (right) directly subordinate cell, will be arranged further to the left (right) than the given cell. Moreover, we assume that there is a cell (the head) to which all the others are subordinate. By transitivity, the conception "further to the left" and "further to the right" extends to the aggregate of all the cell pairs, and this aggregate becomes ordered. Thus, a given order of cells in a reference board should coincide with the order of arrangement of the estimates of the corresponding information elements (to be specific, we shall consider the estimates as increasing from left to right).

In the first address of each cell of the reference board, a place is indicated where the corresponding information element is located. The addresses of the cells of the reference board, which are directly subordinate on the left and right respectively to the given cell, are located in the second and third addresses. If a cell has no directly subordinate cells on either side, then there is zero in the corresponding address. The head address is stored in a certain fixed cell l.

Let us call the sequence of the cells of the tree a *chain* in which each previous cell is directly

1259

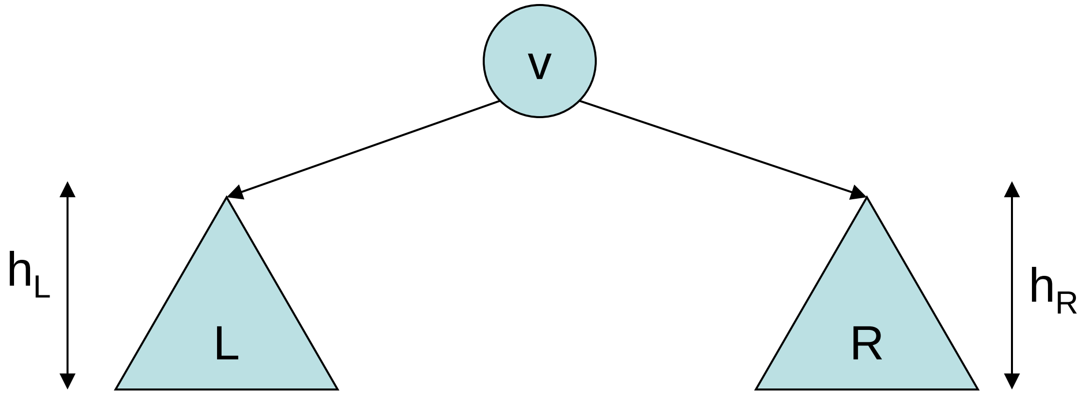
AVL Trees (1/2)

- One of oldest & most well-known balanced binary search tree (BST)
- *Balancing Condition: For each node v, the difference between the height of its left subtree and the height of its right subtree* ≤ 1
- Having this balancing condition will keep the tree height to be O(logn). This implies fast operation time
- How to maintain the balancing condition after insertions & deletions? By *rotations*!

AVL Trees (2/2)

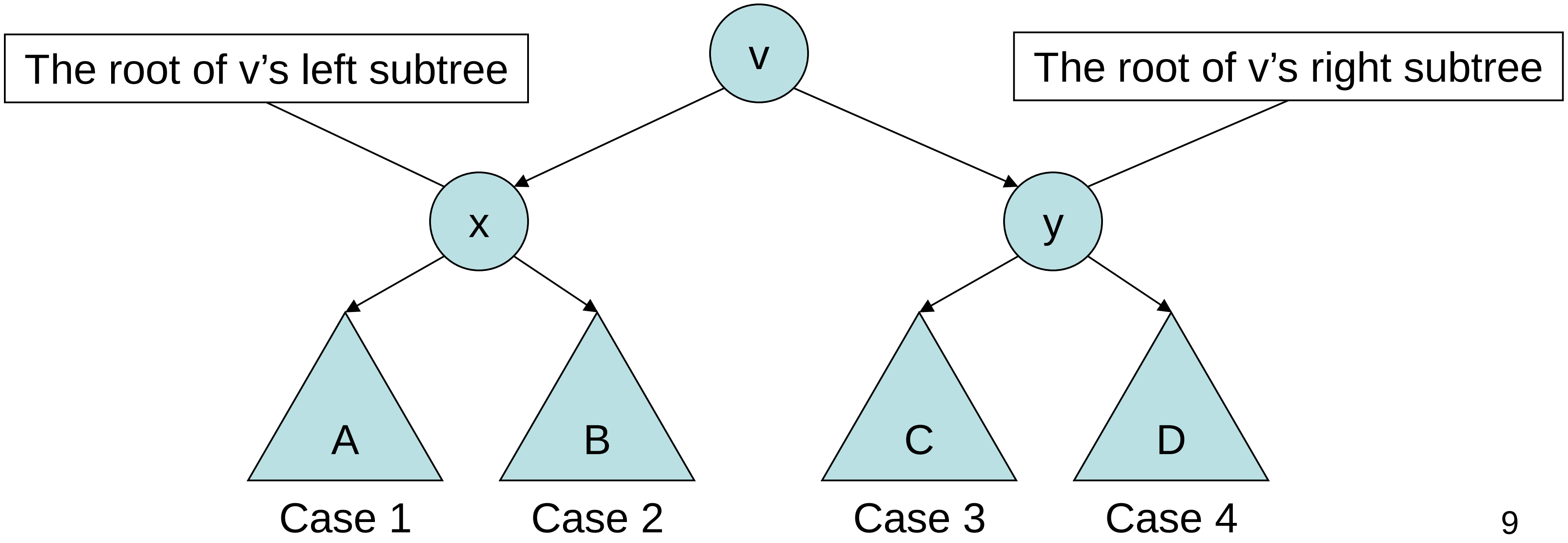
- Height of a tree = number of nodes on a longest root-to-leaf path
- Note: height of the tree below is

  $= \max(h_L, h_R) + 1$



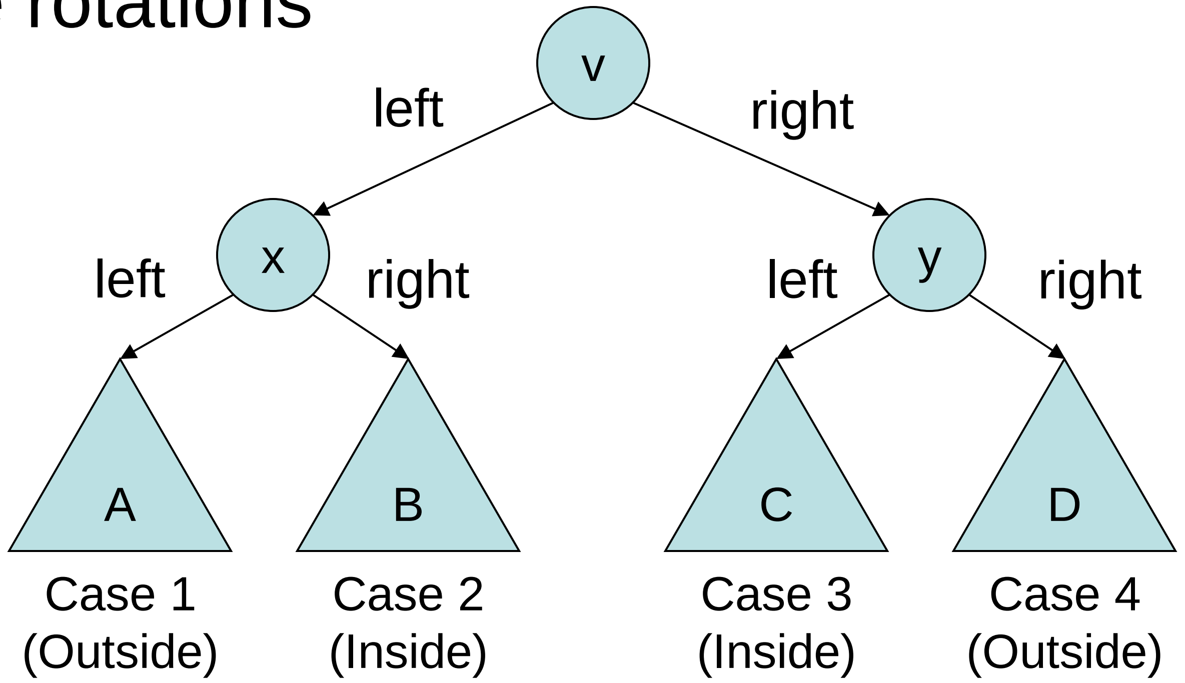
Insertion (1/2)

- Consider insert(u): only nodes along the path from root to the point of insertion may be unbalanced
- Suppose node v unbalanced, 4 cases:
  - Cases 1 & 4 are mirror image symmetries with respect to v
  - Cases 2 & 3 are mirror image symmetries with respect to v



Insertion (2/2)

- In Case 1 (or Case 4), the insertion occurs on the “outside” (i.e., left-left or right-right); it is fixed by a single rotation
- In Case 2 (or Case 3), the insertion occurs on the “inside” (i.e., left-right or right-left); it is handled by double rotations

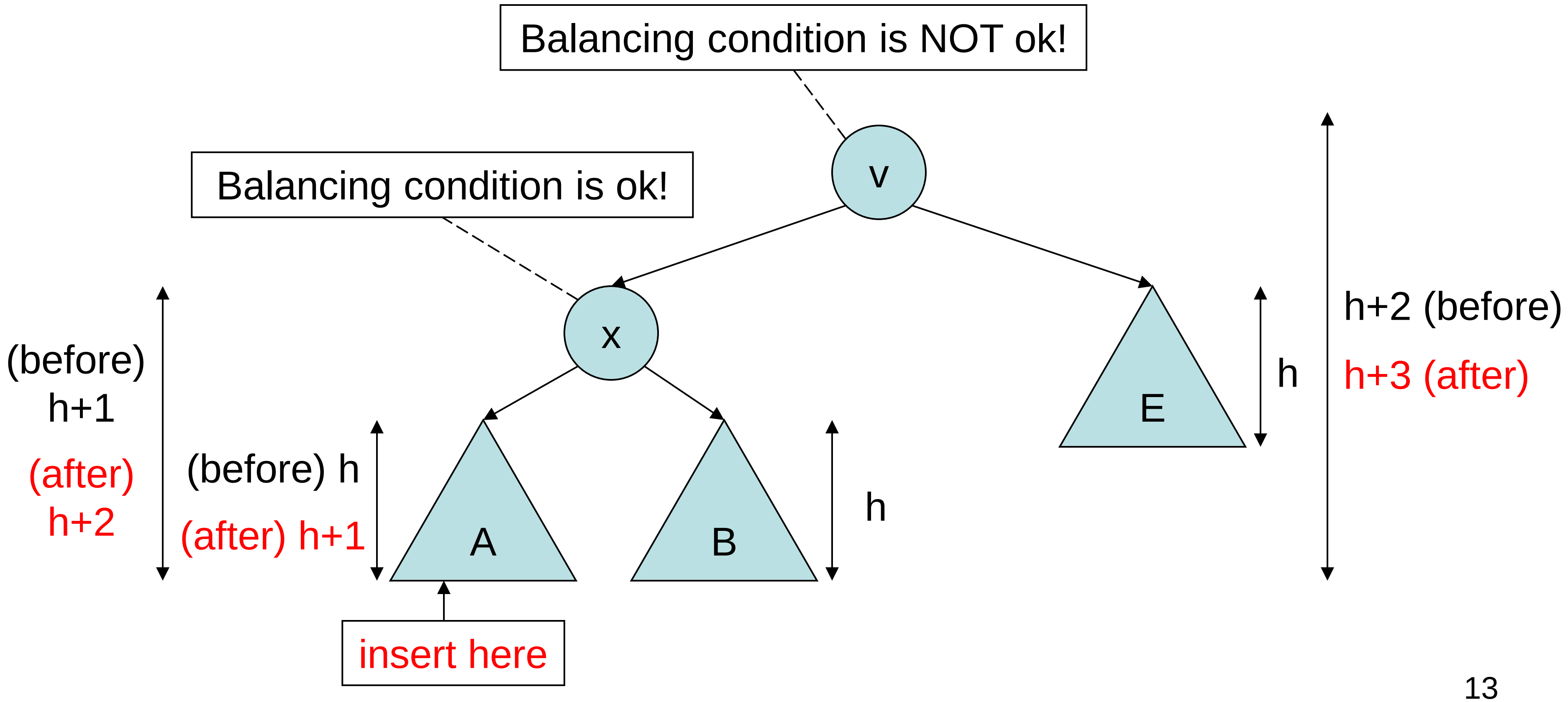


```
struct Node
{
   Node(item x):data(x), height(1),
               lson(NULL), rson(NULL){}
   item data;
   int height;  // height of subtree rooted at this
                // node
   Node* lson;
   Node* rson;
};

int h(Node* t){
   return t==NULL ? 0 : t->height;
}

int max(int x, int y){
   return x>=y ? x : y;
}
```

```
void AVL::insertR(Node*& t, item x)
{
  if (t==NULL) {t==new Node(x); return; }
  else if (x < t->data)
  {
    insertR(t->lson, x);                // insert
    if (h(t->lson)==h(t->rson)+2)       // checking
      if (x < t->lson->data) rotateL(t); // case 1
      else dbl_rotateL(t);              // case 2
  }
  else if (x > t->data)
  {
    insertR(t->rson, x);                 // insert
    if (h(t->rson)==h(t->lson)+2)        // checking
      if (x > t->rson->data) rotateR(t);  // case 4
      else dbl_rotateR(t);               // case 3
  }
  else ;  // duplication
  t->height = max( h(t->lson), h(t->rson) ) + 1;
}
```

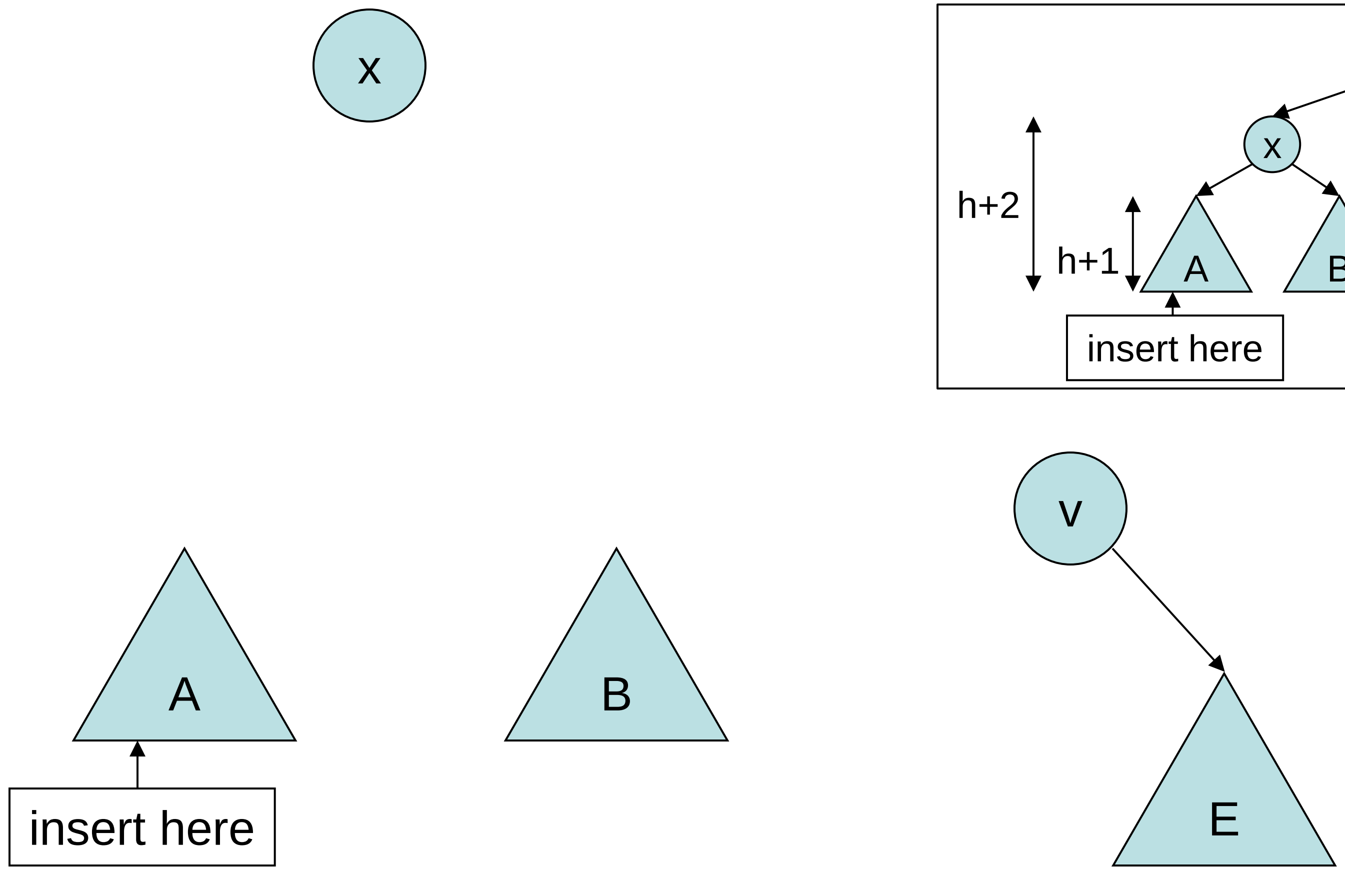
Insertion – Case 1 (1/7)

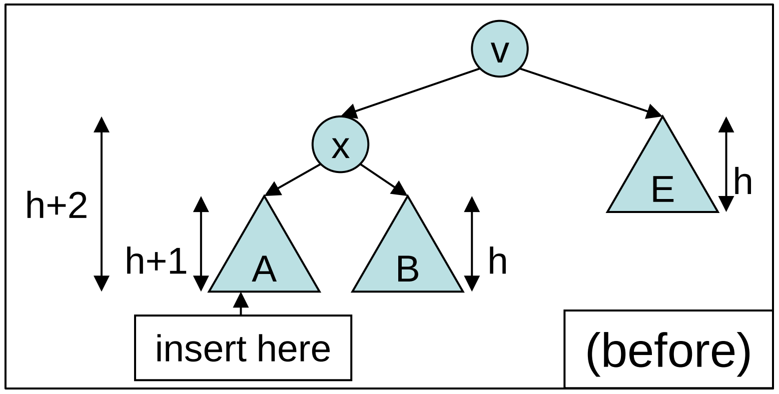
- Before insertion, height of subtree A = h and height of subtree E = h
- After insertion, height of subtree A = h+1



Insertion – Case 1 (2/7)

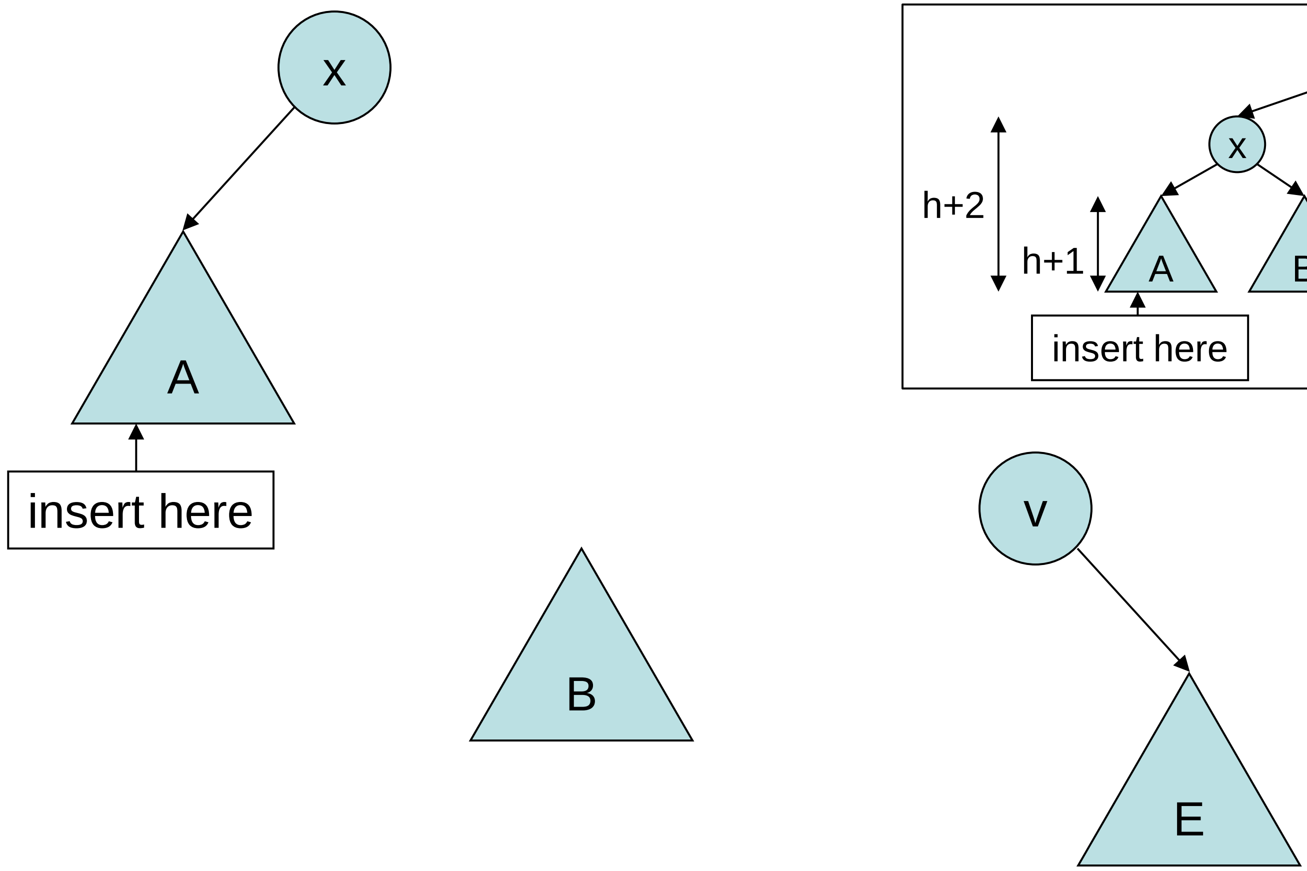
- Solution: single rotation, i.e., make x as root

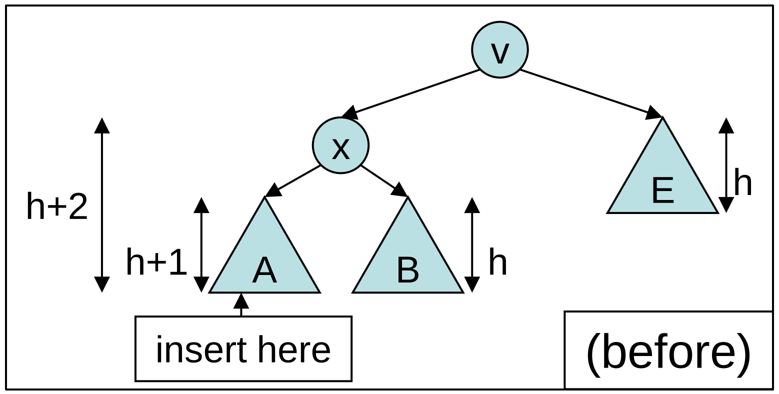




Insertion – Case 1 (3/7)

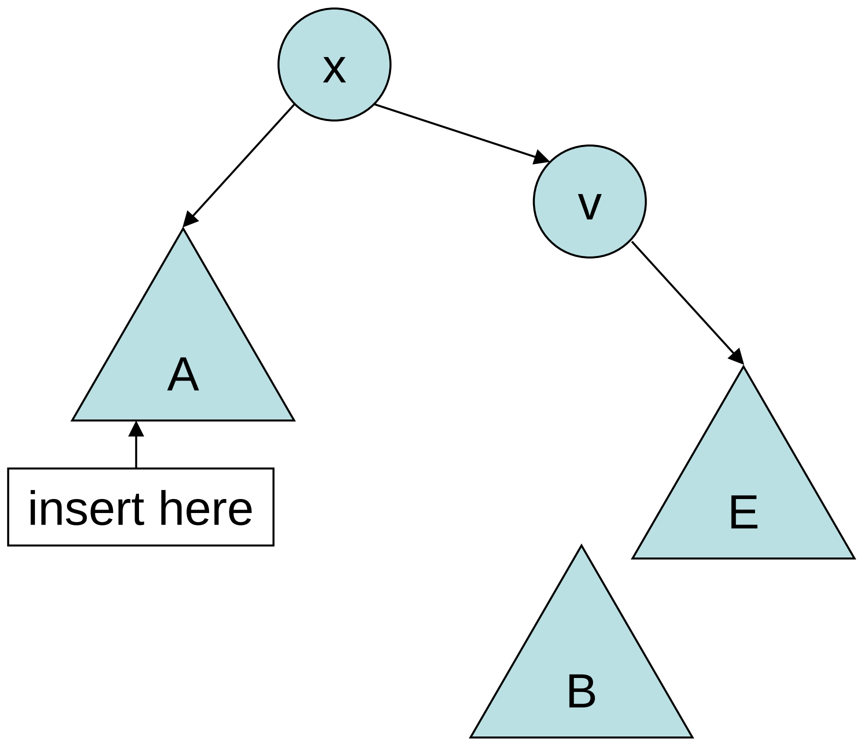
- Only A can be x’s left subtree

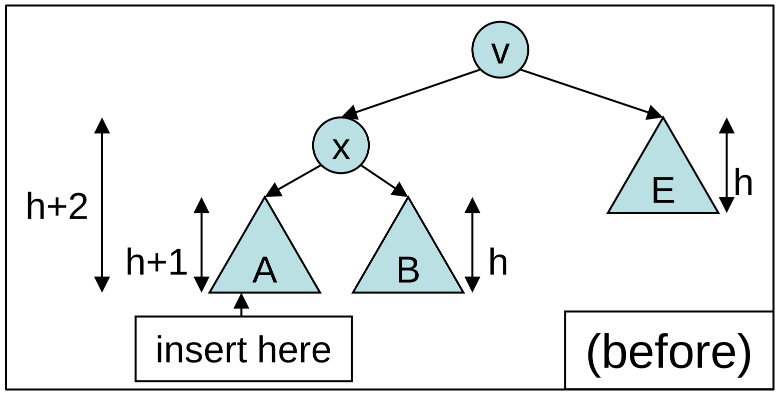




Insertion – Case 1 (4/7)

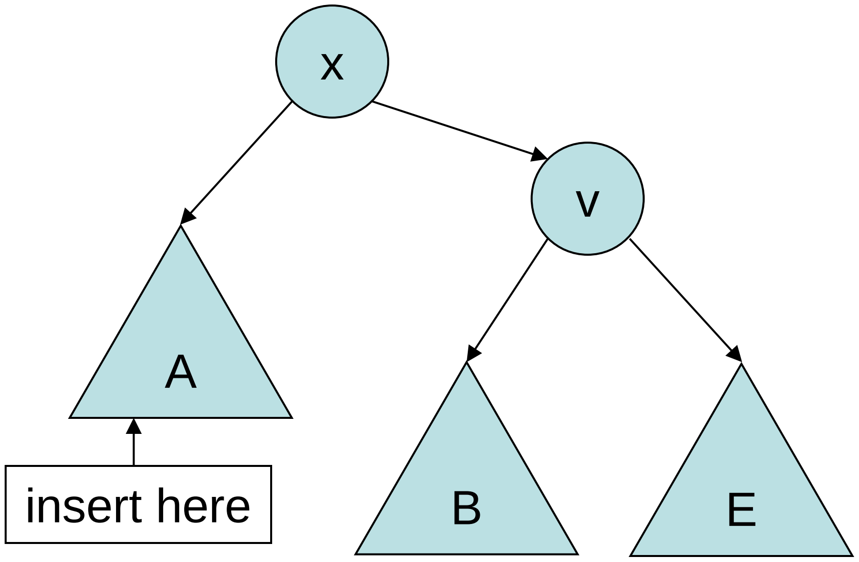
- v must be x’s right subtree

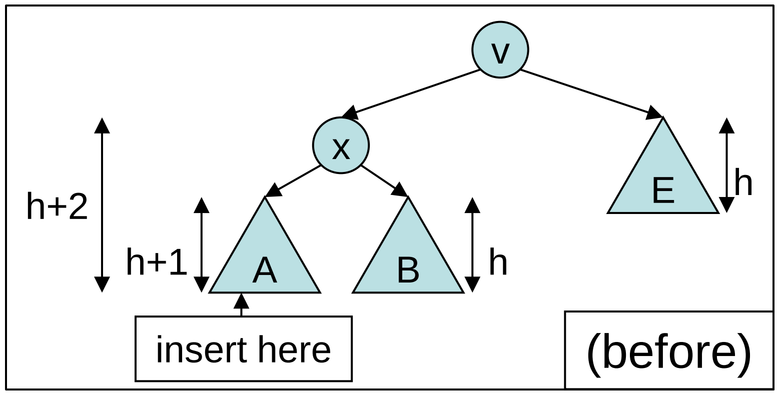




Insertion – Case 1 (5/7)

- B must be v's left subtree

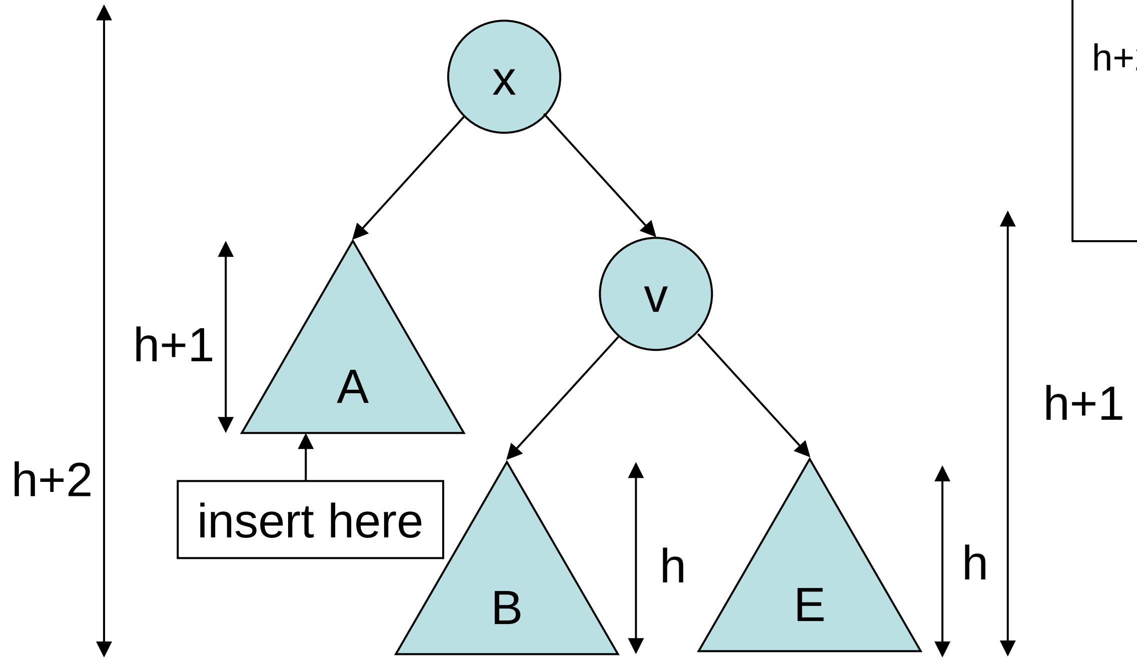


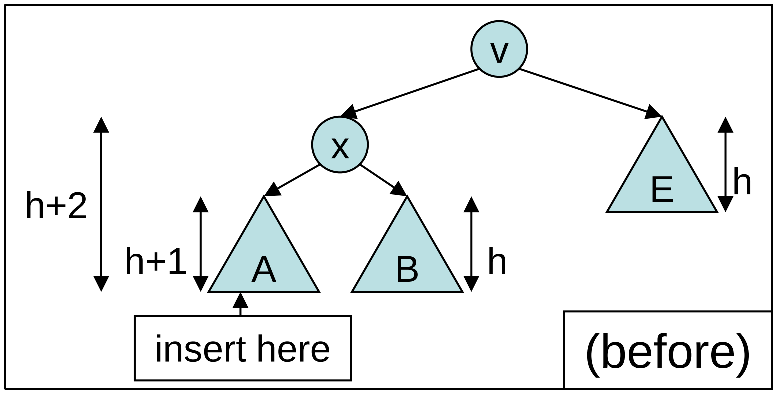


Insertion – Case 1 (6/7)

- After rotation: height of x = h+2

  (= height of subtree rooted at v before insertion)
- The left and right subtrees of x have the same height





Insertion – Case 1 (7/7)

```
void rotateL(Node*& s) // s & s->lson rotate
{
  Node* t=s->lson;
  s->lson=t->rson;
  t->rson=s;
  s->height=max( h(s->lson), h(s->rson) )+1;
  t->height=max( h(t->lson), s->height )+1;
  s=t;
}
```

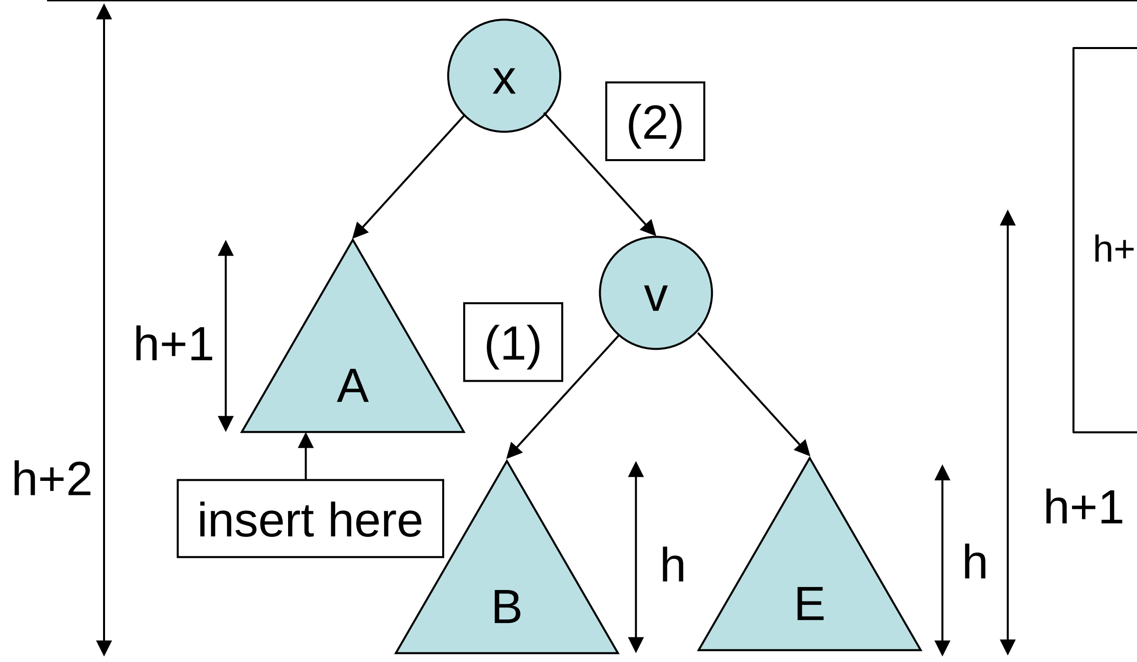
x

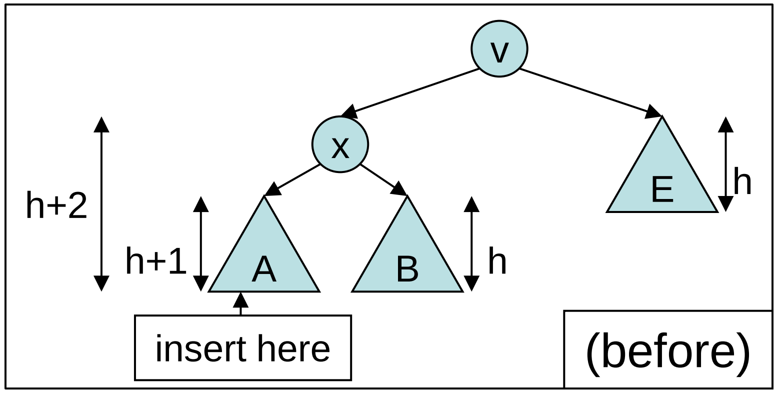
v

(1)

(2)

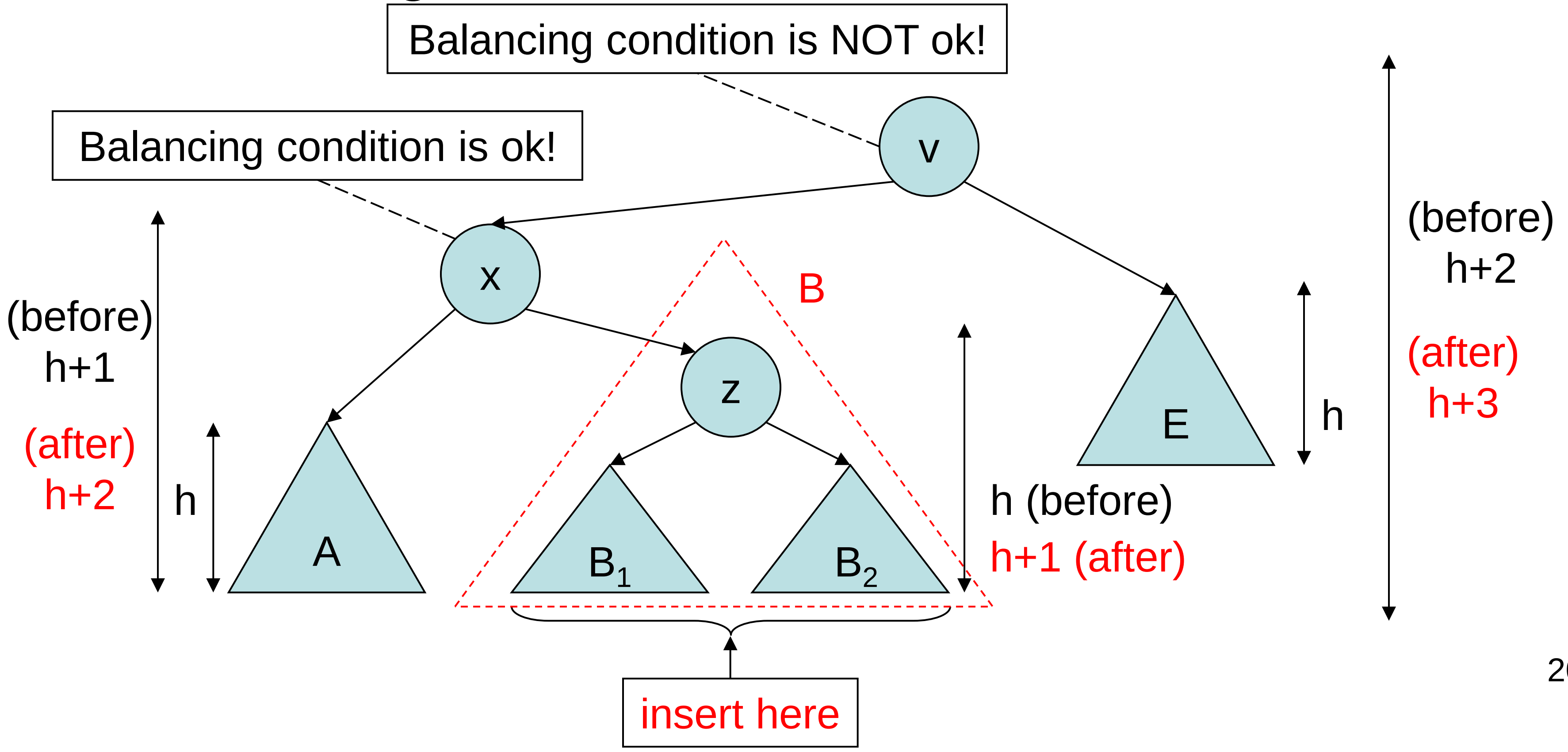
Make x as the root





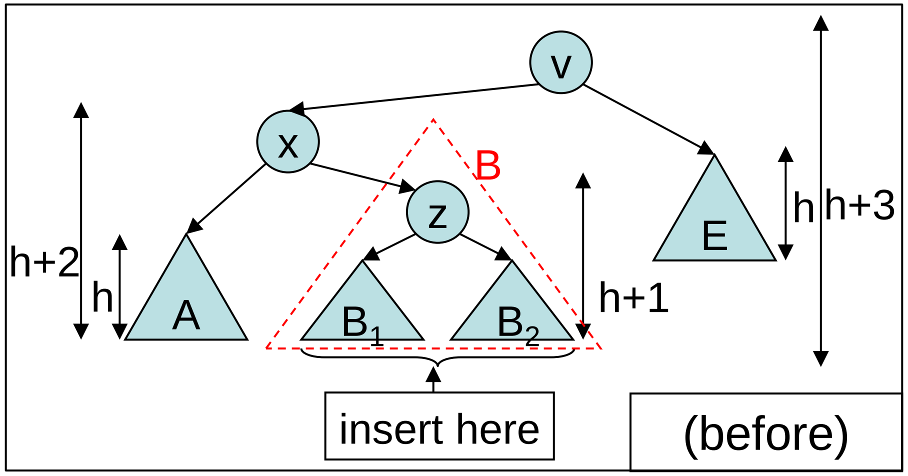
Insertion – Case 2 (1/11)

- Before insertion, height of subtree B = h and height of subtree E = h
- After insertion, height of subtree B = h+1, either B1 or B2 has height h, the other h-1

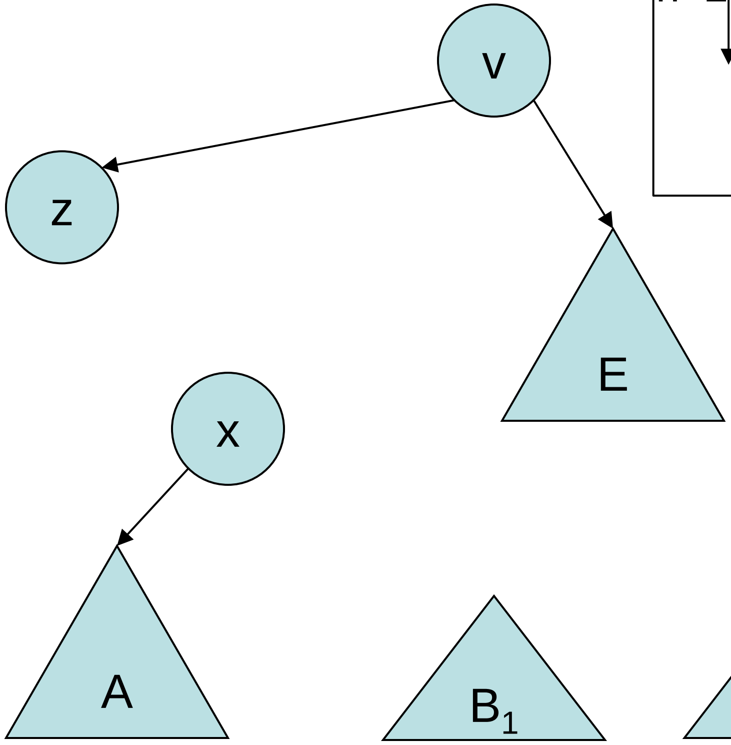


Insertion – Case 2 (2/11)

- Solution: double rotation, i.e., make z as root
- 1st rotation: make z as the root of its parent's subtree, i.e., its parent is x

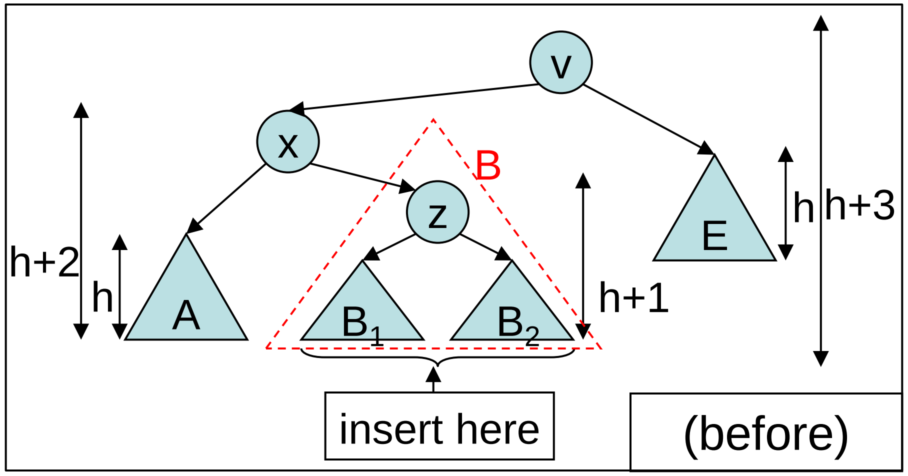


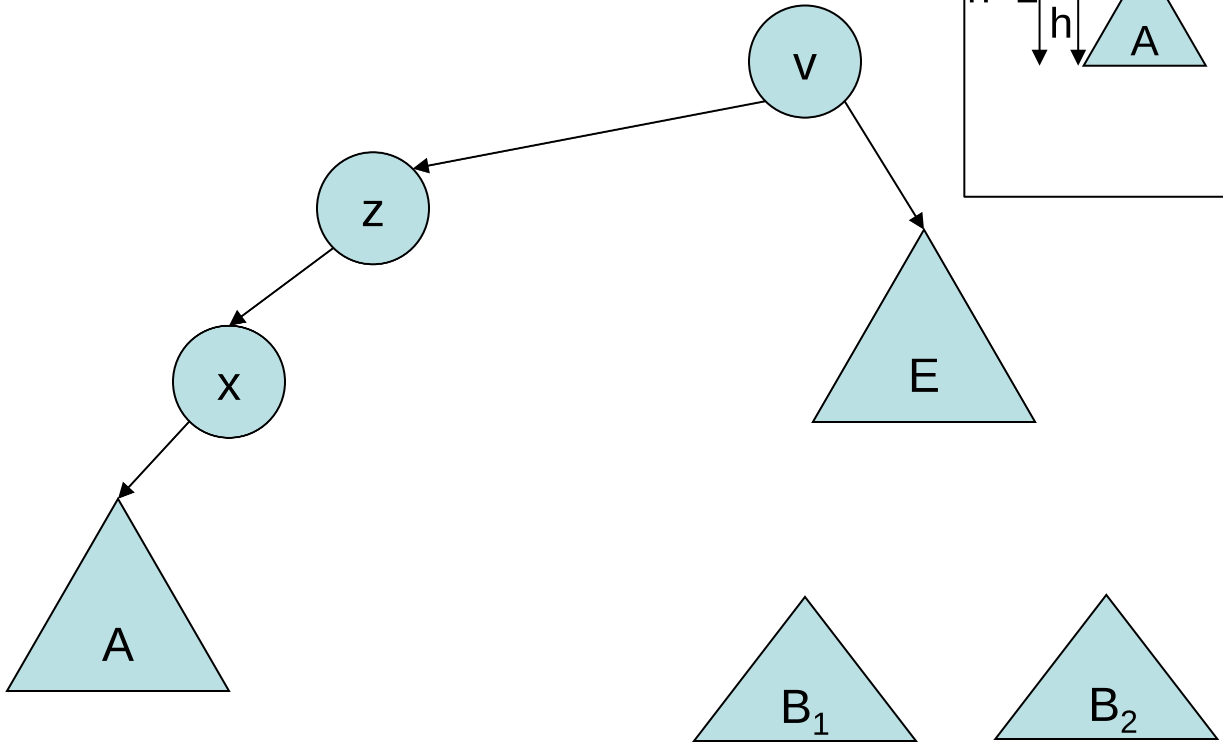
```
void dbl_rotateL(Node*& s)
{
    rotateR(s->lson);
    rotateL(s);
}
```



Insertion – Case 2 (3/11)

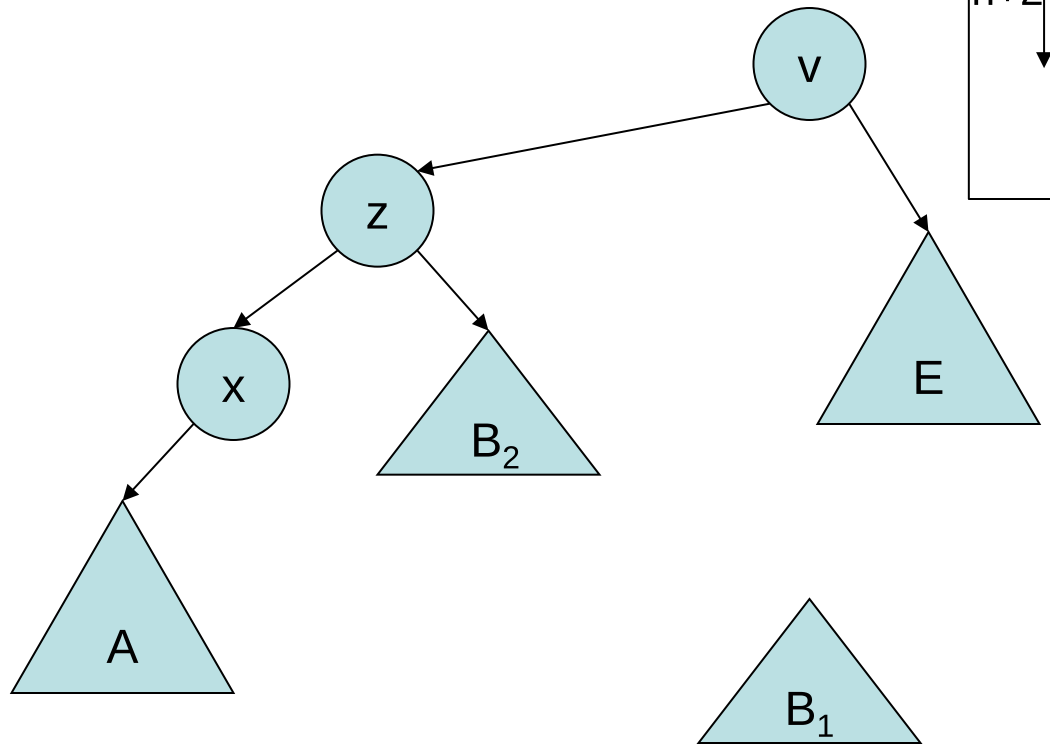
- 1st rotation: x must be z's left son

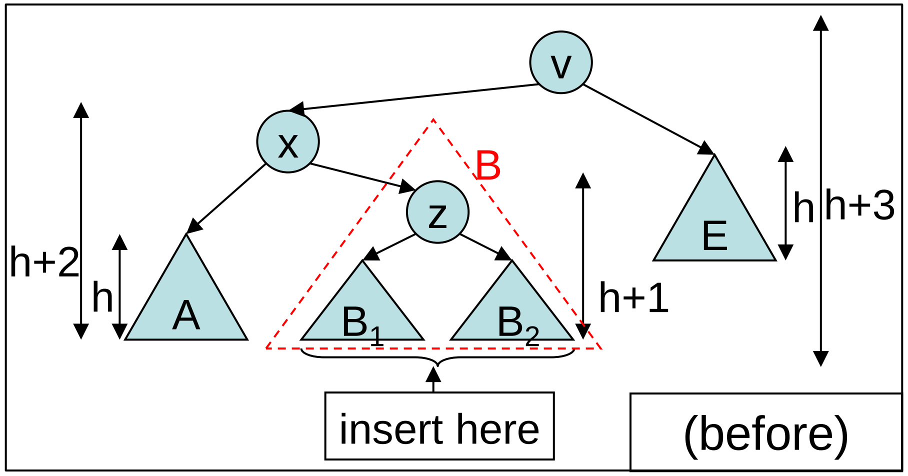




Insertion – Case 2 (4/11)

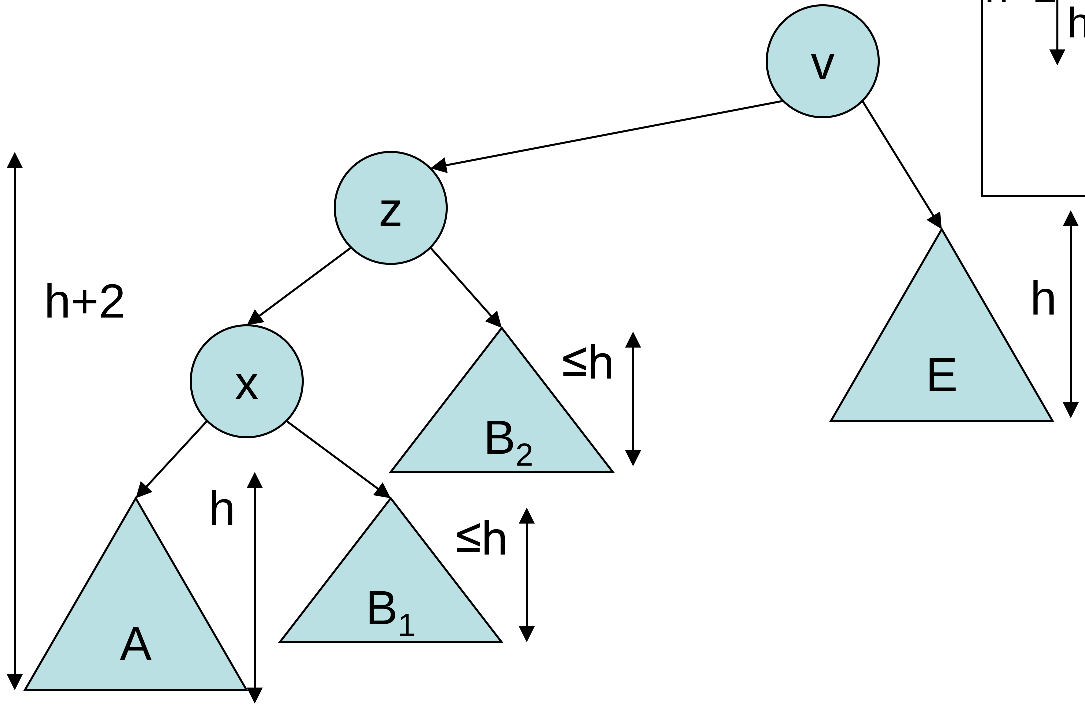
- 1st rotation: B_2 must be z's right subtree

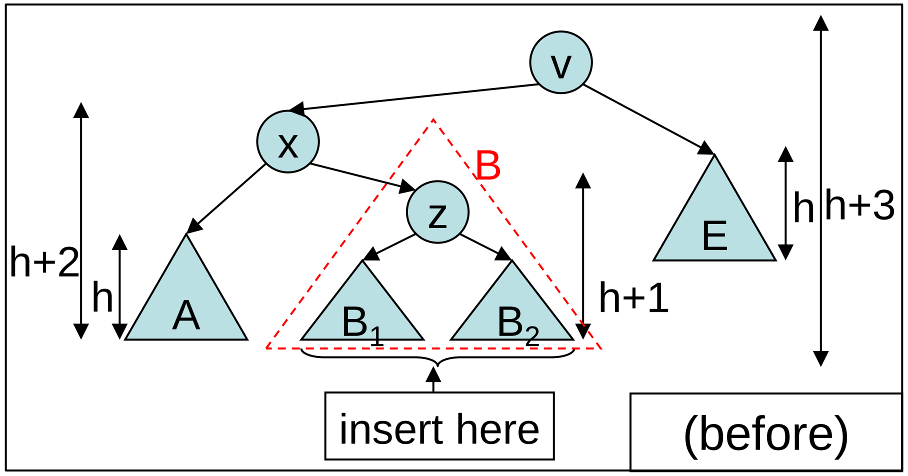




Insertion – Case 2 (5/11)

- 1st rotation: B_1 must be x's right subtree





Insertion – Case 2 (6/11)

```
void rotateR(Node*& s)  //s & s->rson rotate
{
   Node* t=s->rson;
   s->rson=t->lson;
   t->lson=s;
   s->height=max( h(s->lson), h(s->rson) )+1;
   t->height=max( h(t->rson), s->height )+1;
   s=t;
}
```

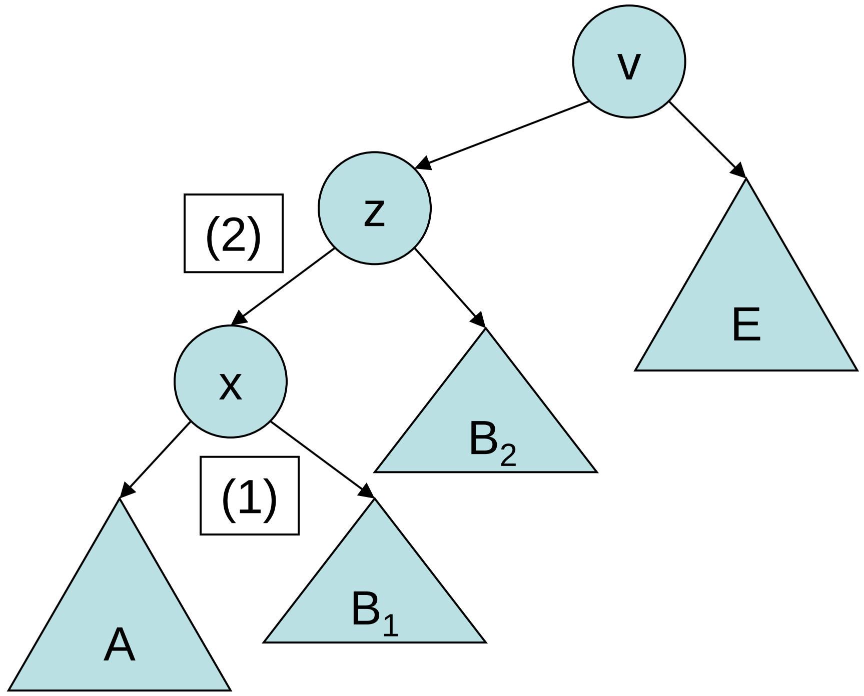
z

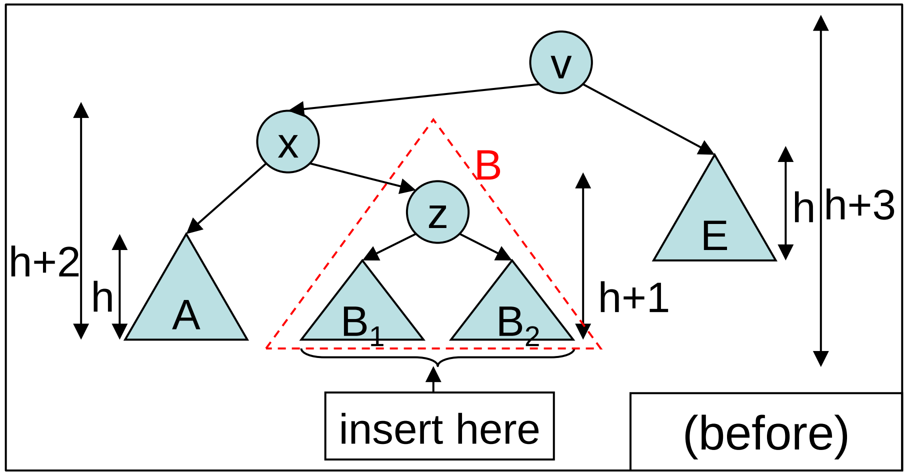
x

(1)

(2)

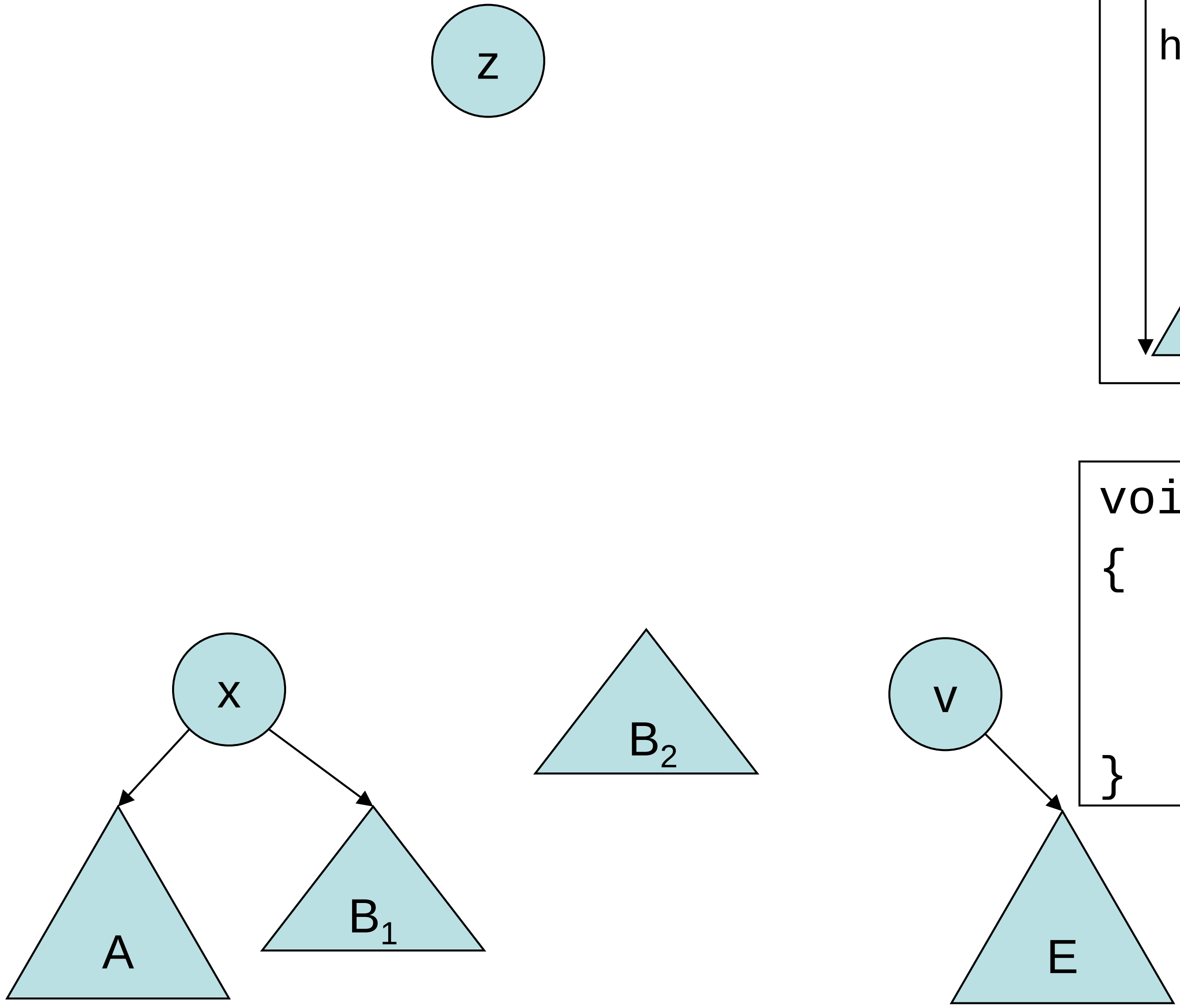
Make z as the root

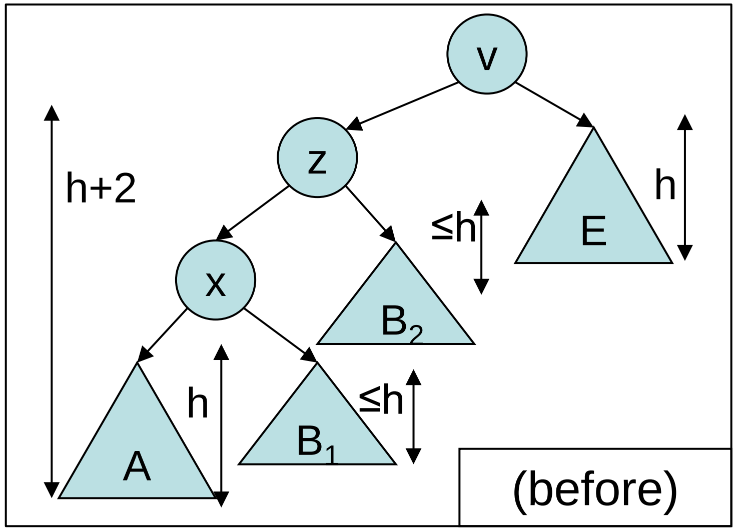




Insertion – Case 2 (7/11)

- 2nd rotation: make z as the root

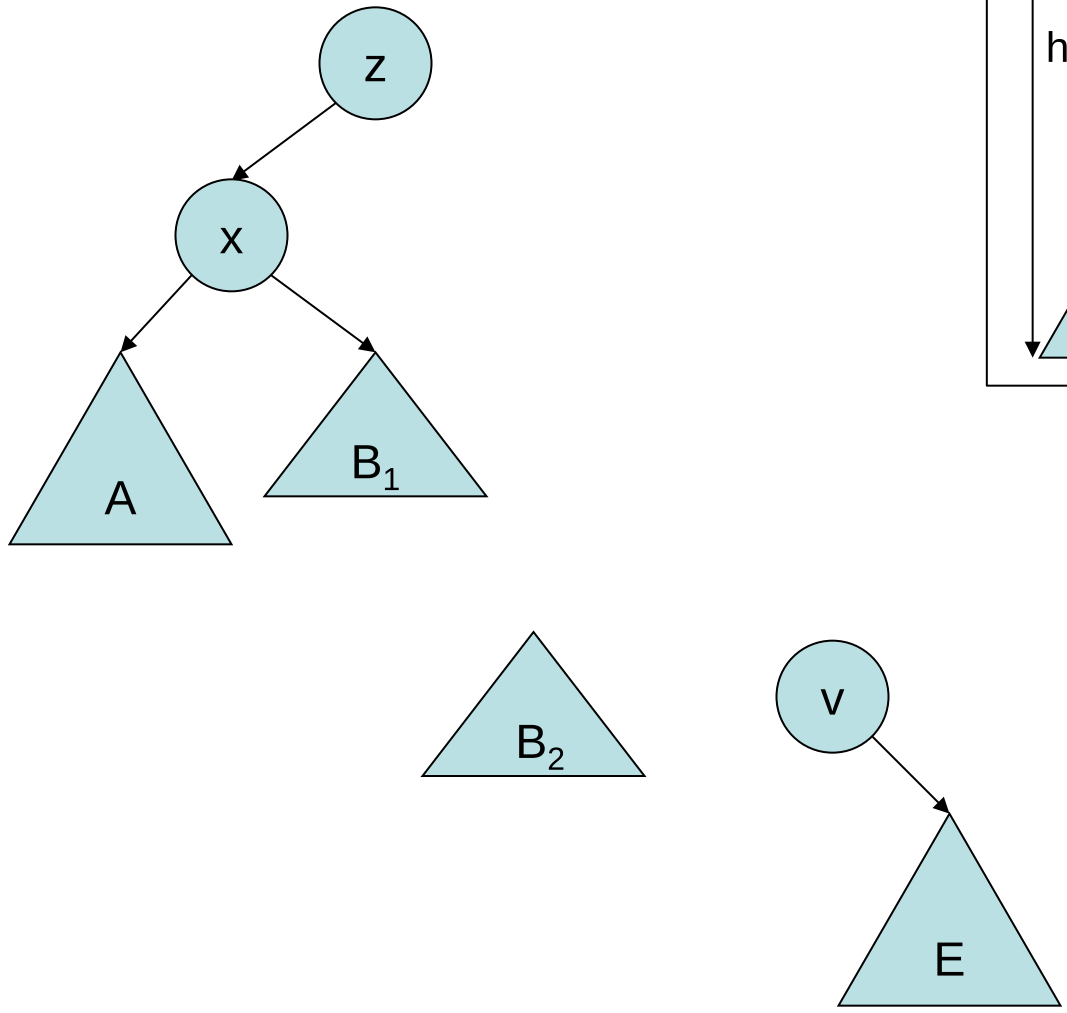


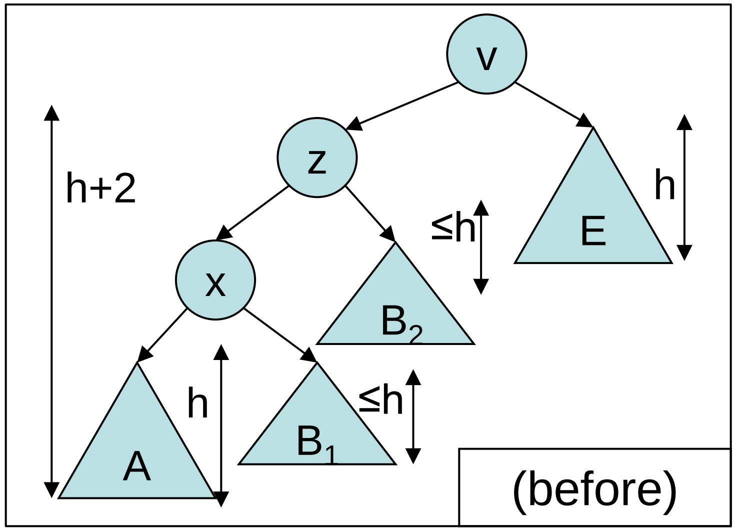


```
void dbl_rotateL(Node*& s)
{
   rotateR(s->lson);
   rotateL(s);
}
```

Insertion – Case 2 (8/11)

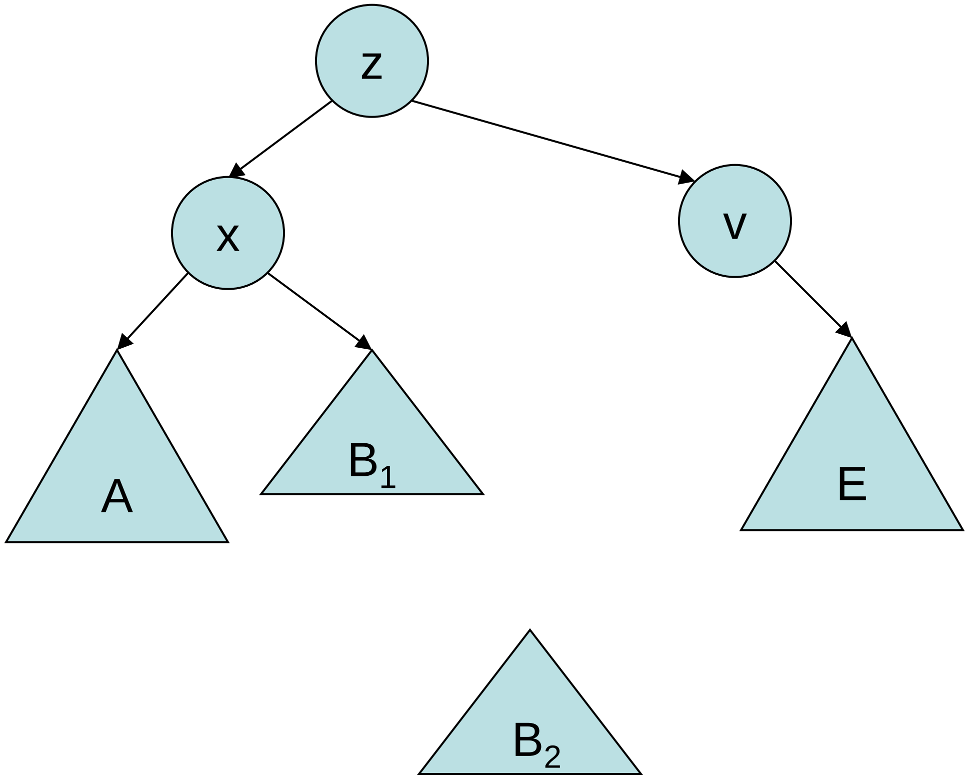
- 2nd rotation: x must be z's left subtree

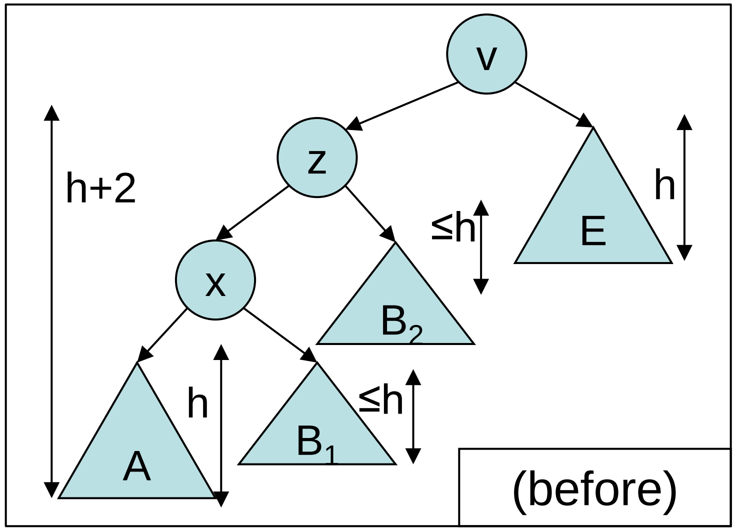




Insertion – Case 2 (9/11)

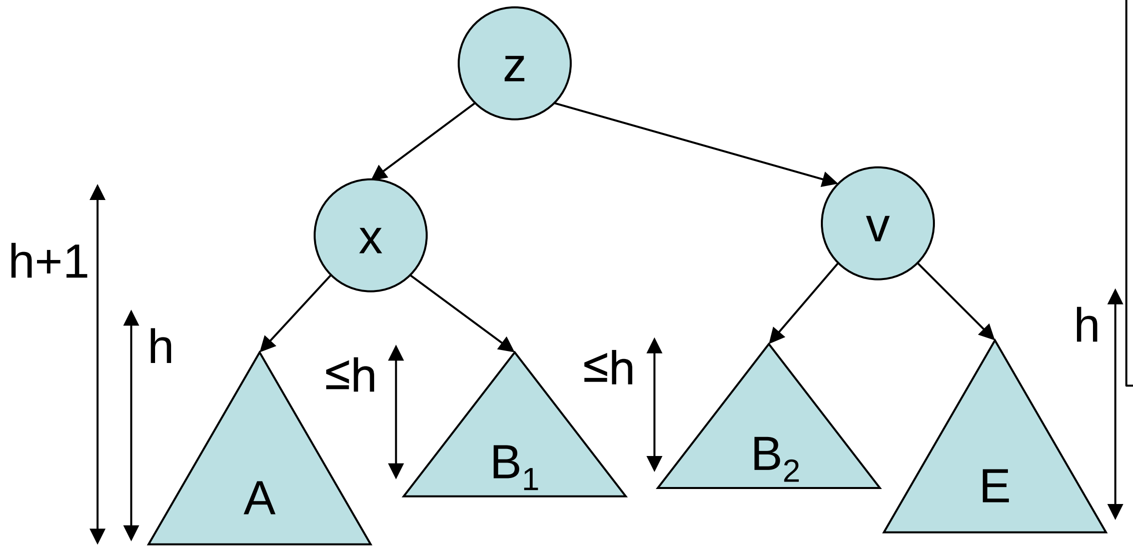
- 2nd rotation: v must be z's right subtree

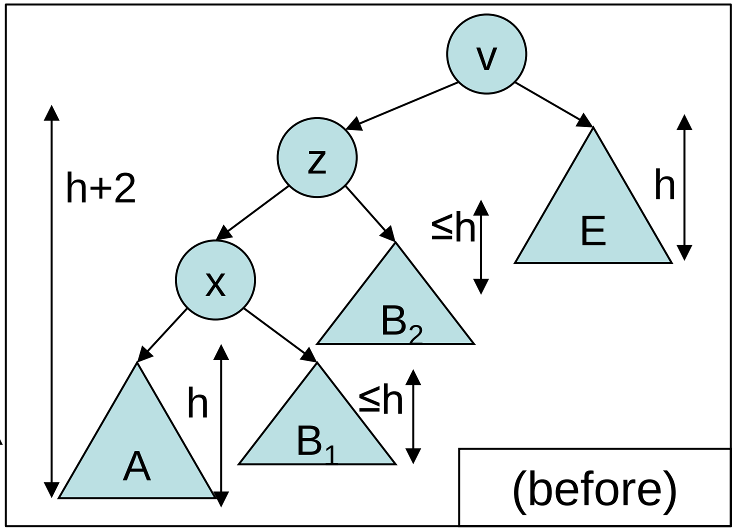




Insertion – Case 2 (10/11)

- 2nd rotation: B_2 must be v's left subtree





- After rotation: height of z = h+2 (= height of subtree rooted at v before insertion)
- The left and right subtrees of z have the same height

Insertion – Case 2 (11/11)

```
void rotateL(Node*& s) // s & s->lson rotate
{
  Node* t=s->lson;
  s->lson=t->rson;
  t->rson=s;
  s->height=max( h(s->lson), h(s->rson) )+1;
  t->height=max( h(t->lson), s->height )+1;
  s=t;
}
```

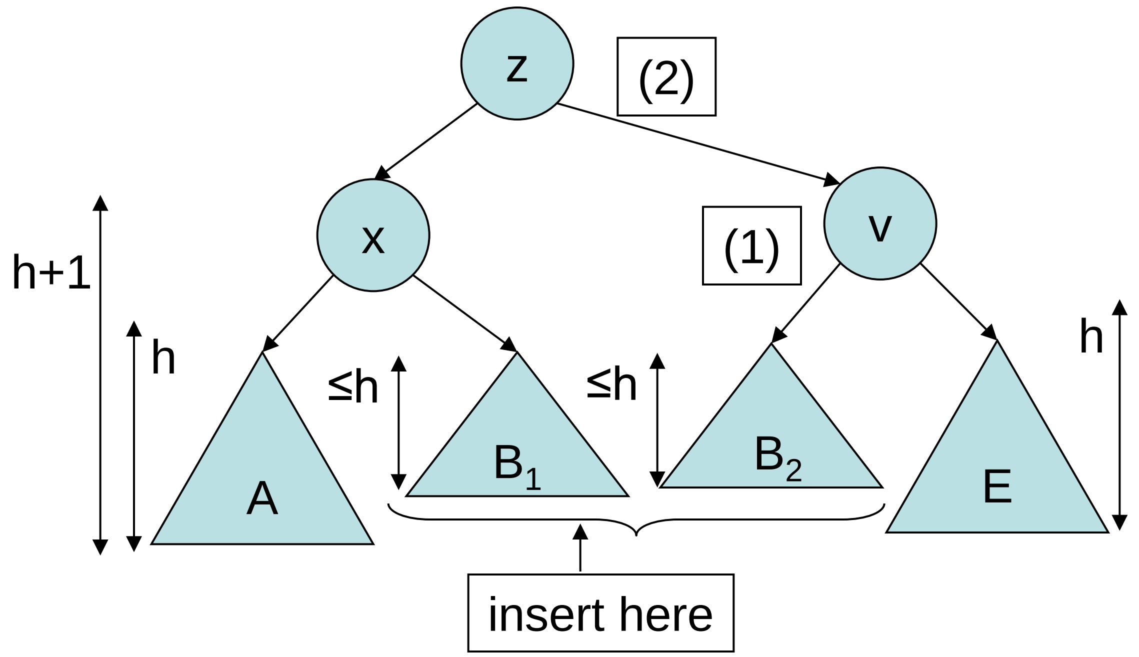
v

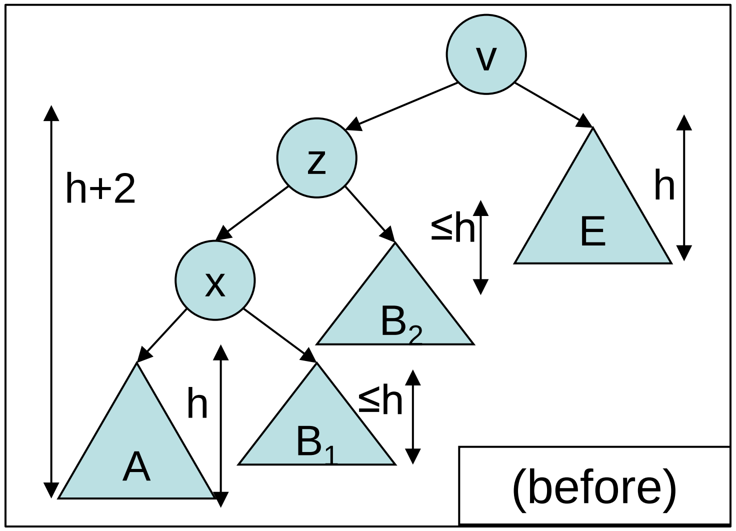
z

(1)

(2)

Make z as the root





Overall Scheme for insert(x)

- Search for x in the tree; insert a new leaf for x (as in previous BST)
- If parent of x not balanced, perform single or double rotation as appropriate
  - How do we know the height of a subtree?
  - Have a “height” attribute in each node
- Set x = parent of x and repeat the above step until x = root

```
void rotateL(Node*& s) // s & s->lson rotate (Case 1)
{
  Node* t=s->lson;
  s->lson=t->rson; t->rson=s;
  s->height=max( h(s->lson), h(s->rson) )+1;
  t->height=max( h(t->lson), s->height )+1;
  s=t;
}

void rotateR(Node*& s)  //s & s->rson rotate (Case 4)
{
  Node* t=s->rson;
  s->rson=t->lson; t->lson=s;
  s->height=max( h(s->lson), h(s->rson) )+1;
  t->height=max( h(t->rson), s->height )+1;
  s=t;
}
```

```
void dbl_rotateL(Node*& s) // (Case 2)
{
   rotateR(s->lson);
   rotateL(s);
}

void dbl_rotateR(Node*& s) // (Case 3)
{
   rotateL(s->rson);
   rotateR(s);
}
```

Complexity

- Worst case time complexity of insertR(t,x):
  - Local work requires constant time c
  - At most 1 recursive call with tree height k-1 where k = height of subtree pointed to by t
  - So, T(k) = T(k-1) + c

$$\begin{aligned} T(k) &= T(k-1) + c \\ &= T(k-2) + 2c \\ &\cdots \\ &= T(0) + kc \\ &= O(k) \end{aligned}$$

(Another approach)

$$\left.\begin{aligned} T(k)&=T(k-1)+c \\ T(k-1)&=T(k-2)+c \\ T(k-2)&=T(k-3)+c \\ &\cdots \\ +)\quad T(1)&=T(0)+c \end{aligned}\right\} \text{k equations}$$

$$\overline{T(k)=T(0)+kc}$$

- Worst case time complexity of insert(x)

  = worst case time complexity of insertR(root, x)

  = O(h)

  where h = height of the whole tree

AVL Tree Theorem

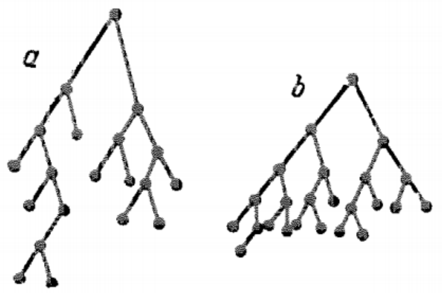


Figure 1

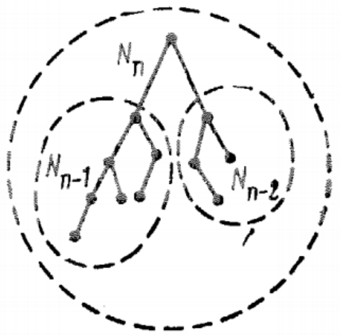


Figure 2

The recording algorithm is such that at each moment, the reference board is an admissible tree.

Lemma 1. *Let the number of cells of the admissible tree be equal to* N. *Then the maximum length of the branch is not greater than* $(3/2)\log_2(N+1)$.

Proof. Let us denote by N_n the minimum number of cells in the admissible tree when the given maximum length of the branch is n. Then it can be easily proven (see Figure 2) that $N_n = N_{n-1} + N_{n-2} + 1$.

When we solve this equation in finite remainders, we get

$$N_n = \left(1 + \frac{2}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(1 - \frac{2}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^n - 1.$$

Whence

$$n < \log_{\frac{1+\sqrt{5}}{2}}(N+1) < \frac{3}{2}\log_2(N+1),$$

q.e.d.

Recall the NIM Game
we played once before

AVL Tree Theorem

- What is h (as a function of n)?
- Theorem: Every AVL-tree of height h has

  $\geq \alpha^h - 1$ nodes

  – where $\alpha = (1+\mathrm{sqrt}(5))/2 \fallingdotseq 1.618$ (Golden ratio)

  – Note: $\alpha^2 = \alpha + 1$
- Proof: (by induction on h)

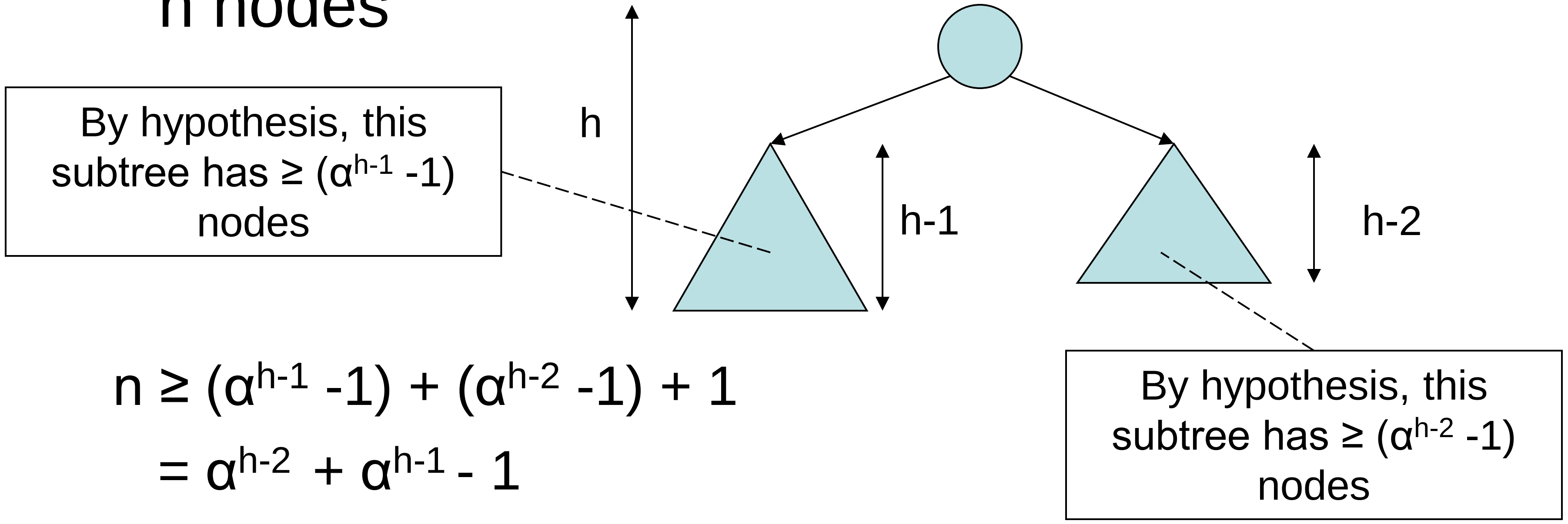
  Base Case (h=1)

  – An AVL tree of height 1 has 1 node

  – and $1 \geq \alpha^h - 1$

Induction Step

- Assume every AVL-tree of height k has $\geq \alpha^k - 1$ nodes for all k<h for some h>1
- Consider an arbitrary AVL tree of height h with n nodes

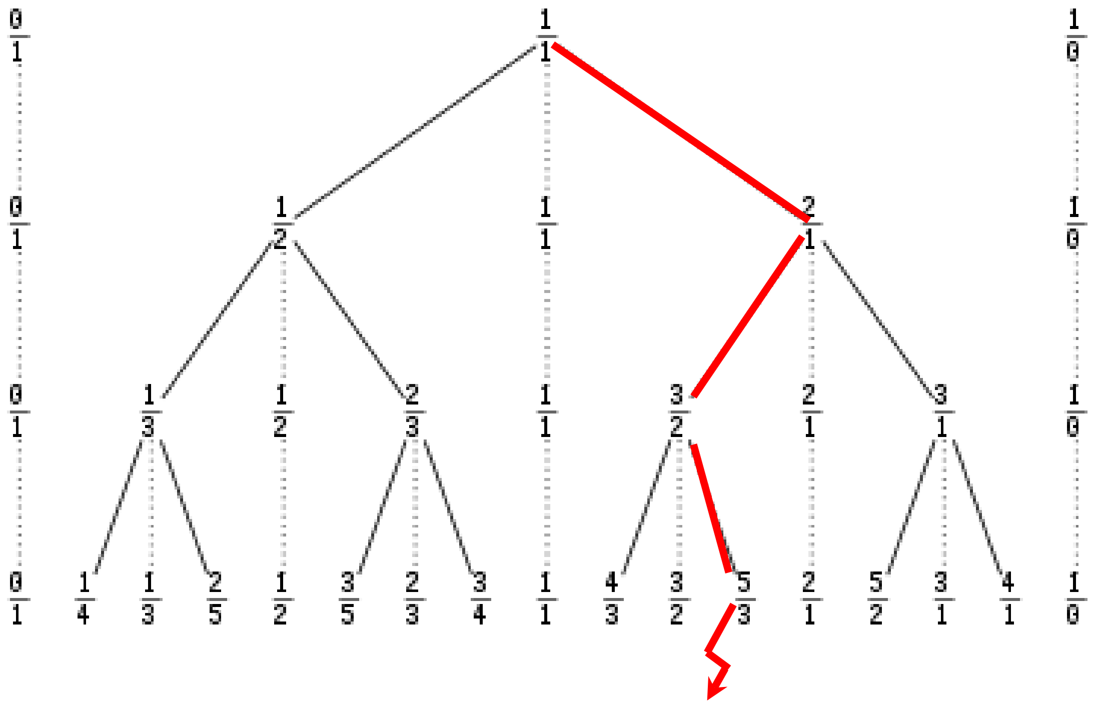


By hypothesis, this subtree has $\geq (\alpha^{h-1} - 1)$ nodes
h
h-1
h-2
By hypothesis, this subtree has $\geq (\alpha^{h-2} - 1)$ nodes

$$
\begin{aligned}
n &\geq (\alpha^{h-1} - 1) + (\alpha^{h-2} - 1) + 1 \\
&= \alpha^{h-2} + \alpha^{h-1} - 1 \\
&= \alpha^{h-2} (\alpha + 1) - 1 \\
&= \alpha^{h-2} (\alpha^2) - 1 = \alpha^h - 1 \text{ (proved)}
\end{aligned}
$$

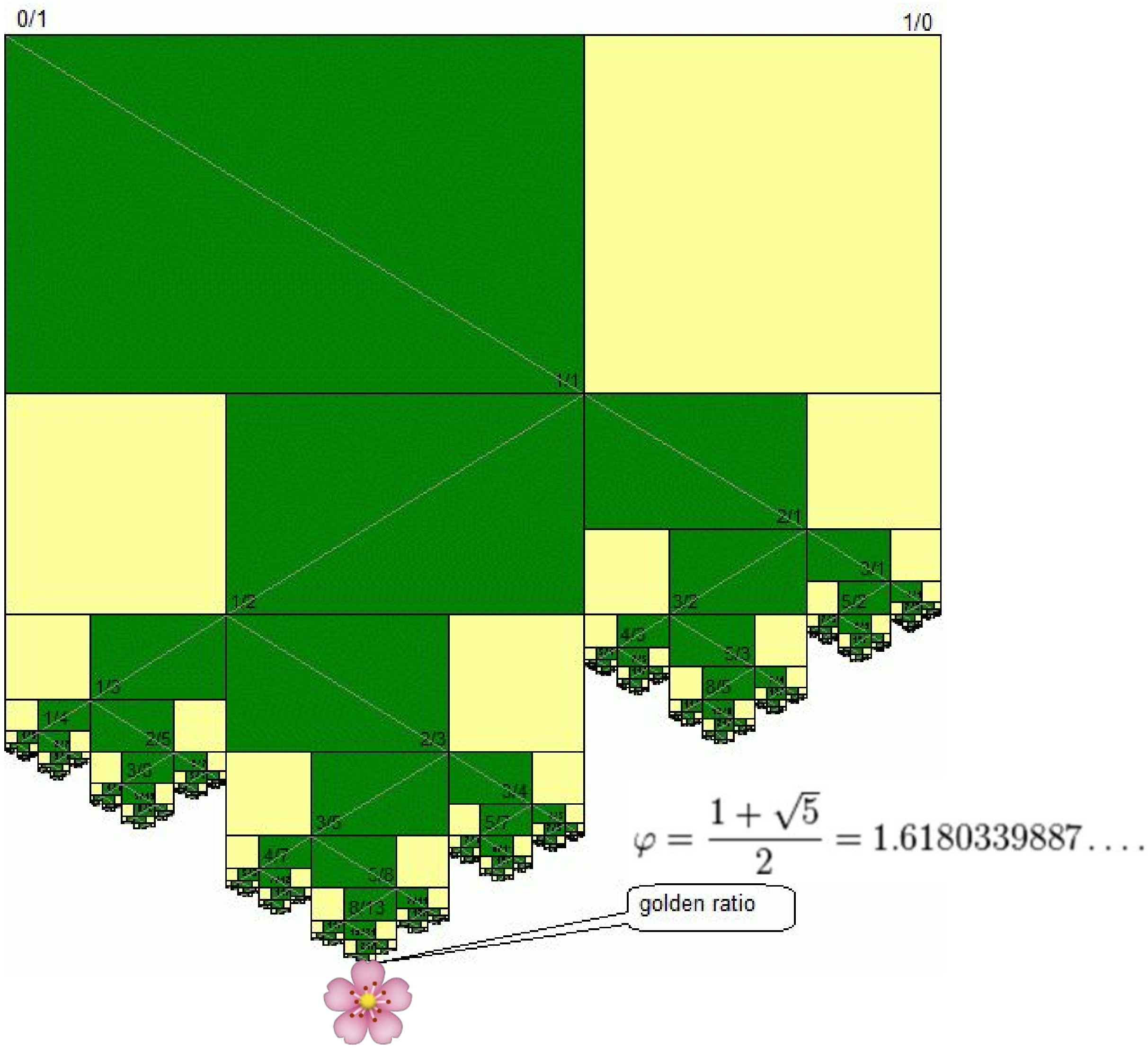
So, $\alpha^h \leq n+1$, i.e., $h \leq \log_\alpha(n+1) = O(\log n)$

A Pretty Flower in Stern-Brocot Tree



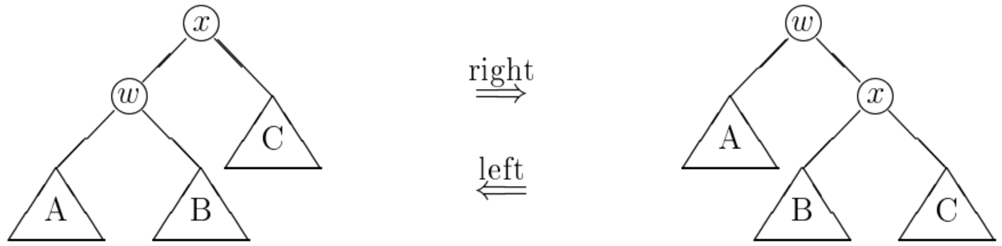
Can you write an algorithm to encode the Stern-Brocot representation of *e* (Euler number) and Pi (Archimedes’ constant)?
Graham, Knuth and Patashnik, “Concrete mathematics: A foundation for computer science,” Page 123

A Pretty Flower in Stern-Brocot Tree

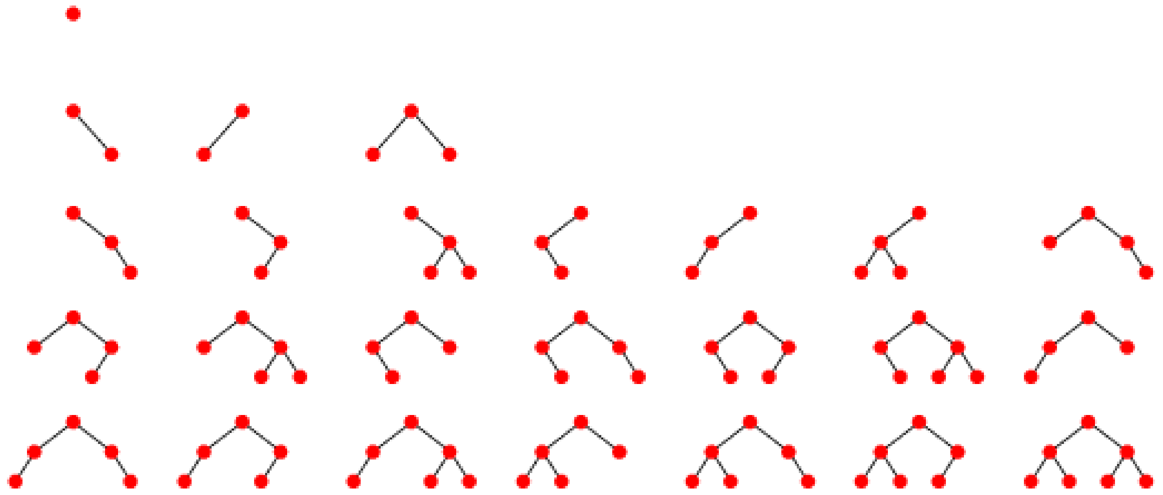


$$\varphi = \frac{1+\sqrt{5}}{2} = 1.6180339887\ldots.$$

Rotations in Trees



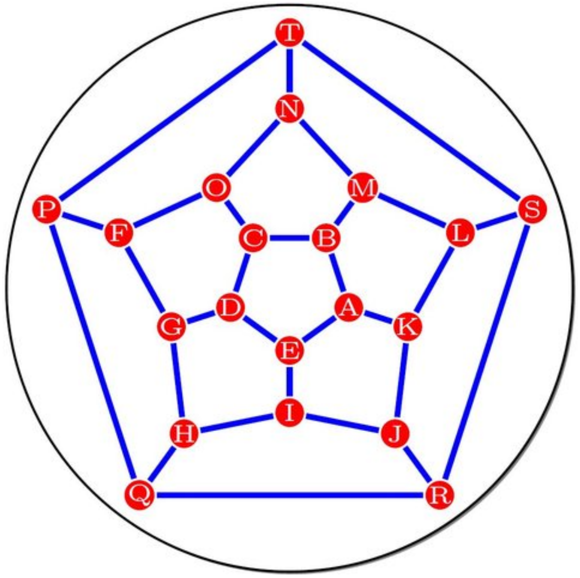
Rotations are fundamental primitives in data structures that alter the shape of the BST tree. How many possible shapes are there?
Rotations are fundamental mathematical primitives that play similar roles as comparison-based counting and swapping in permutation inversions



Let's Play a Game – Icosian Game

- The game consists of a graph in which 20 vertices represented major cities in Europe. The objective of the game is to find a path that visited each of the 20 vertices exactly once.
- In honor of Hamilton and his game, a path that uses each vertex of a graph exactly once is known as a **Hamiltonian path.**
- If the path ends at the starting vertex, it is called a **Hamiltonian circuit.**

Sir William Rowan Hamilton (1805-1865)



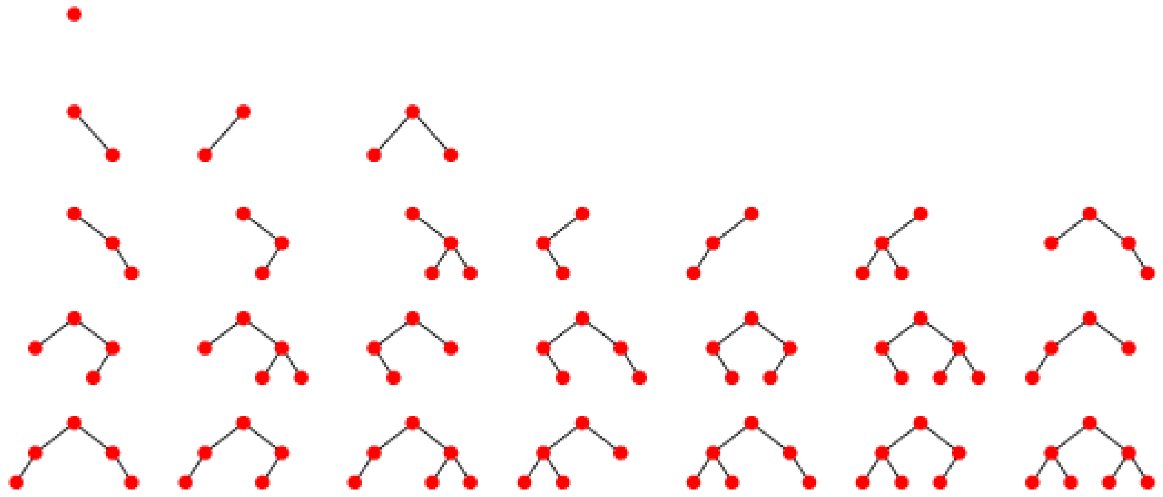
https://wordplay.blogs.nytimes.com/2014/10/06/icosian/

Mathematicians and computer scientists are intrigued by this type of problem, because a simple test for determining whether a graph has a Hamiltonian circuit has not been found. The search continues but it now appears that a general solution may be impossible.

Rotations in Trees are Hamiltonian

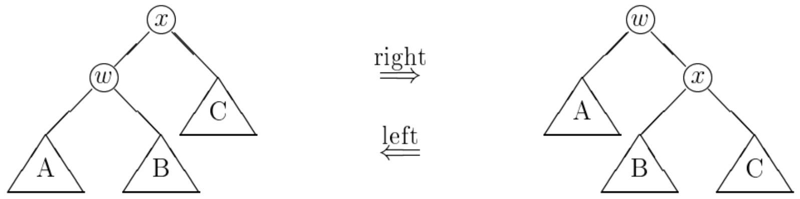
The number of binary trees with *n* nodes is the well-known Catalan number $\frac{\binom{2n}{n}}{n+1}$

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, ….



The rotation graph has vertex set consisting of all binary trees with *n* nodes. Two vertices are connected by an edge if a single rotation transforms one tree into the other.

Joan M. Lucas, Journal of Algorithms Volume 8(4), Dec 1987
https://www.cs.princeton.edu/research/techreps/TR-021-86



Let’s Play a Game – Icosian Game

- What does this integer sequence say?

0, 1, 8, 78, 944, 13800, 237432, …

https://wordplay.blogs.nytimes.com/2014/10/06/icosian/

Let’s Play a Game – Icosian Game

- What does this integer sequence say?

  0, 1, 8, 78, 944, 13800, 237432, …

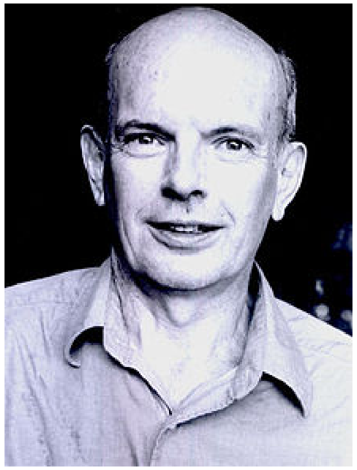
- Normalized total height of all nodes in all rooted trees with n labeled nodes.

- *The Sequence* that starts the On-Line Encyclopedia of Integer Sequences (OEIS) database project

- The OEIS also catalogs sequences of rational numbers (recall the Stern-Brocot Tree)

- J. Riordan and N. J. A. Sloane, Enumeration of rooted trees by total height, J. Austral. Math. Soc., vol. 10 pp. 278-282, 1969.

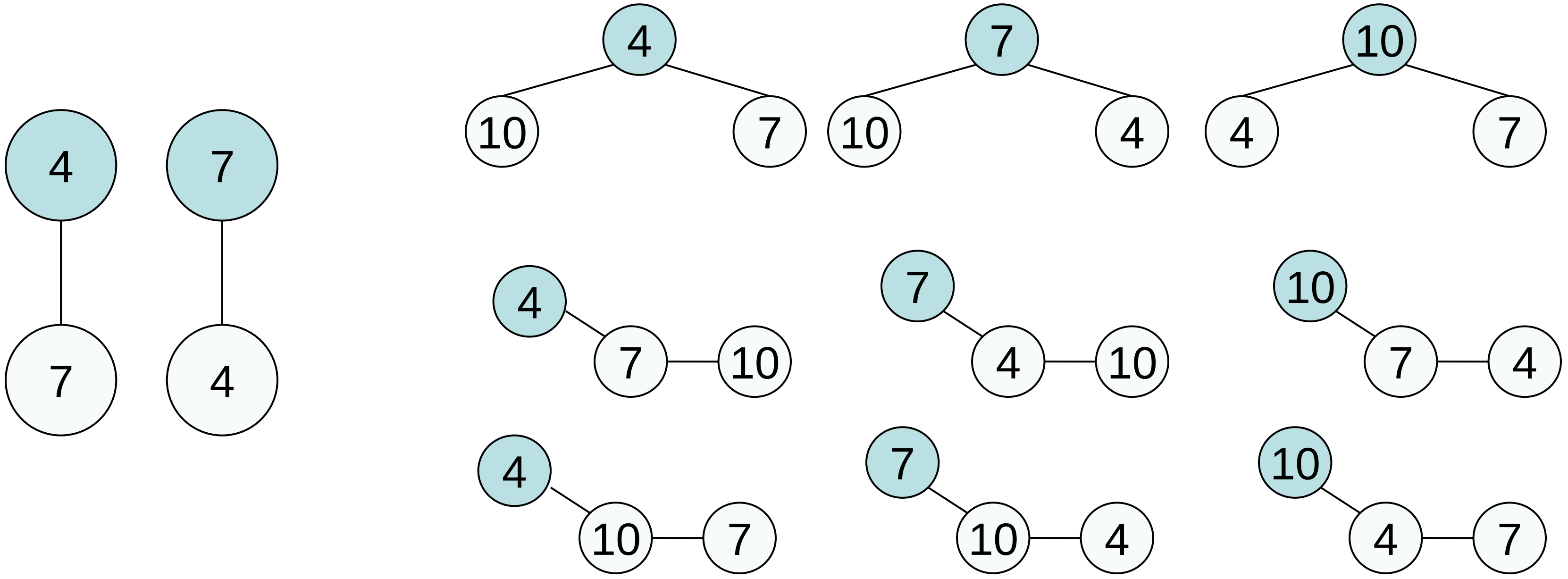
https://wordplay.blogs.nytimes.com/2014/10/06/icosian/


Neil Sloane

http://neilsloane.com/

Sloane, a British-American mathematician is best known for being the creator and maintainer of the On-Line Encyclopedia of Integer Sequences (OEIS).

Let’s Play a Game – Icosian Game

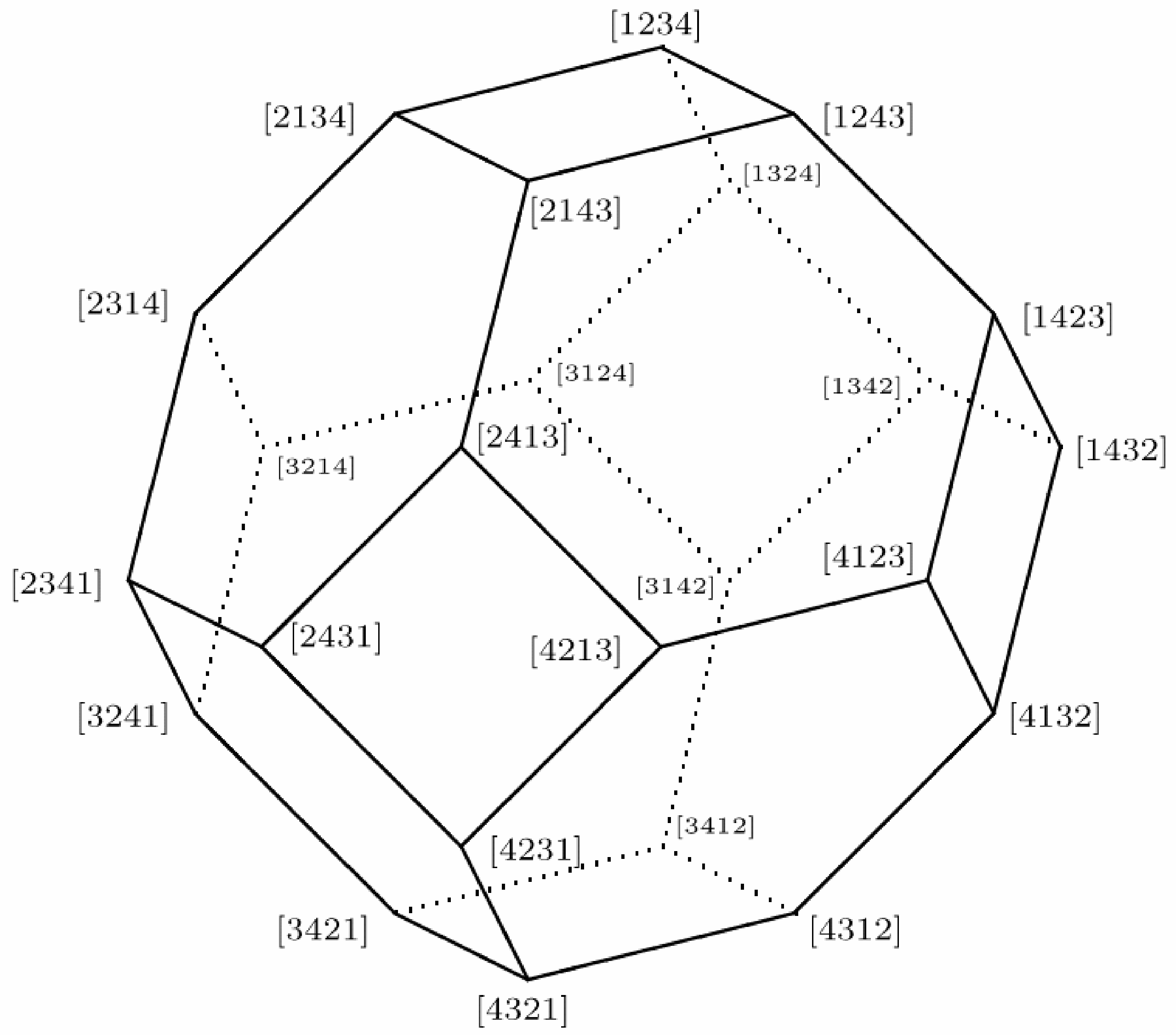


Number of rooted labelled trees = n^{n-1}

Shapes of BST trees are determined by the permutations of unique elements

The height of a point in a rooted tree is the length of the path from it to the root; the total height of a rooted tree is the sum of the heights of its points.

Let's Play the Icosian Game


[1234]
[2134]
[1243]
[1324]
[2143]
[2314]
[1423]
[3124]
[1342]
[2413]
[1432]
[3214]
[4123]
[2341]
[3142]
[2431]
[4213]
[3241]
[4132]
[3412]
[4231]
[3421]
[4312]
[4321]