Data Structures and Algorithms

Instructor: Prof. Tianyi ZANG

School of Computer Science and Technology

Harbin Institute of Technology

tianyi.zang@gmail.com

Searching

Instructor: Prof. Tianyi ZANG

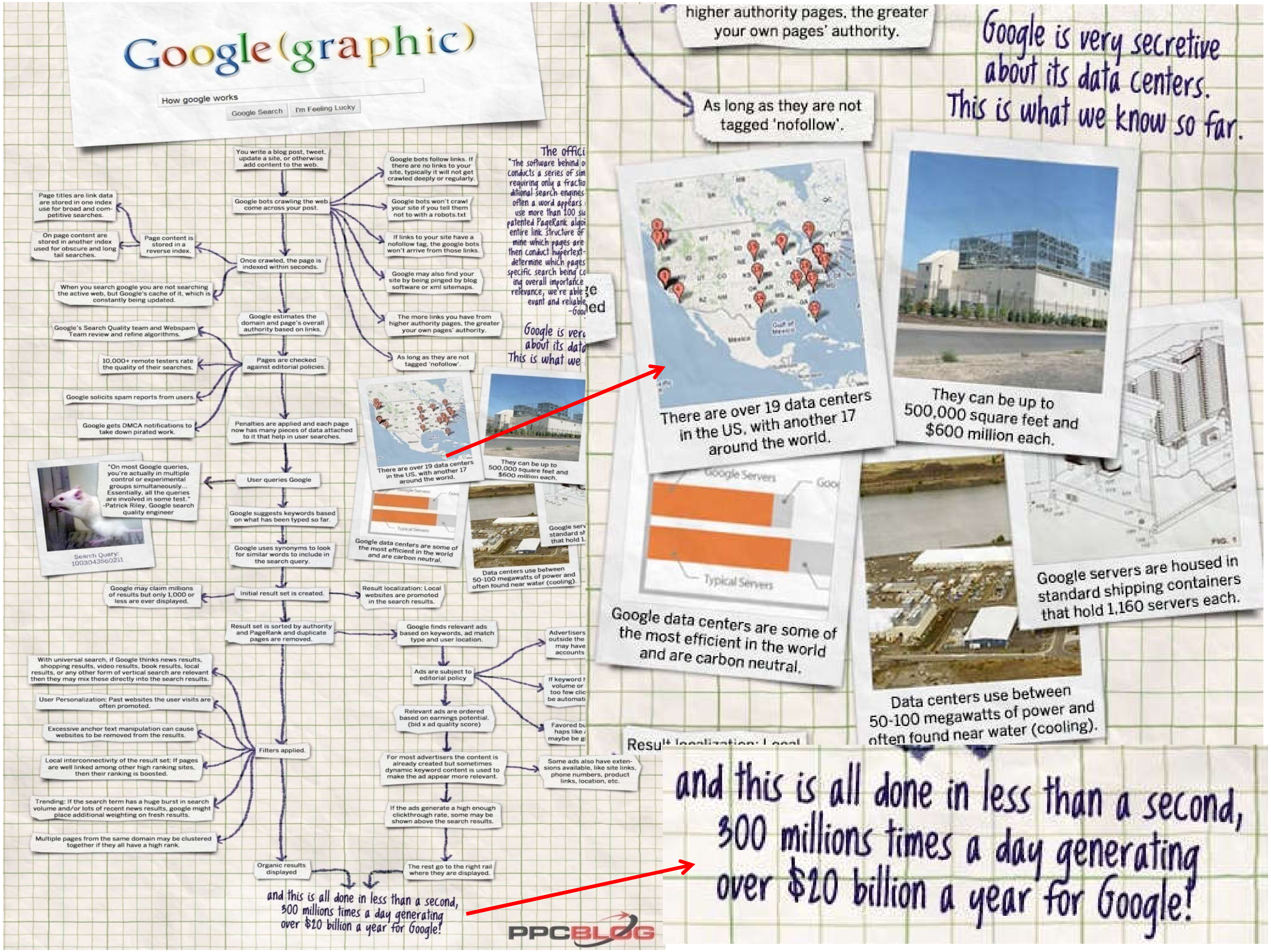
School of Computer Science and Technology

Harbin Institute of Technology

tianyi.zang@gmail.com

Outline

- Searching
- Searching on unsorted arrays
- Searching on sorted arrays
- Binary search
- Binary Search Trees (BST)
- Hash tables
- B tree



Google(graphic)
How google works
Google Search
I'm Feeling Lucky
You write a blog post, tweet, update a site, or otherwise add content to the web.
Google bots crawling the web come across your post.
Google bots follow links. If there are no links to your site, typically it will not get crawled deeply or regularly.
Google bots won't crawl your site if you tell them not to with a robots.txt
If links to your site have a nofollow tag, the google bots won't arrive from those links.
Google may also find your site by being pinged by blog software or xml sitemaps.
The more links you have from higher authority pages, the greater your own pages' authority.
As long as they are not tagged 'nofollow'.
Page titles are link data are stored in one index use for broad and competitive searches.
On page content are stored in another index used for obscure and long tail searches.
Page content is stored in a reverse index.
Once crawled, the page is indexed within seconds.
When you search google you are not searching the active web, but Google's cache of it, which is constantly being updated.
Google estimates the domain and page's overall authority based on links.
Google's Search Quality team and Webspam Team review and refine algorithms.
10,000+ remote testers rate the quality of their searches.
Pages are checked against editorial policies.
Google solicits spam reports from users.
Google gets DMCA notifications to take down pirated work.
Penalties are applied and each page now has many pieces of data attached to it that help in user searches.
"On most Google queries, you're actually in multiple control or experimental groups simultaneously... Essentially, all the queries are involved in some test." -Patrick Riley, Google search quality engineer
Search Query: 1003043560211
User queries Google
Google suggests keywords based on what has been typed so far.
Google uses synonyms to look for similar words to include in the search query.
Google may claim millions of results but only 1,000 or less are ever displayed.
Initial result set is created.
Result localization: Local websites are promoted in the search results.
Result set is sorted by authority and PageRank and duplicate pages are removed.
Google finds relevant ads based on keywords, ad match type and user location.
Ads are subject to editorial policy
Relevant ads are ordered based on earnings potential. (bid x ad quality score)
For most advertisers the content is already created but sometimes dynamic keyword content is used to make the ad appear more relevant.
Some ads also have extensions available, like site links, phone numbers, product links, location, etc.
If the ads generate a high enough clickthrough rate, some may be shown above the search results.
The rest go to the right rail where they are displayed.
With universal search, if Google thinks news results, shopping results, video results, book results, local results, or any other form of vertical search are relevant then they may mix these directly into the search results.
User Personalization: Past websites the user visits are often promoted.
Excessive anchor text manipulation can cause websites to be removed from the results.
Local interconnectivity of the result set: If pages are well linked among other high ranking sites, then their ranking is boosted.
Trending: If the search term has a huge burst in search volume and/or lots of recent news results, google might place additional weighting on fresh results.
Multiple pages from the same domain may be clustered together if they all have a high rank.
Filters applied.
Organic results displayed
Google is very secretive about its data centers. This is what we know so far.
There are over 19 data centers in the US, with another 17 around the world.
They can be up to 500,000 square feet and $600 million each.
Google Servers
Typical Servers
Google data centers are some of the most efficient in the world and are carbon neutral.
Data centers use between 50-100 megawatts of power and often found near water (cooling).
Google servers are housed in standard shipping containers that hold 1,160 servers each.
and this is all done in less than a second, 300 millions times a day generating over $20 billion a year for Google!
PPCBLOG

Searching

- Search can be viewed abstractly as a process to determine if an element with a particular value is a member of a particular set.

- The more common view of searching is an attempt to find the record within a collection of records that has
  - a particular key value, or
  - those records in a collection whose key values meet some criterion such as falling within a range of values.

Searching: formal definition

- Suppose that we have a collection L of n records of the form
  $(k_1; I_1); (k_2; I_2); :::; (k_n; I_n)$
  where I_j is information associated with key k_j from record j for $1 \leq j \leq n$.
- Given a particular key value K, the search problem is to locate a record $(k_j ; I_j)$ in L such that $k_j = K$(if one exists).

- Searching is a systematic method for locating the record (or records) with key value $k_j = K$.

Searching

- A successful search is one in which a record with key $k_j = K$ is found.
- An unsuccessful search is one in which no record with $k_j = K$ is found (and no such record exists).
- An exact-match query is a search for the record whose key value matches a specified key value.
- A range query is a search for all records whose key value falls within a specified range of key values.

Searching Algorithms

- Categorize search algorithms into three general approaches:
  1. Sequential and list methods.
     - Key value based
  2. Tree indexing methods.
     - Key value based
  3. Direct access by key value (hashing)
     - Location-based

Searching on Unsorted Arrays

- **Sequential search algorithm**
  - the simplest form of search
  - Sequential search on an unsorted list requires time in the worst case. O(n)
- **Basic algorithm:**

  Get the search criterion (key)

  Get the first record from the file

  While ((record != key) and (still more records))

      Get the next record

  End_while

- **When do we know that there wasn't a record in the file that matched the key?**

Searching on sorted Arrays

- **Basic algorithm:**

```
Basic algorithm:
Get the search criterion (key)
Get the first record from the file
While ( (record < key) and (still more records) )
    Get the next record
End_while
If ( record = key ) Then success
    Else there is no match in the file
End_else
```

- **When do we know that there wasn’t a record in the file that matched the key?**

Unsorted vs. Sorted

- **Observation:**
  - The search is faster on an sorted list only when the item being searched for is not in the list.
  - The list has to first be placed in order for the ordered search.
- **Conclusion:**
  - The efficiency of these algorithms is roughly the same.

- So, if we need a faster search, we need a completely different algorithm.

Binary search

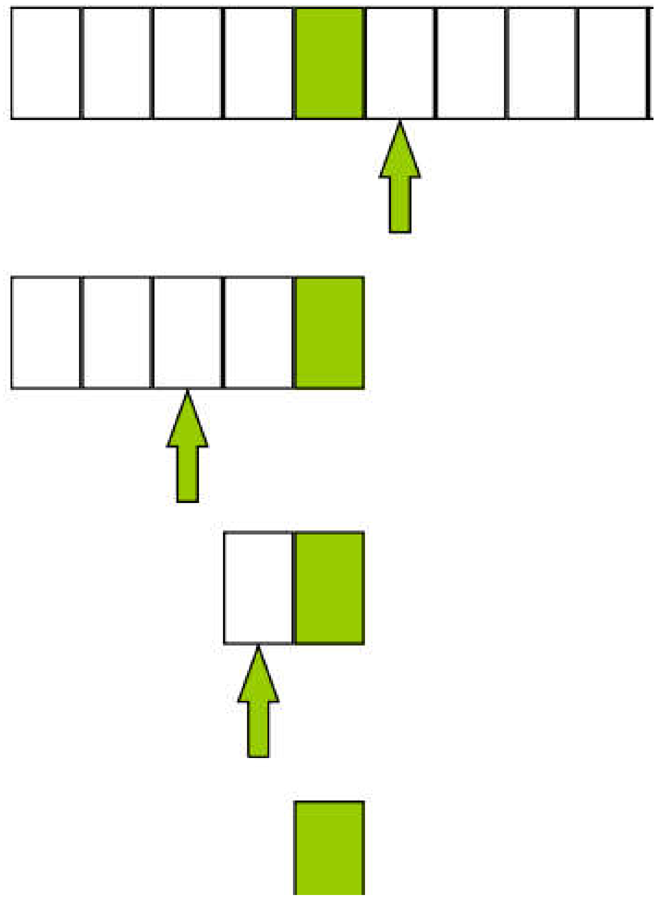
- If we have an ordered list and we know how many things are in the list (i.e., number of records in a file), we can use a different strategy.

- The binary search gets its name because the algorithm continually divides the list into two parts.

How a Binary Search Works

- Always look at the center value.
- Each time you get to discard half of the remaining list.

- Is this fast ?



Binary search: non-recursive Implementation

```
int BinSearch1(int r[ ], int n, int k)
{
   low=1; high=n;
   while (low<=high)
   {
     mid=(low+high)/2;
     if (k<r[mid])  high=mid-1;
     else if (k>r[mid])  low=mid+1;
          else return mid;
   }
   return 0;
```

Binary search: recursive Implementation

```
int BinSearch2(int r[ ], int low, int high, int k)
{
   if (low>high) return 0;
   else {
     mid=(low+high)/2;
     if (k<r[mid])
        return BinSearch2(r, low, mid-1, k);
     else  if (k>r[mid])
             return BinSearch2(r, mid+1, high, k);
           else return mid;
    }
 }
```

How Fast is a Binary Search?

- Worst case: 11 items in the list took 4 tries
- How about the worst case for a list with 32 items ?
  - 1st try - list has 16 items
  - 2nd try - list has 8 items
  - 3rd try - list has 4 items
  - 4th try - list has 2 items
  - 5th try - list has 1 item
- What's the Pattern?

A Very Fast Algorithm!

- How long (worst case) will it take to find an item in a list 30,000 items long?

$2^{10} = 1024$	$2^{13} = 8192$
$2^{11} = 2048$	$2^{14} = 16384$
$2^{12} = 4096$	$2^{15} = 32768$

So, it will take only 15 tries!

Lg n Efficiency

- We say that the binary search algorithm runs in $\log_2 n$ time. (Also written as lg n)
- Lg n means the log to the base 2 of some value of n.

There are no algorithms that run faster than lg n time.

Sorting

- So, the binary search is a very fast search algorithm.
- But, the list has to be sorted before we can search it with binary search.

- To be really efficient, we also need a fast sort algorithm.

Summary

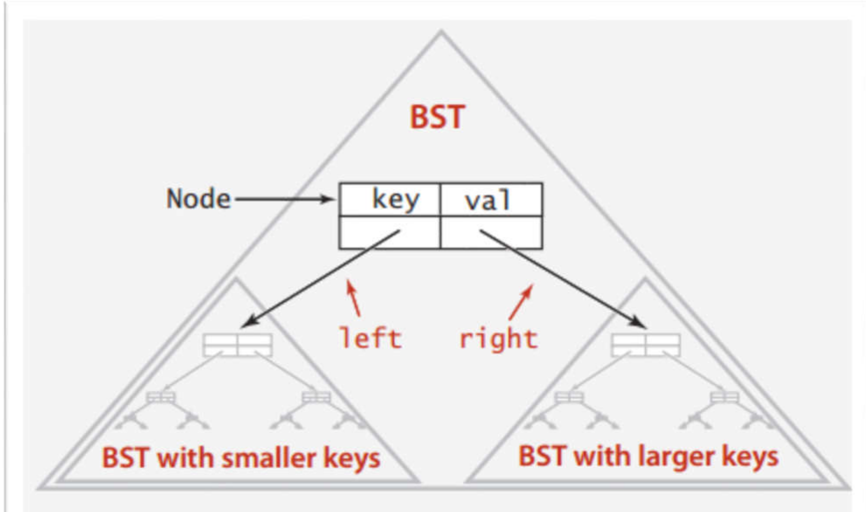
ST implementation	worst-case cost (after N inserts)		average case (after N random inserts)		ordered iteration?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N / 2	N	no	`equals()`
binary search (ordered array)	log N	N	log N	N / 2	yes	`compareTo()`

Challenge. Efficient implementations of both search and insert.

Binary Search Trees

A Taxonomy of Trees

- General Trees –any number of children / node
- Binary Trees –max 2 children / node
  - Binary Search Trees

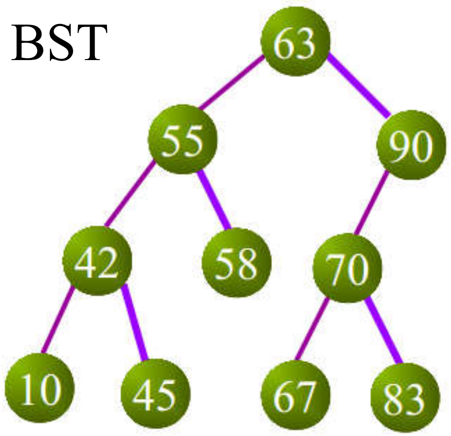


Binary search tree

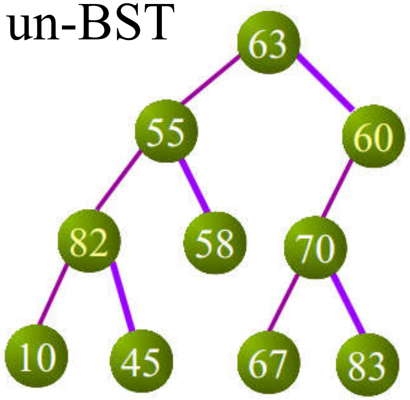
Binary Search Trees

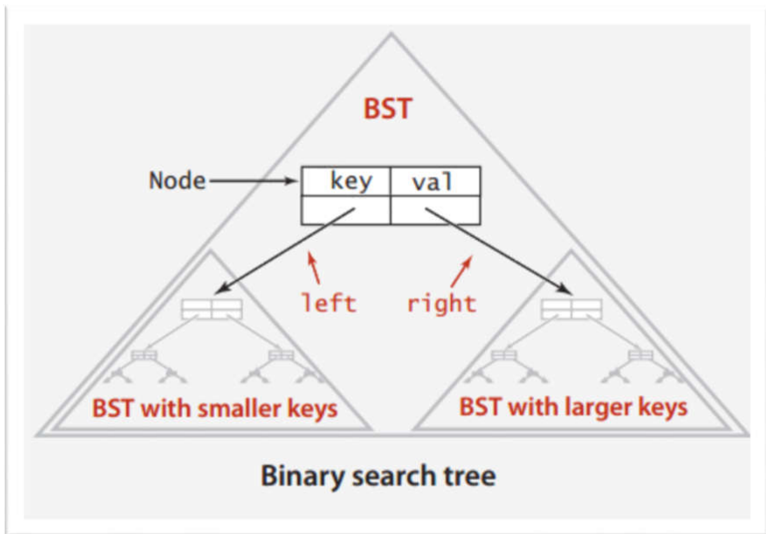
- Definition of Binary search tree (BST)
  - Every element has a unique key.
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
  - The left and right subtrees are also binary search trees.

BST



un-BST





Binary Search Trees

- Binary Search Trees (BST) are a type of Binary Trees with a special organization of data.
- Organization Rule for BST
  - the values in all nodes in the left subtree of a node are less than the node value
  - the values in all nodes in the right subtree of a node are greater than the node values
- This data organization leads to **O(log n) complexity** for searches, insertions and deletions in certain types of the BST (balanced trees).
  - O(h) in general

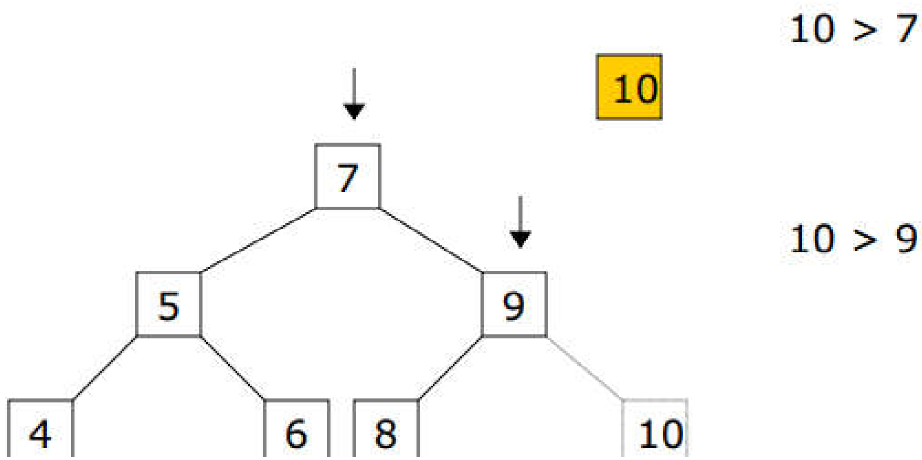
BST Operations: Insertion

method insert(key)

- places a new item near the frontier of the BST while retaining its organization of data:
  - to locate the insertion point is based on comparisons of the new item and values of nodes in the BST
  - starting at the root, it probes down the tree
  - till it finds a node whose left or right pointer is empty and is a logical place for the new value
- Elements in nodes must be comparable!

Insertion in BST - Example

- Case 1:The Tree is Empty
  – Set the root to a new node containing the item •
- Case 2:The Tree is Not Empty
  – Call a recursive helper method to insert the item



Insertion in BST - Pseudocode

```
if  tree is empty
    create a root node with the new key
else
    compare key with the top node
    if key =  node key
        replace the node with the new value
    else if  key >  node key
        compare key with the right subtree:
          if  subtree is empty create a leaf node
          else add key  in right subtree
     else  key <  node key
        compare key with the left subtree:
          if the subtree is empty create a leaf node
          else add key to the left subtree
```

BST Operations: Search

method search(key)

- implements the binary search based on comparison of the items in the tree
- the items in the BST must be comparable (e.g integers, string, etc.)
  - The search starts at the root.
  - It probes down, comparing the values in each node with the target, till
  - it finds the first item equal to the target.
  - Returns this item or null if there is none.

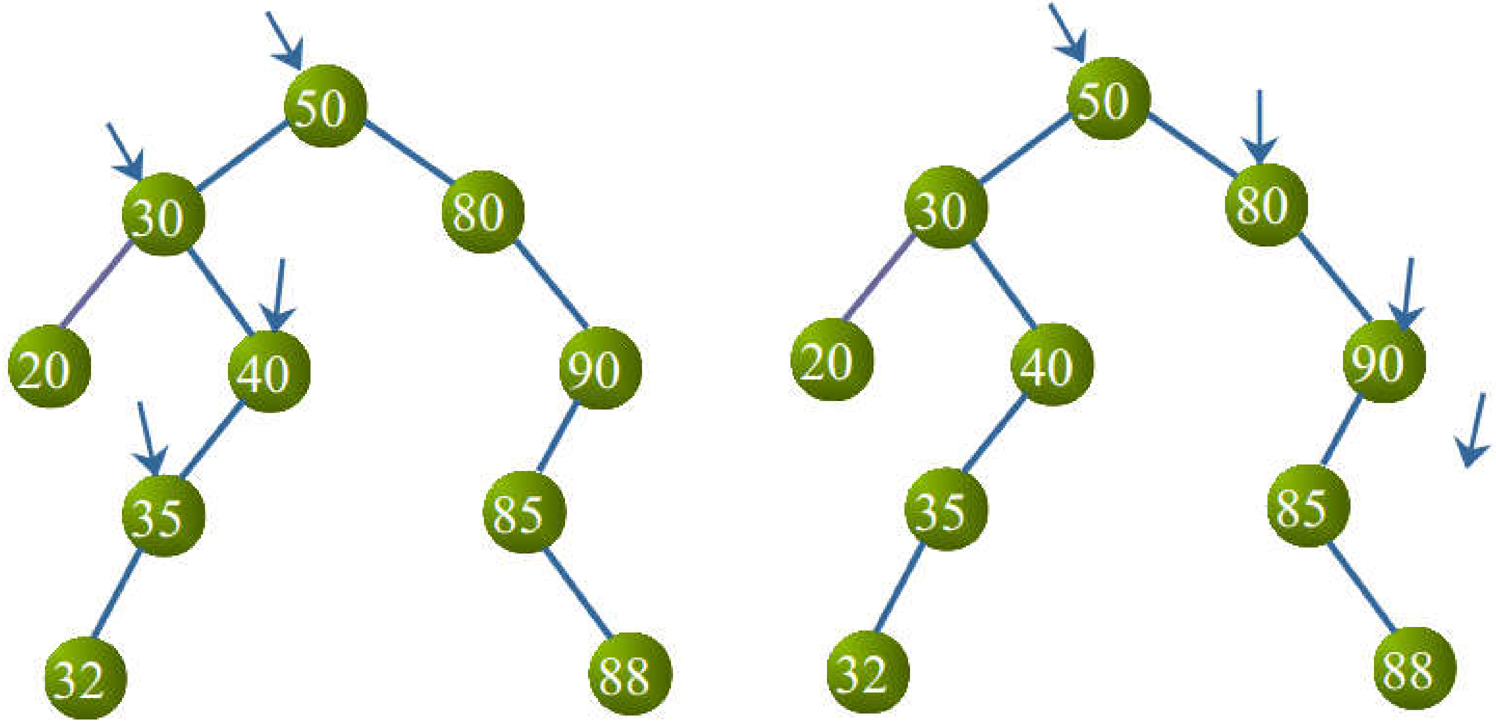
Search: If less, go left; if greater, go right; if equal, search hit.

Search in BST - Pseudocode

```
if the tree is empty
        return NULL

else if  the item in the node equals the target
        return the node value

else if  the item in the node is greater than the
target return the result of searching the left
subtree

else if  the item in the node is smaller than the
target return the result of searching the right
subtree
```

Search in BST - Example

- Search for 35， 95



BST Operations: Removal

Removes a specified item from the BST and

Adjusts the tree

- uses a binary search to locate the target item:
  - starting at the root it probes down the tree till
  - it finds the target or
  - reaches a leaf node (target not in the tree)

- removal of a node must not leave a ‘gap’ in the tree,

Removal in BST - Pseudocode

method remove (key)

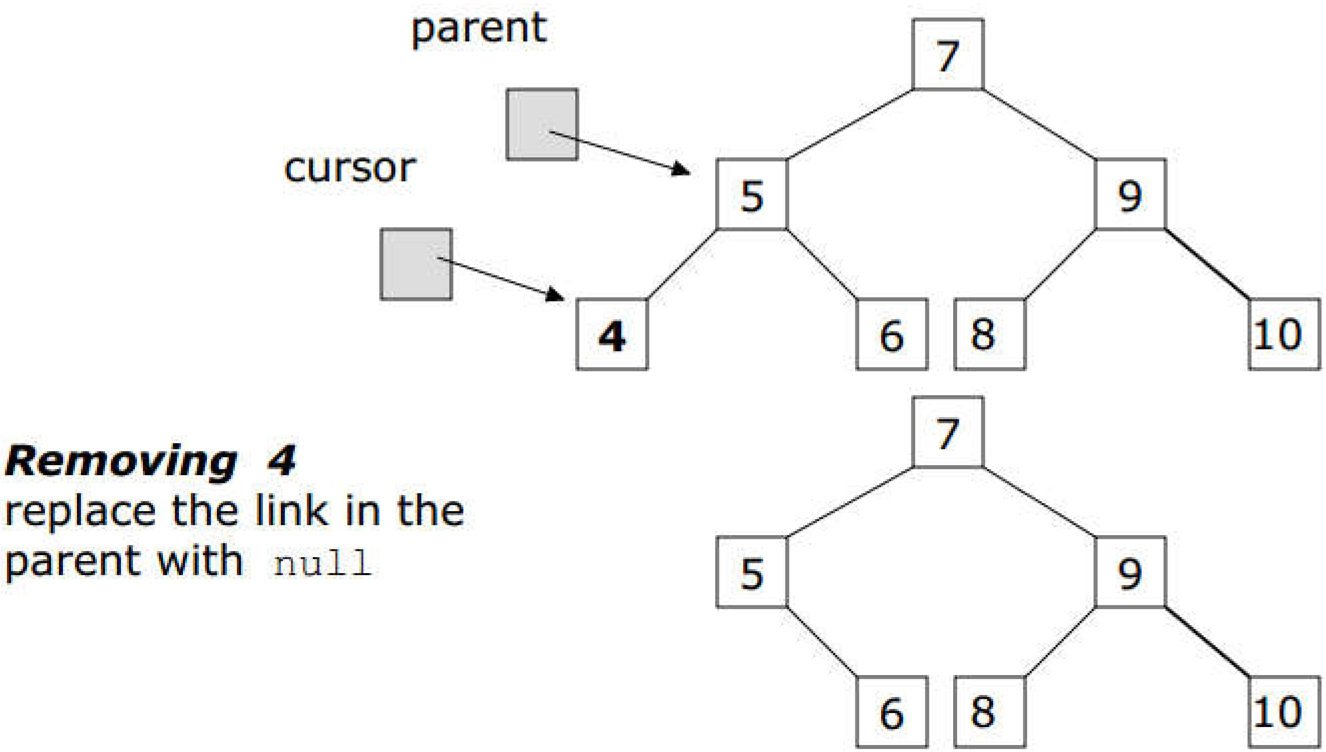
- I if the tree is empty return false
- II Attempt to locate the node containing the target
  - using the binary search algorithm
  - if the target is not found return false
  - else the target is found, so remove its node:
    - **Case 1**: if the node has 2 empty subtrees
      - replace the link in the parent with null
    - **Case 2:** if the node has a left and a right subtree
      - replace the node's value with the <u>max value in the left subtree</u>
      - delete the max node in the left subtree

Removal in BST - Pseudocode

- **Case 3**: if the node has no left child
  - link the parent of the node to the right (non-empty) subtree
- **Case 4:** if the node has no right child
  - link the parent of the target to the left (non-empty) subtree

Removal in BST: Example

- **Case 1**: removing a node with 2 EMPTY SUBTREES

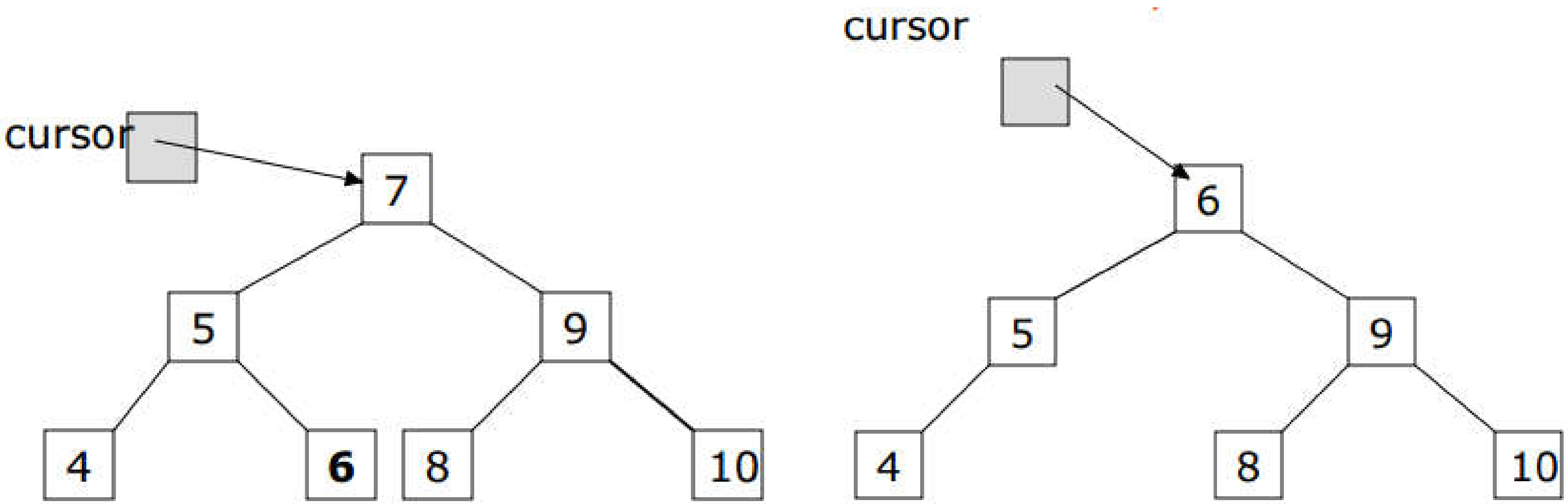


Removal in BST: Example

- **Case 2**:removing a node with 2 SUBTREES
  - replace the node's value with the max value in the left subtree
  - delete the max node in the left subtree

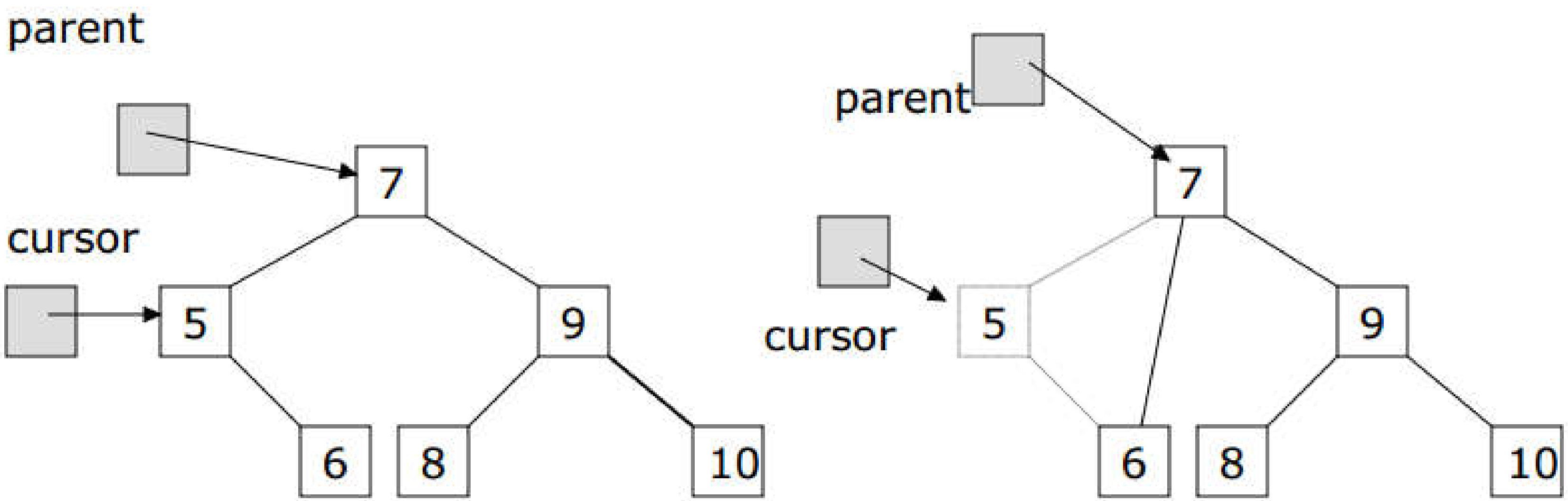
What other element can be used as replacement?

Removing 7



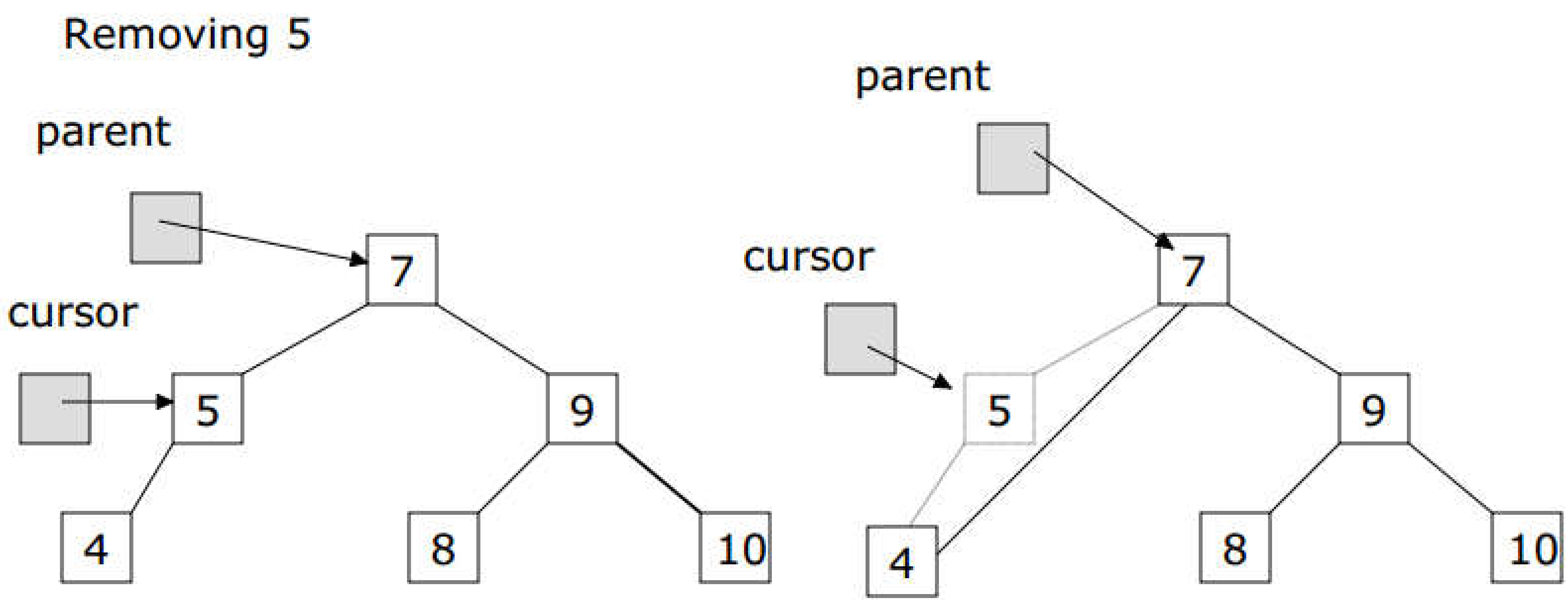
Removal in BST: Example

- **Case 3**:removing a node with 1 EMPTY SUBTREE
  – the node has no left child:
  – link the parent of the node to the right (non-empty) subtree



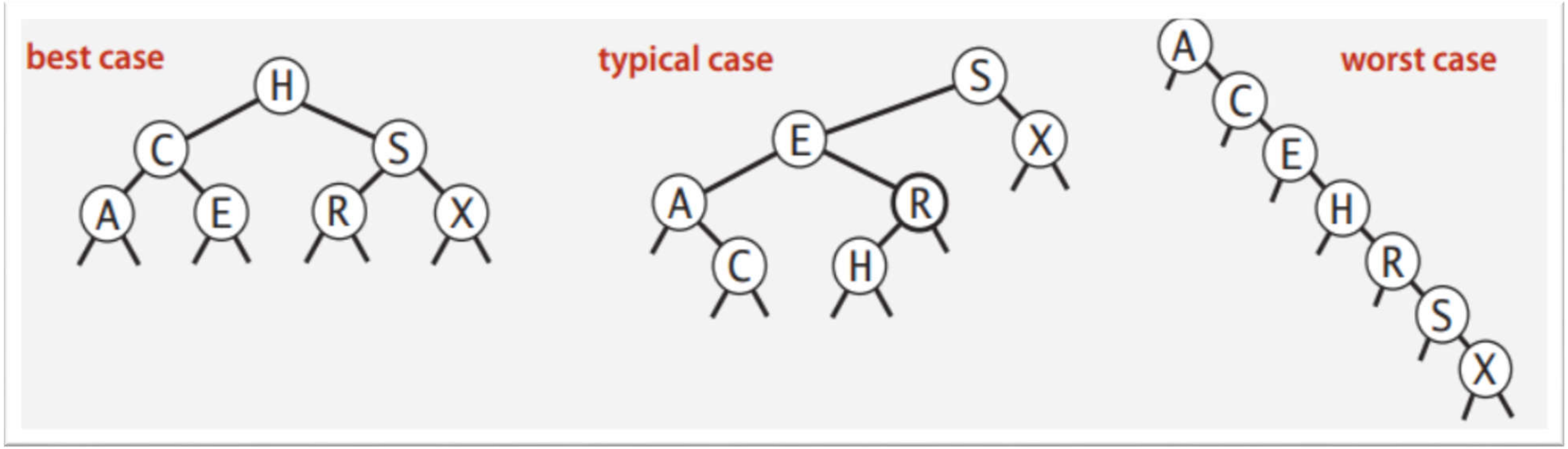
Removal in BST: Example

- **Case 4**:removing a node with 1 EMPTY SUBTREE
  - the node has no right child:
  - link the parent of the node to the left (non-empty) subtree



Tree shape

- Number of compares for search/insert is equal to 1 + depth of node.



Tree shape depends on order of insertion.

Analysis of BST Operations

- The complexity of operations **get, insert** and **remove** in BST is O(h) , where h is the height.
  - O(log n) when the tree is balanced.
  - The updating operations cause the tree to become unbalanced.
  - The tree can degenerate to a linear shape and the operations will become O (n)

Summary

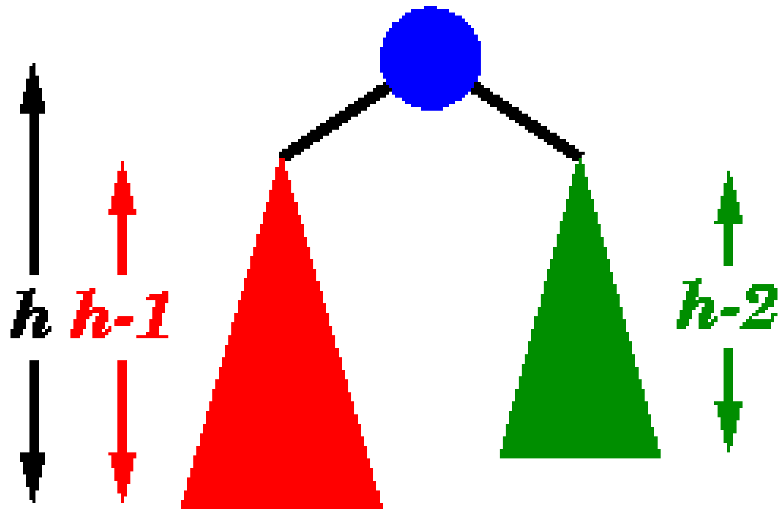
implementation	guarantee		average case		ordered ops?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N/2	N	no	`equals()`
binary search (ordered array)	lg N	N	lg N	N/2	yes	`compareTo()`
BST	N	N	1.39 lg N	1.39 lg N	next	`compareTo()`

Q. Can we do better?
A. Yes, but with different access to the data.

AVL tree

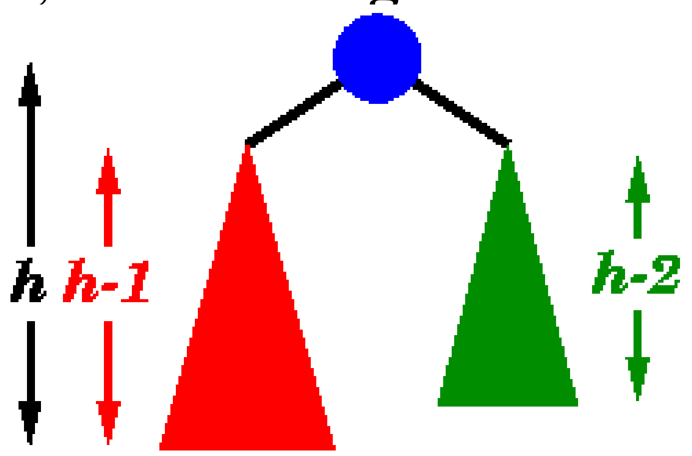
Adelson-Velskii & Landis trees, 1962

(Height-balanced trees)



AVL tree

- Is a binary search tree
- Has an additional ***height constraint***:
  – For each node x in the tree, Height(x.left) differs from Height(x.right) by at most 1
- I promise:
  – If you satisfy the ***height constraint***, then the **height of the tree is O(lg n)**.
  – (Proof is easy, but no time!)

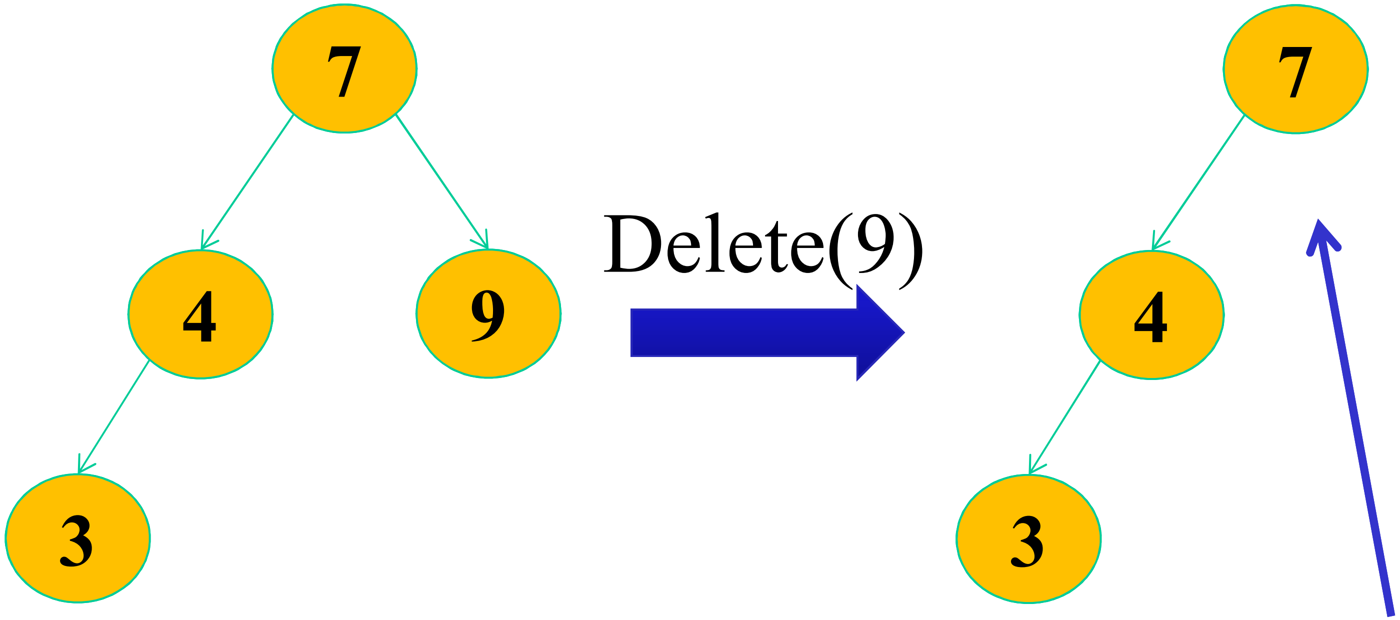


AVL tree

- To be an AVL tree, must **always:**
  - (1) Be a ***binary search tree***
  - (2) Satisfy the ***height constraint***

- Suppose we start with an AVL tree, then insert/delete as if we're in a regular BST.
- Will the tree be an AVL tree after the insert/delete?
  - (1) It will still be a BST… that's one part.
  - (2) Will it satisfy the ***height constraint***?

BST Delete breaks an AVL tree



h(left) > h(right)+1

so **NOT** an AVL tree!

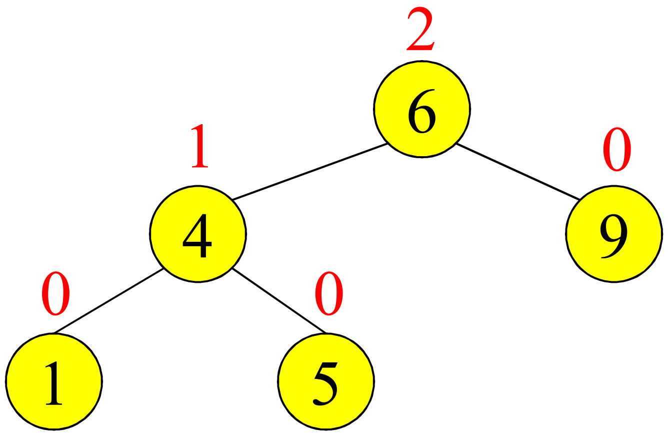
Balance factors

- To check the **balance constraint**, we have to know the ***height h*** of each node
- Or do we?
- In fact, we can store ***balance factors*** instead.
- The balance factor bf(x) = h(x.left) – h(x.right)
  - bf(x) values -1, 0, and 1 are allowed.
  - If bf(x) < -1 or bf(x) > 1 then tree is **NOT AVL**

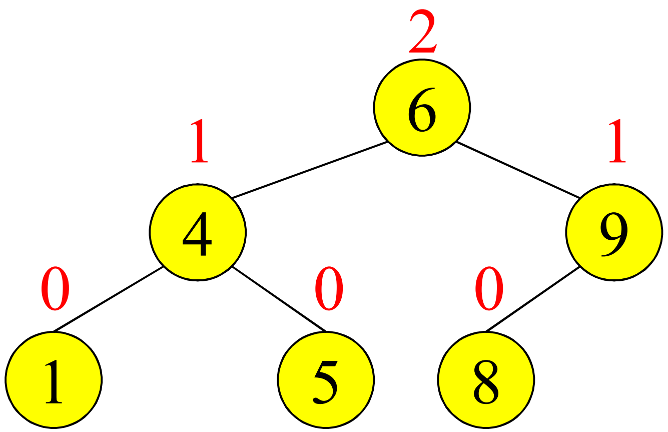
Node Heights

Tree A (AVL)

height=2   BF=1-0=1



Tree B (AVL)



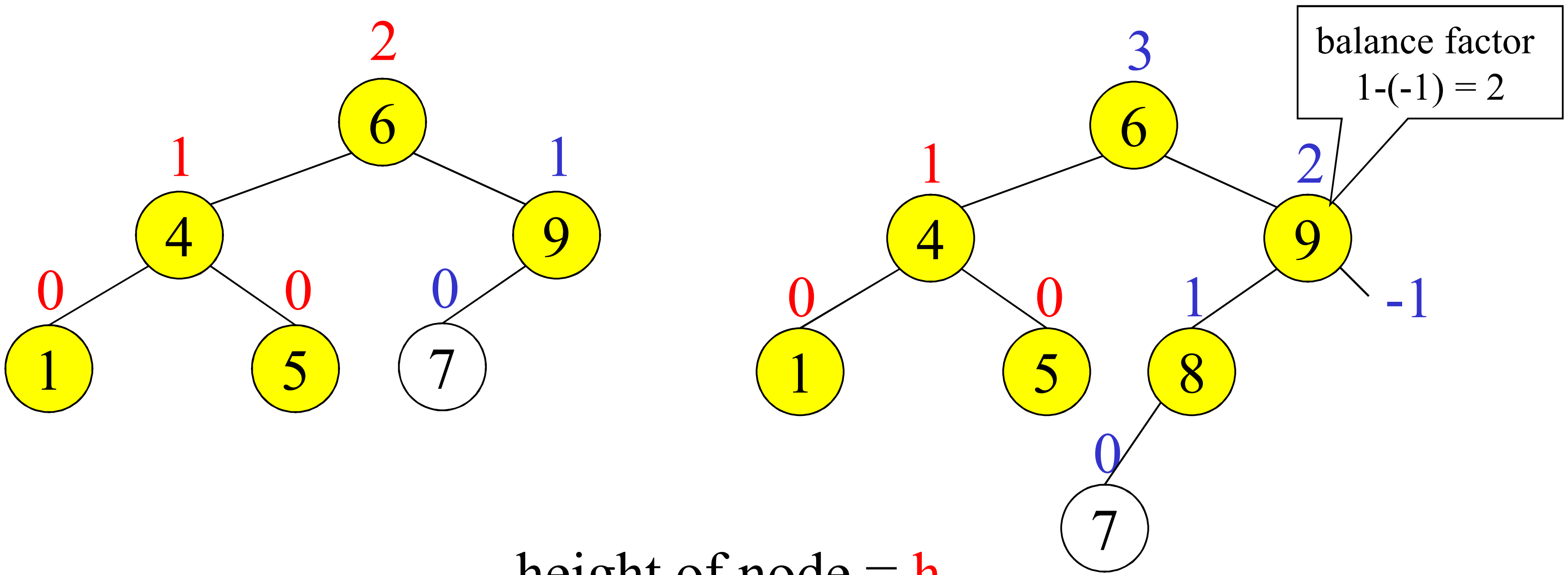
height of node = h

balance factor = $h_{left}-h_{right}$

empty height = -1

Node Heights after Insert 7

Tree A (AVL) | Tree B (not AVL)



height of node = h

balance factor = $h_{left}-h_{right}$

empty height = -1

How to maintain the balance?

Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :

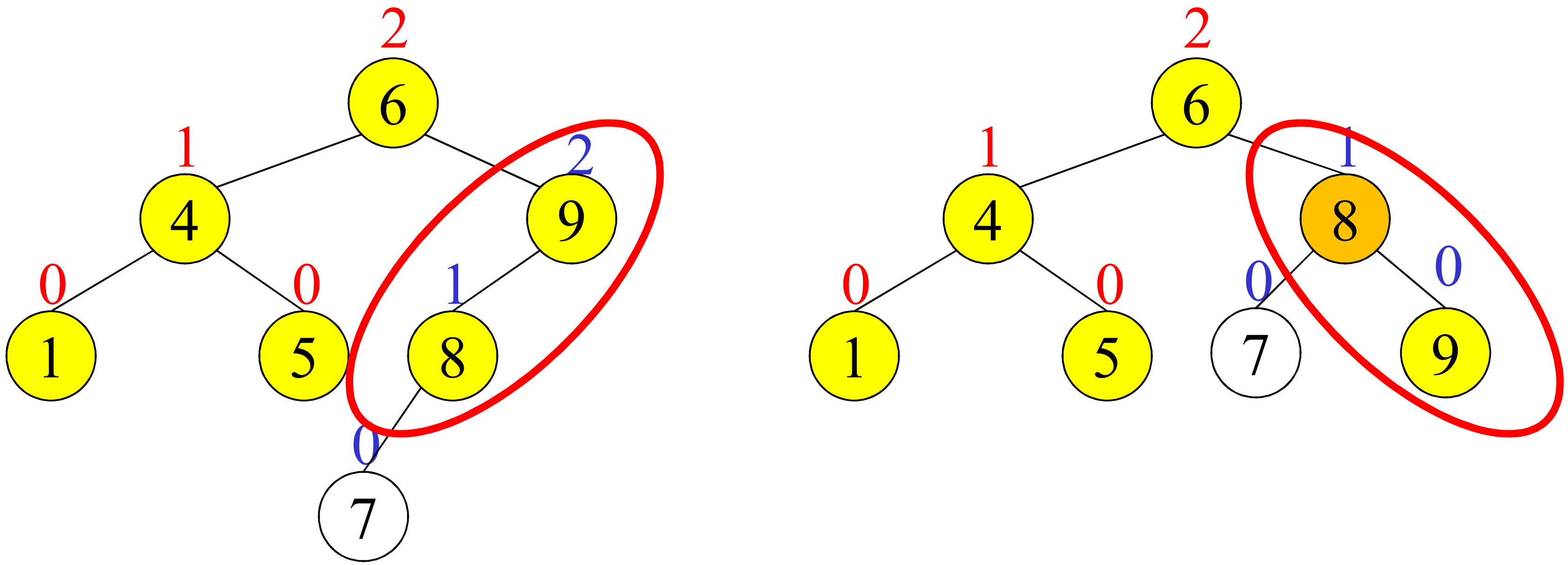
1. Insertion into left subtree of left child of α. (LL)
2. Insertion into right subtree of right child of α. (RR)

Inside Cases (require double rotation) :

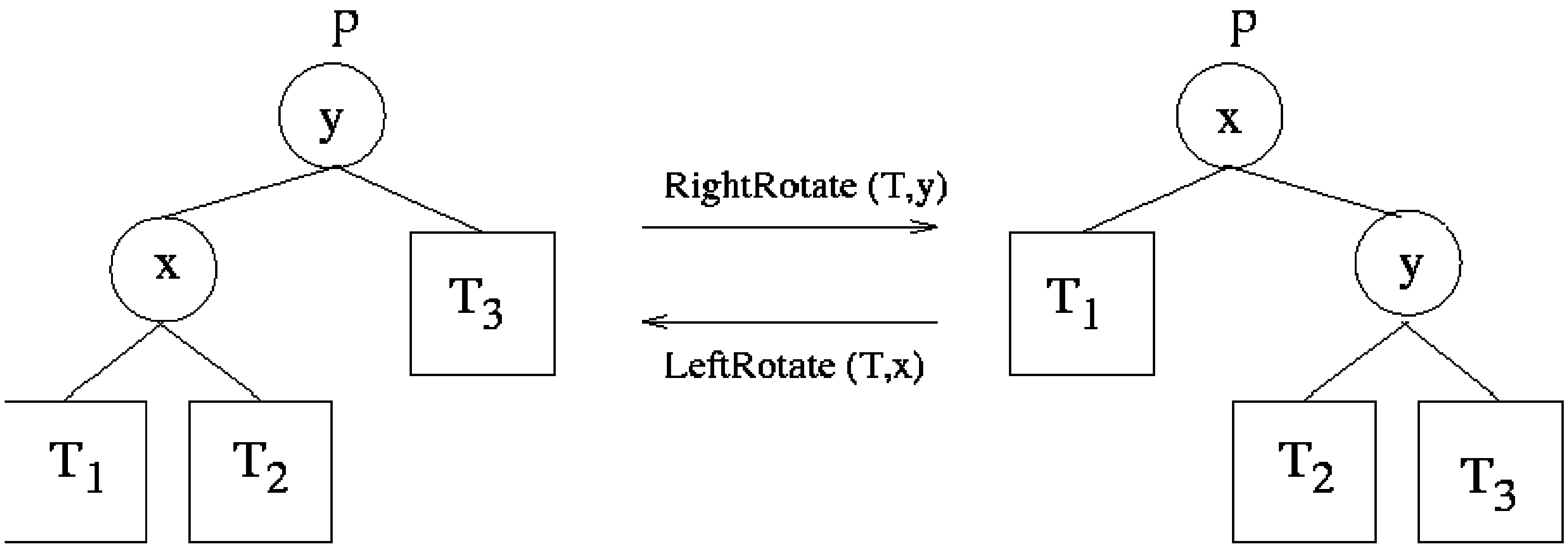
3. Insertion into right subtree of left child of α. (LR)
4. Insertion into left subtree of right child of α. (RL)

The rebalancing is performed through four separate rotation algorithms.

Single Rotation in an AVL Tree

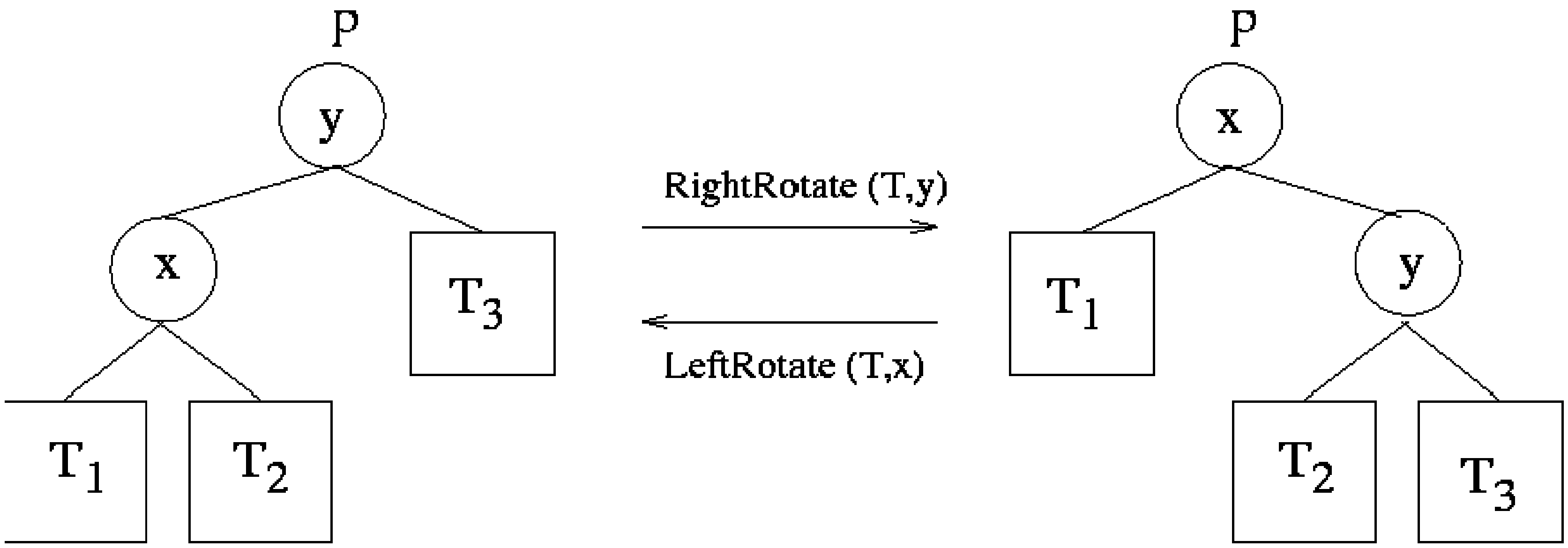


How To Do Single Rotations



- The names describe which way the node moves.
  - For example, a left rotation moves the node down and left.
- Nodes always move down when rotated.

How To Do Single Rotations

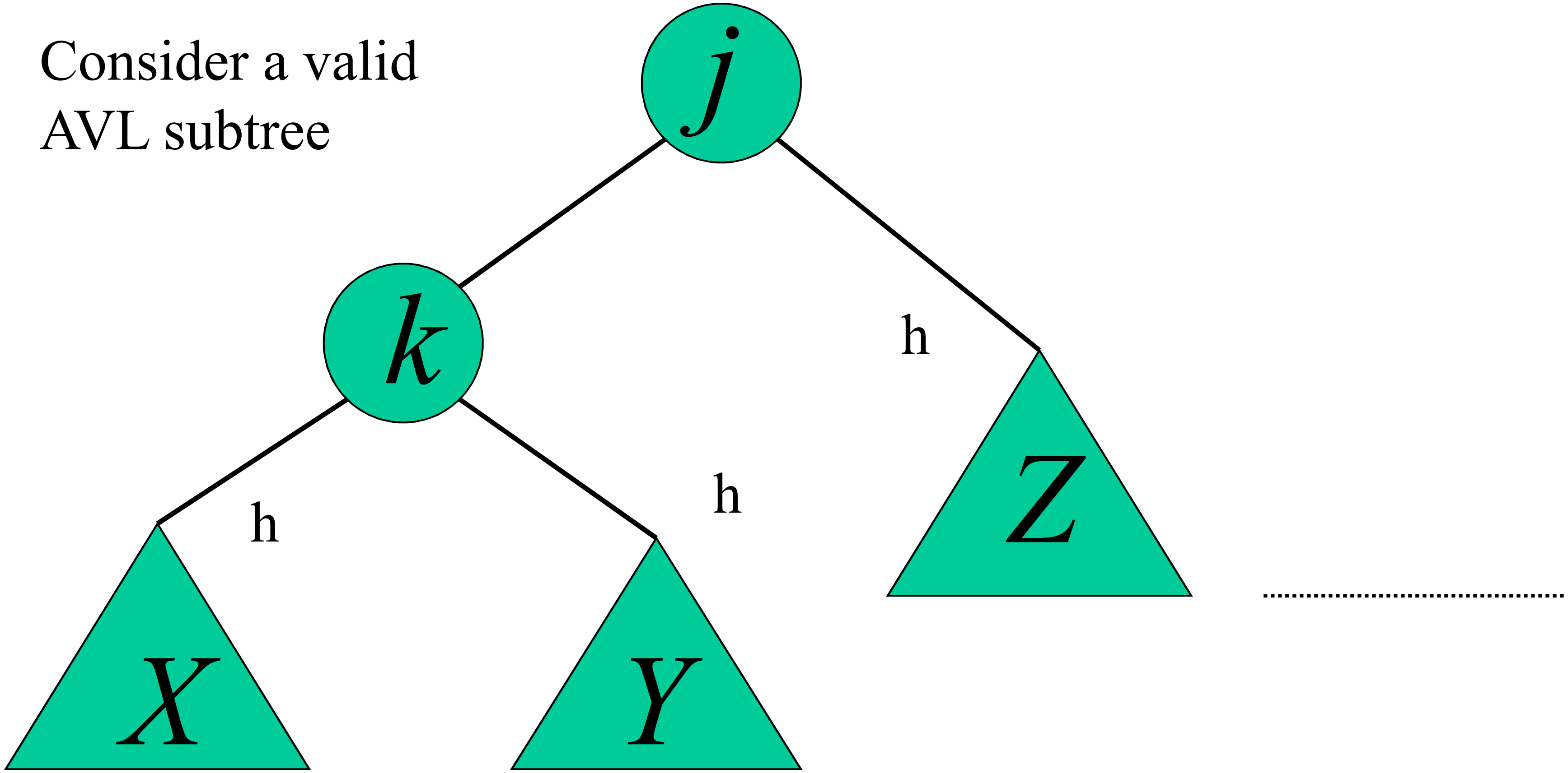


A Left Rotation via Pointers:

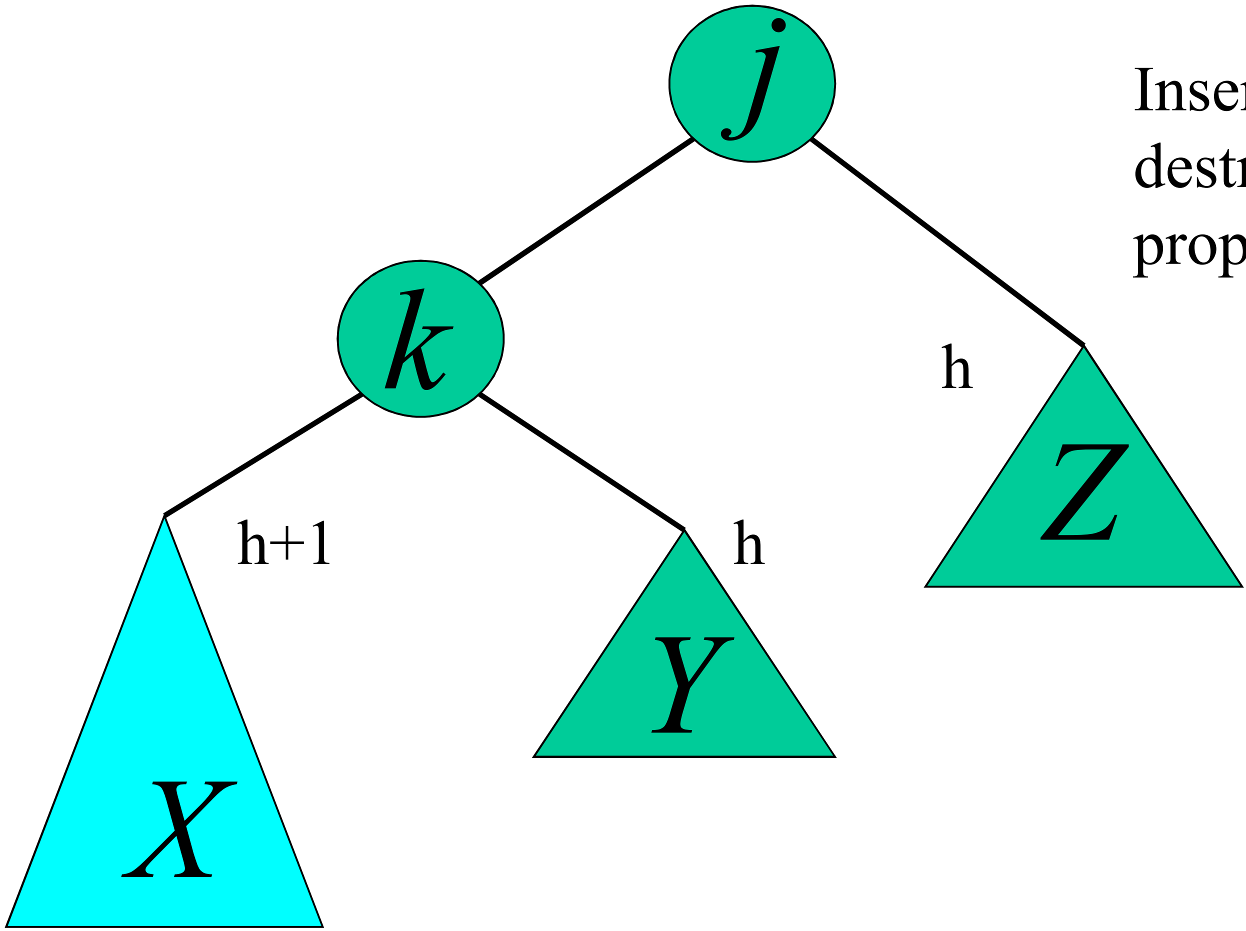
```
temp=p->right ;
p->right=temp->left ;
temp->left=p ;
p=temp ;
```

AVL Insertion: Outside Case

Consider a valid AVL subtree

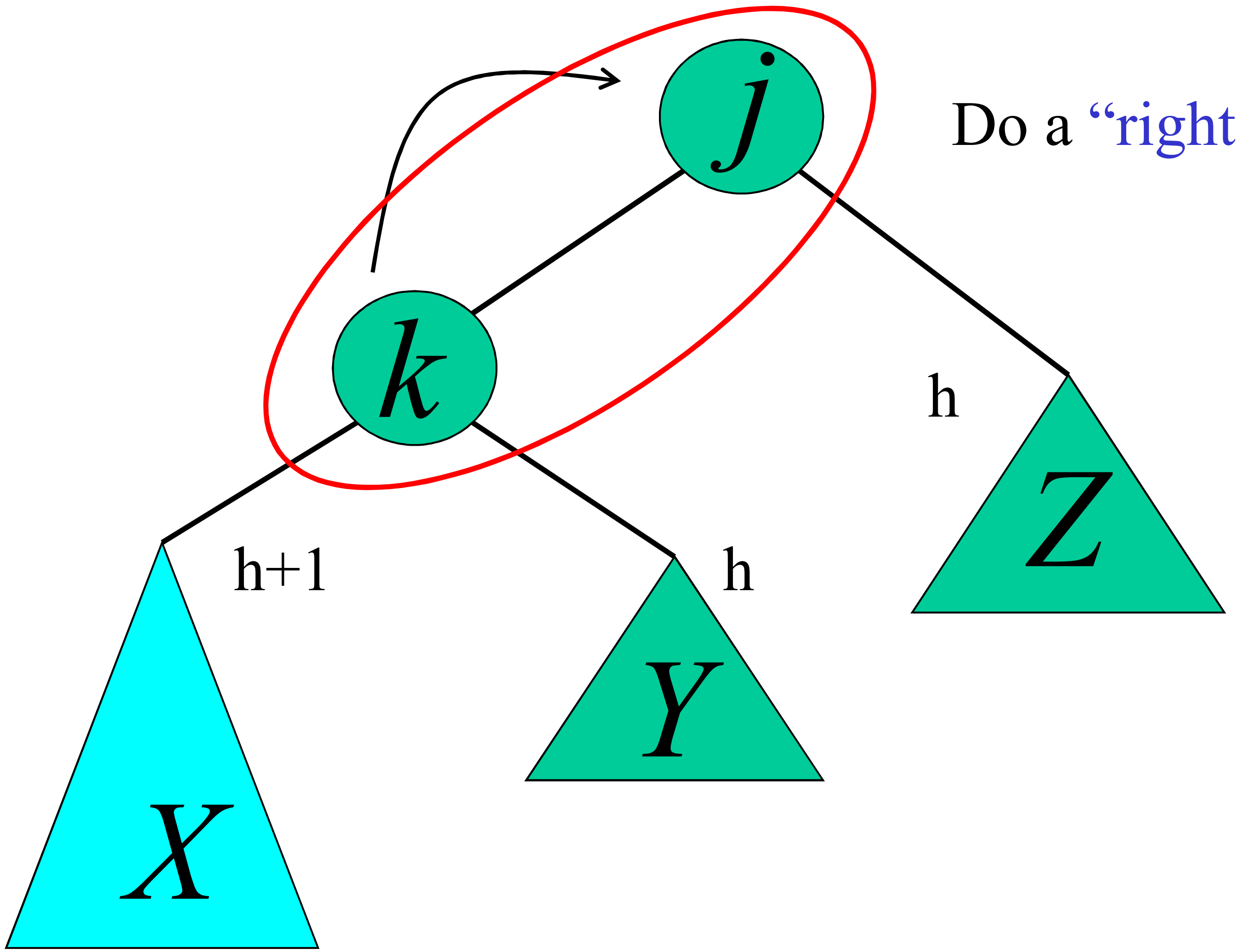


AVL Insertion: Outside Case



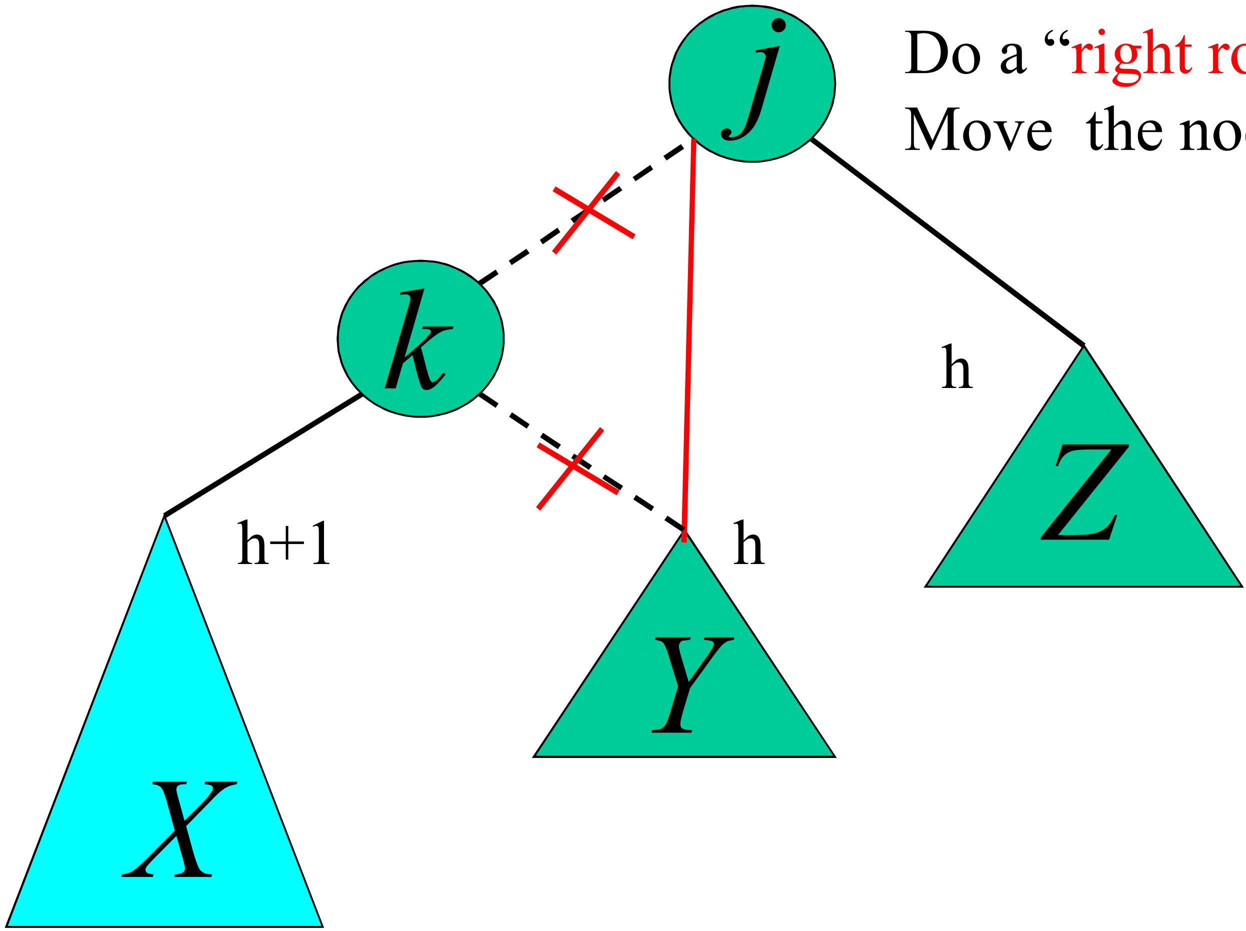
Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case



Do a "right rotation"

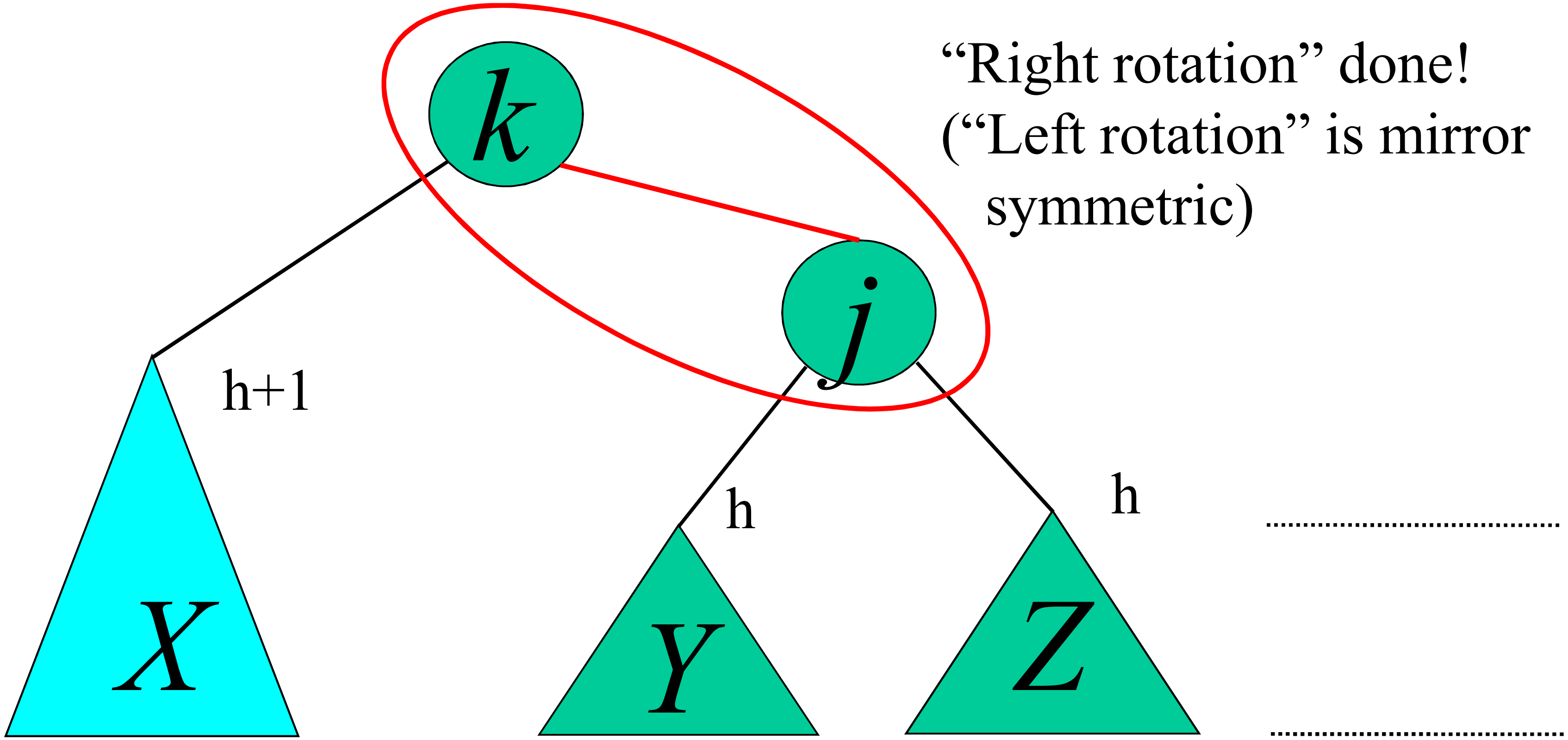
Single right rotation



Do a "right rotation" :
Move the node down and right.

Nodes always move down when rotated

Outside Case Completed

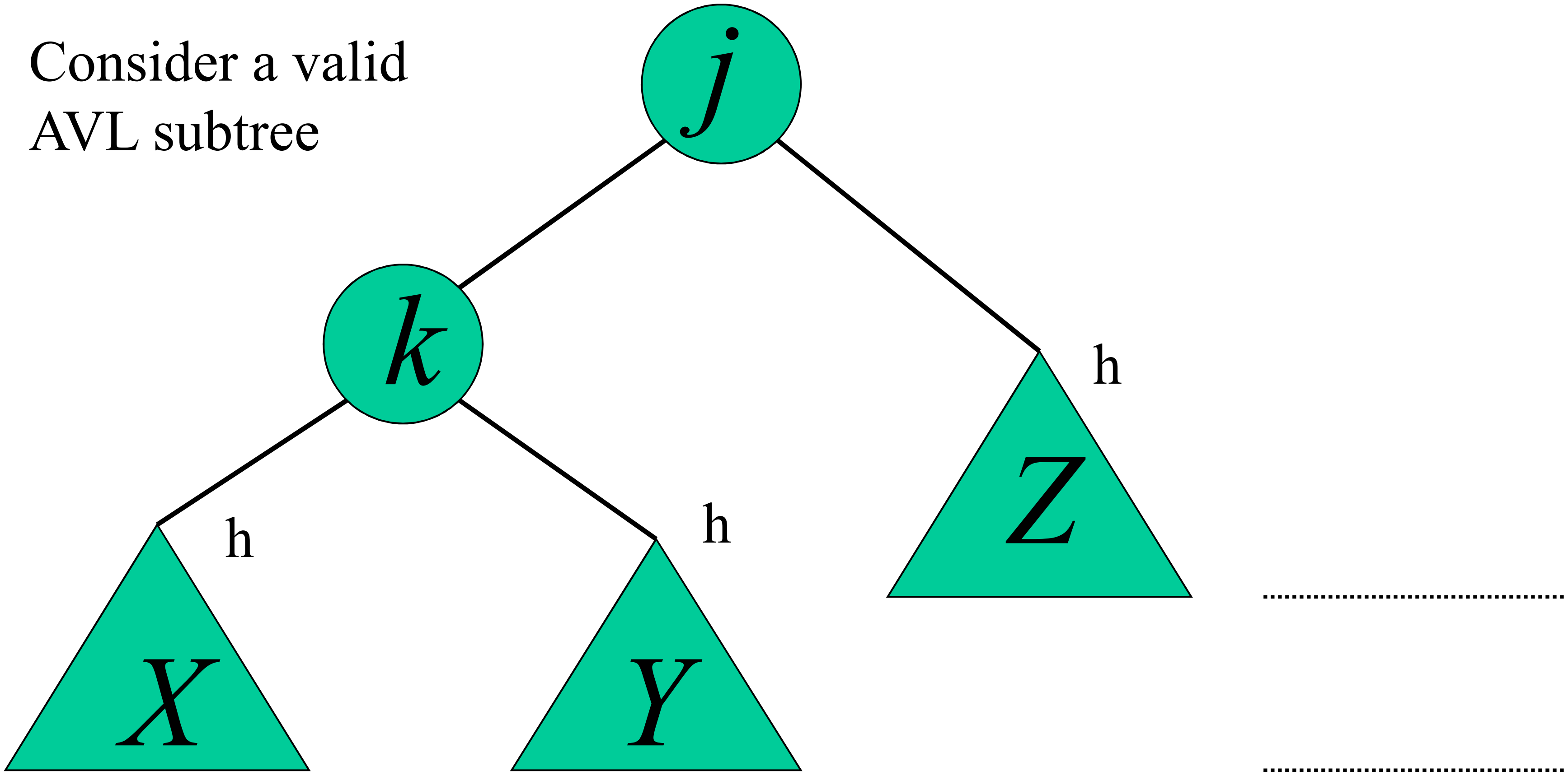


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

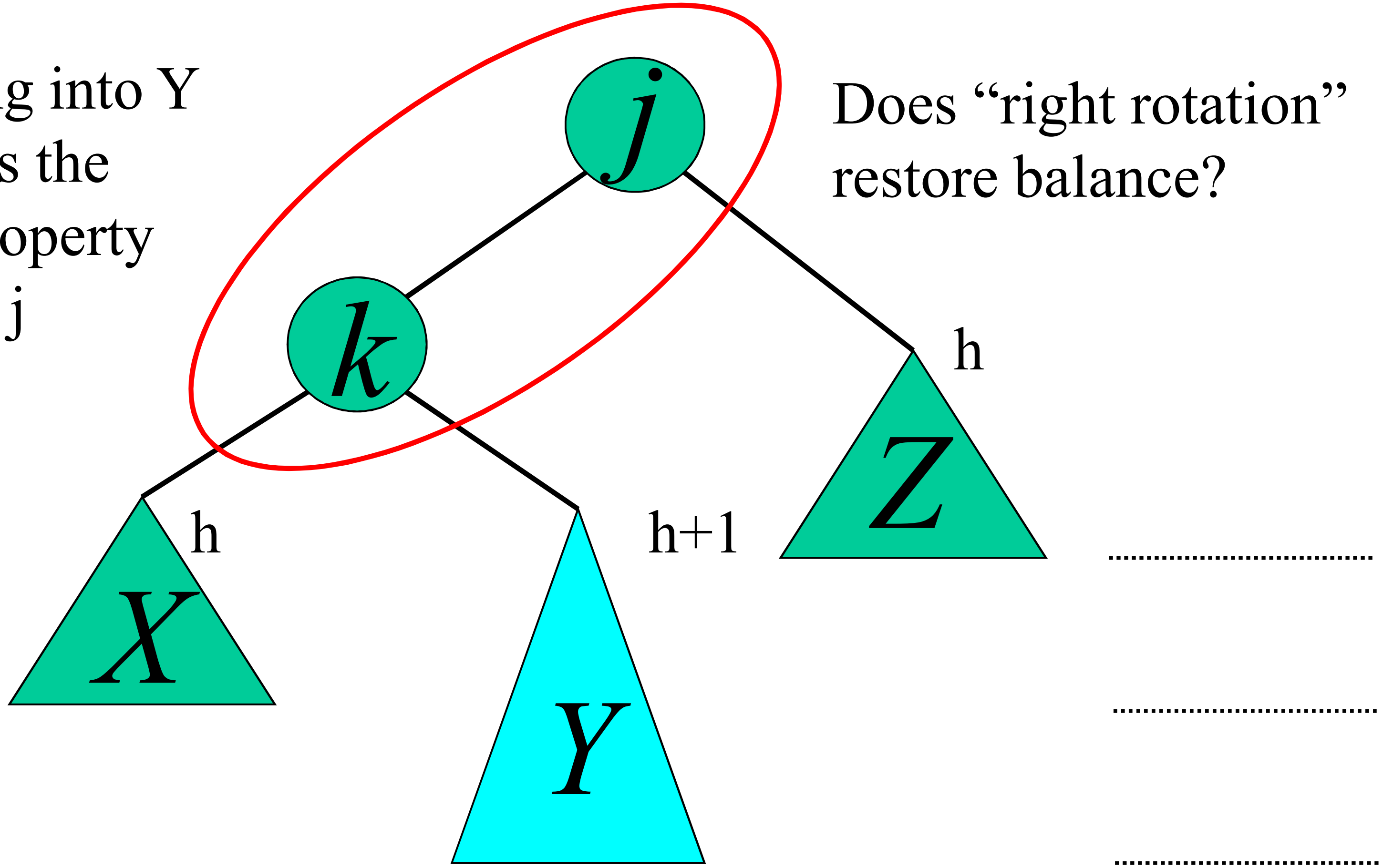
AVL Insertion: Inside Case

Consider a valid
AVL subtree



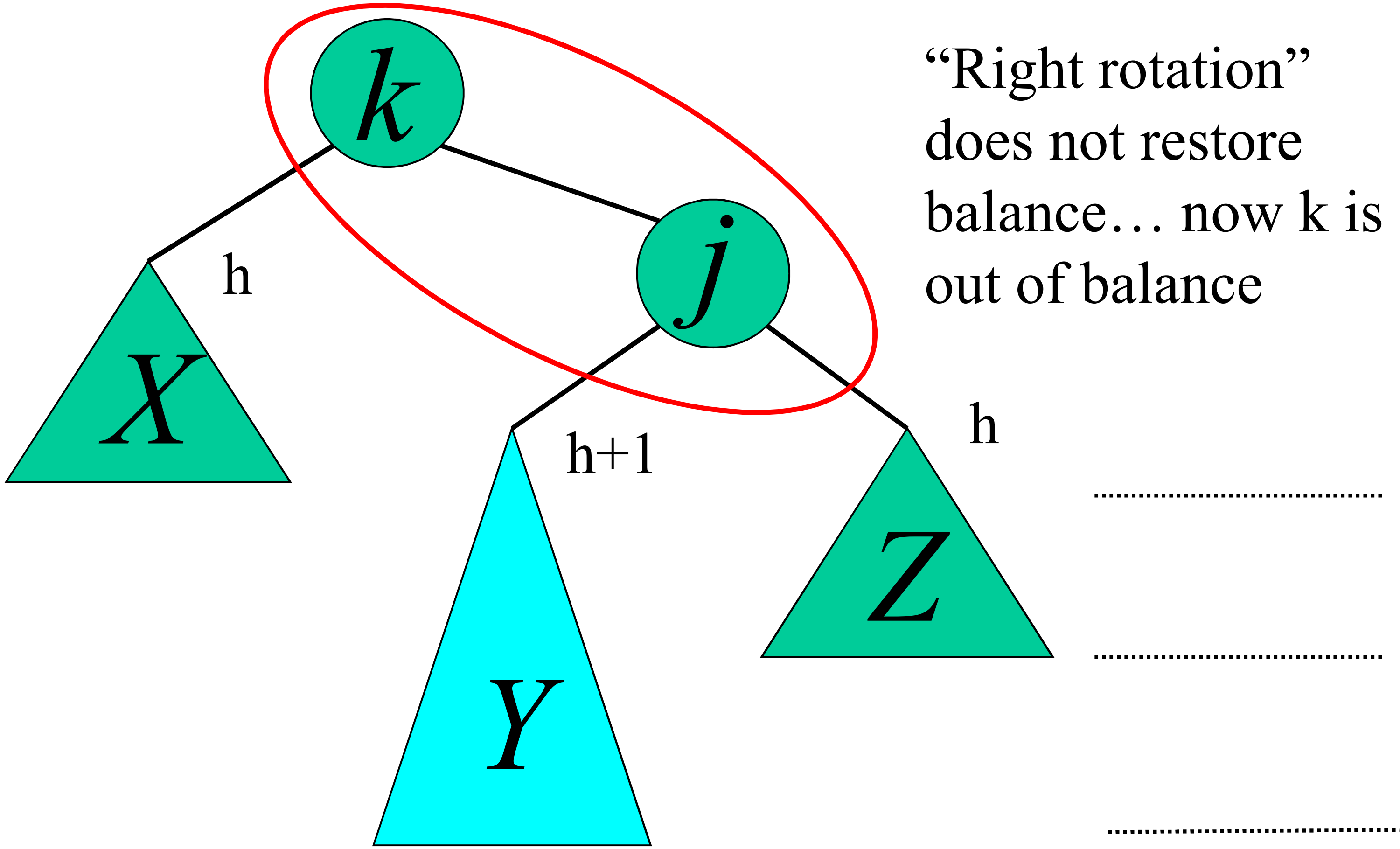
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j



Does "right rotation" restore balance?

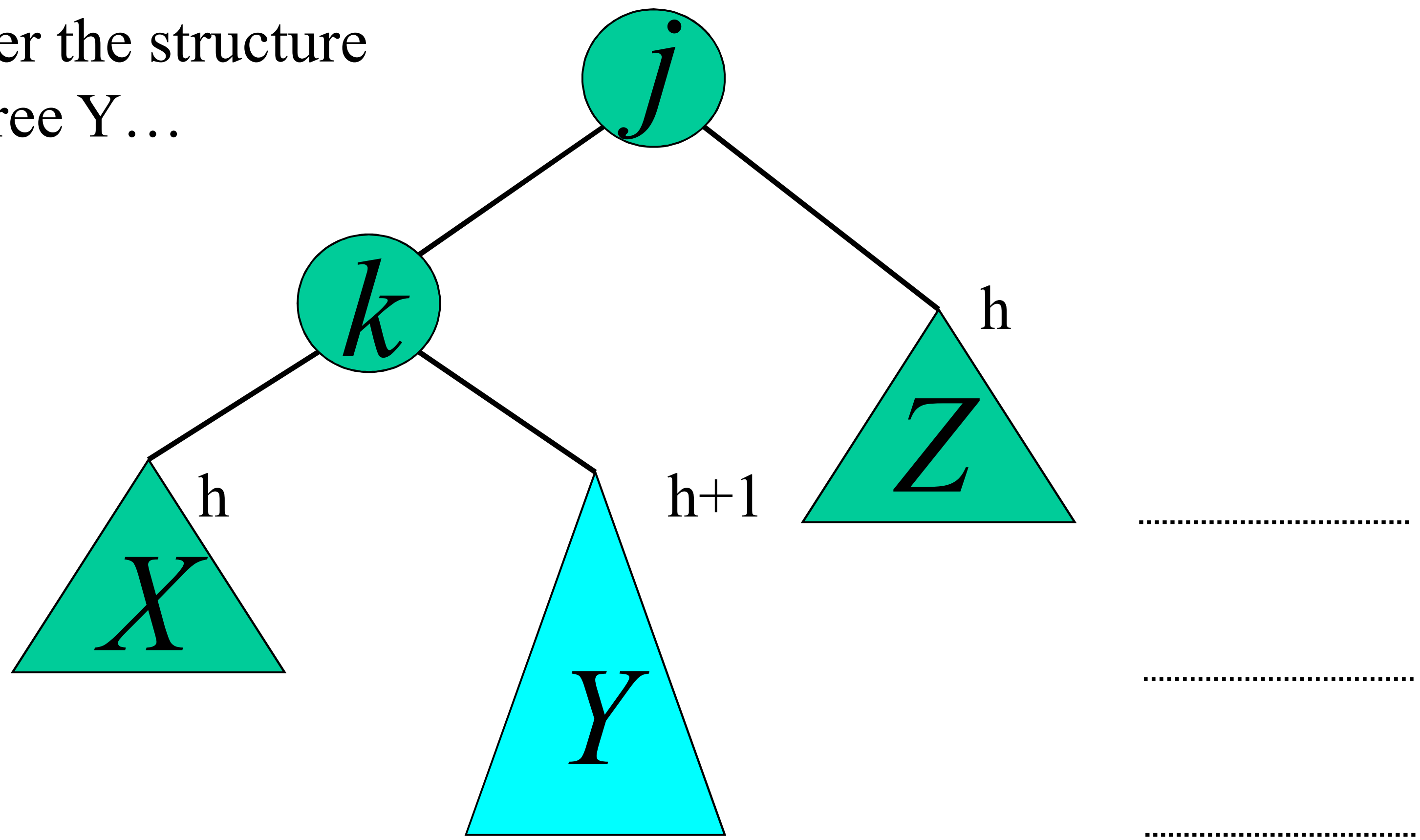
AVL Insertion: Inside Case



"Right rotation" does not restore balance… now k is out of balance

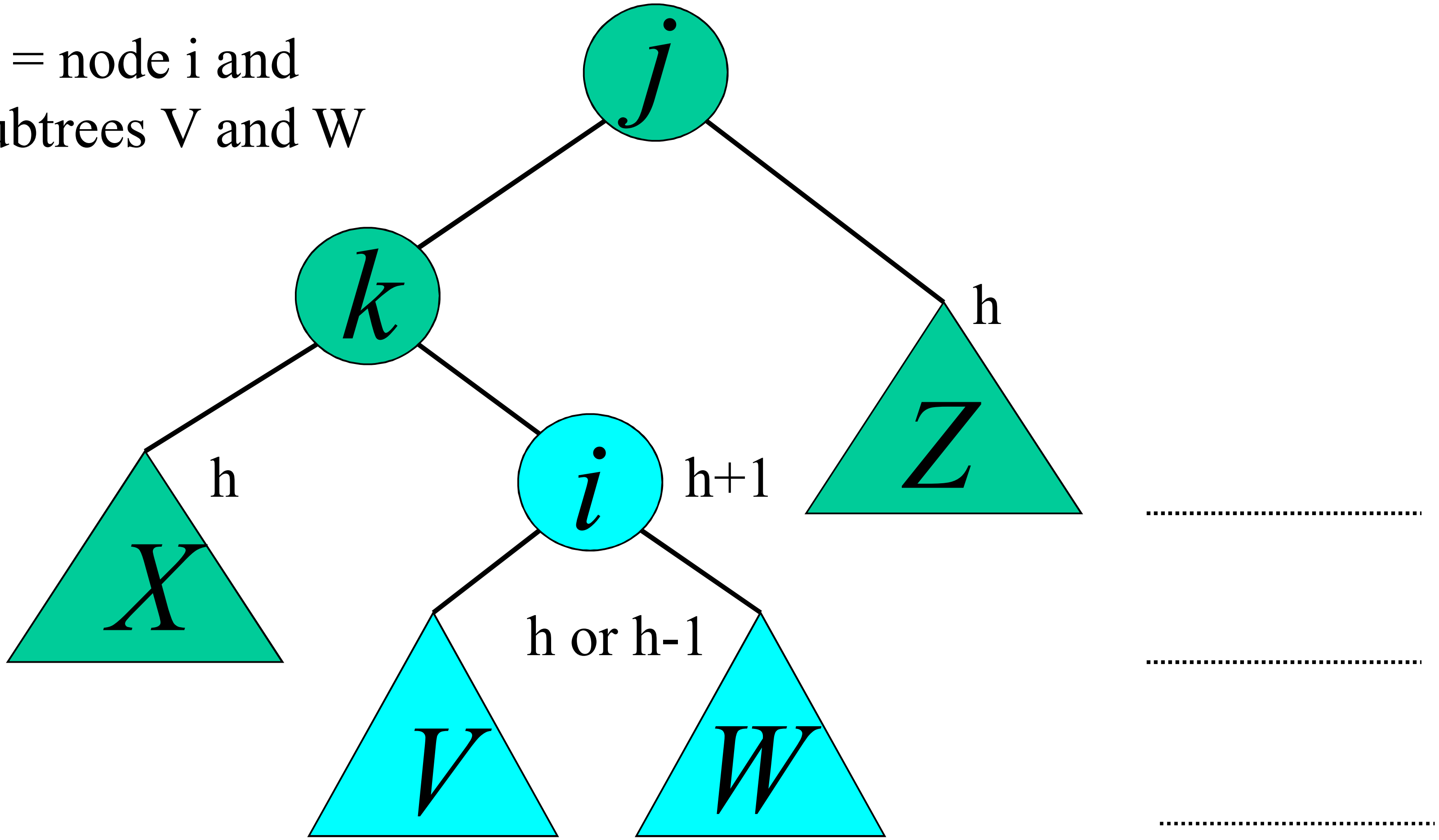
AVL Insertion: Inside Case

Consider the structure of subtree Y…

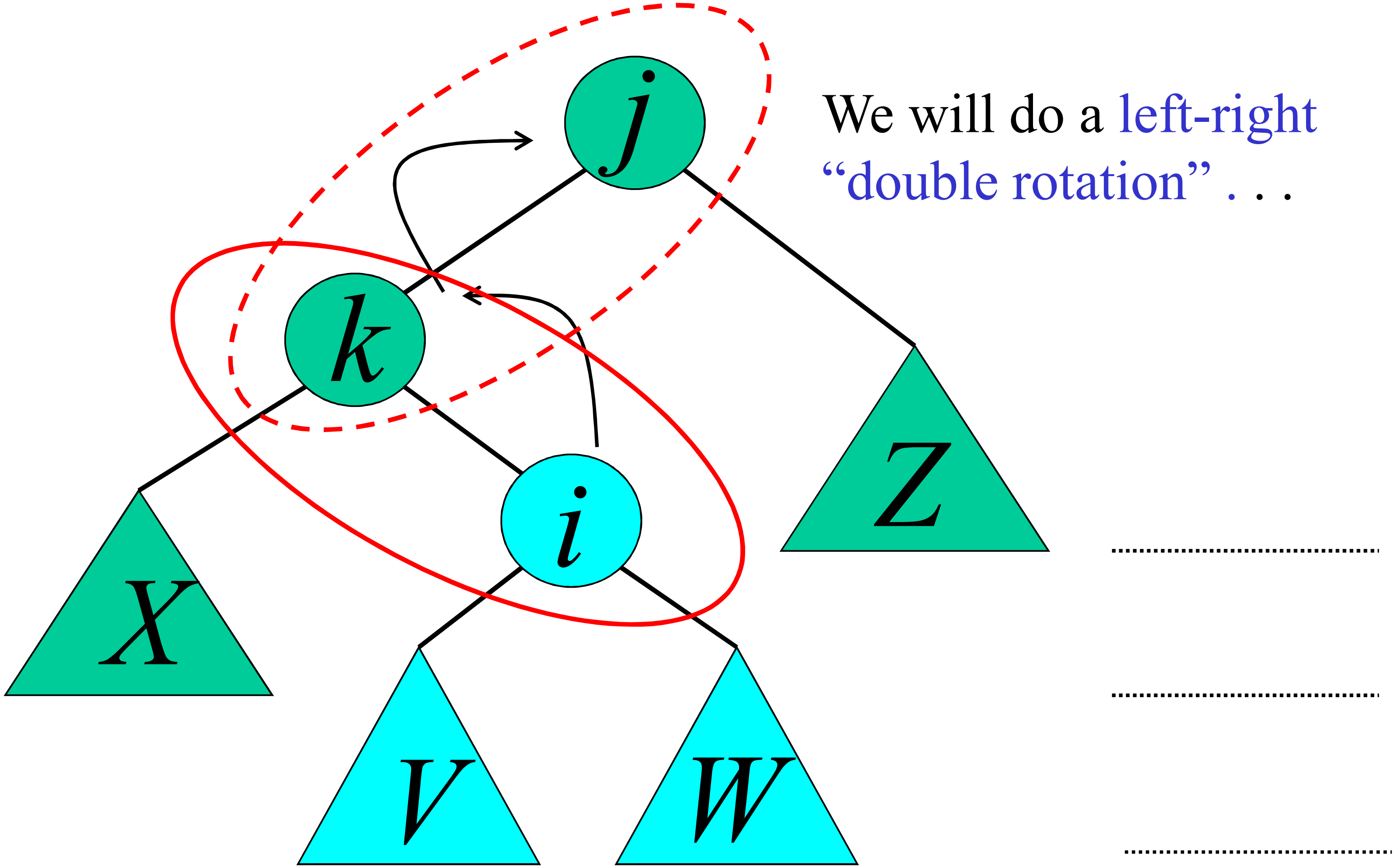


AVL Insertion: Inside Case

Y = node i and
subtrees V and W



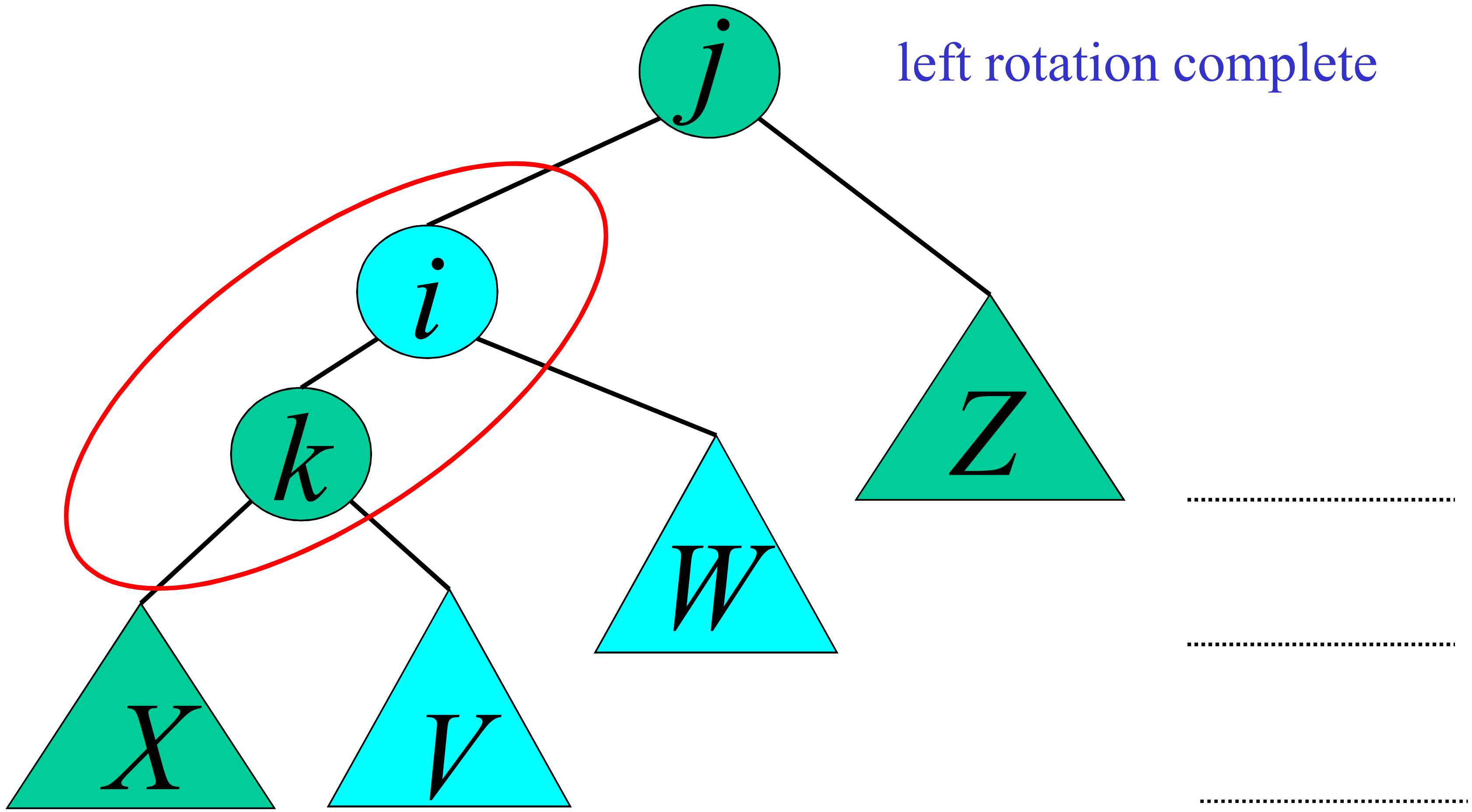
AVL Insertion: Inside Case



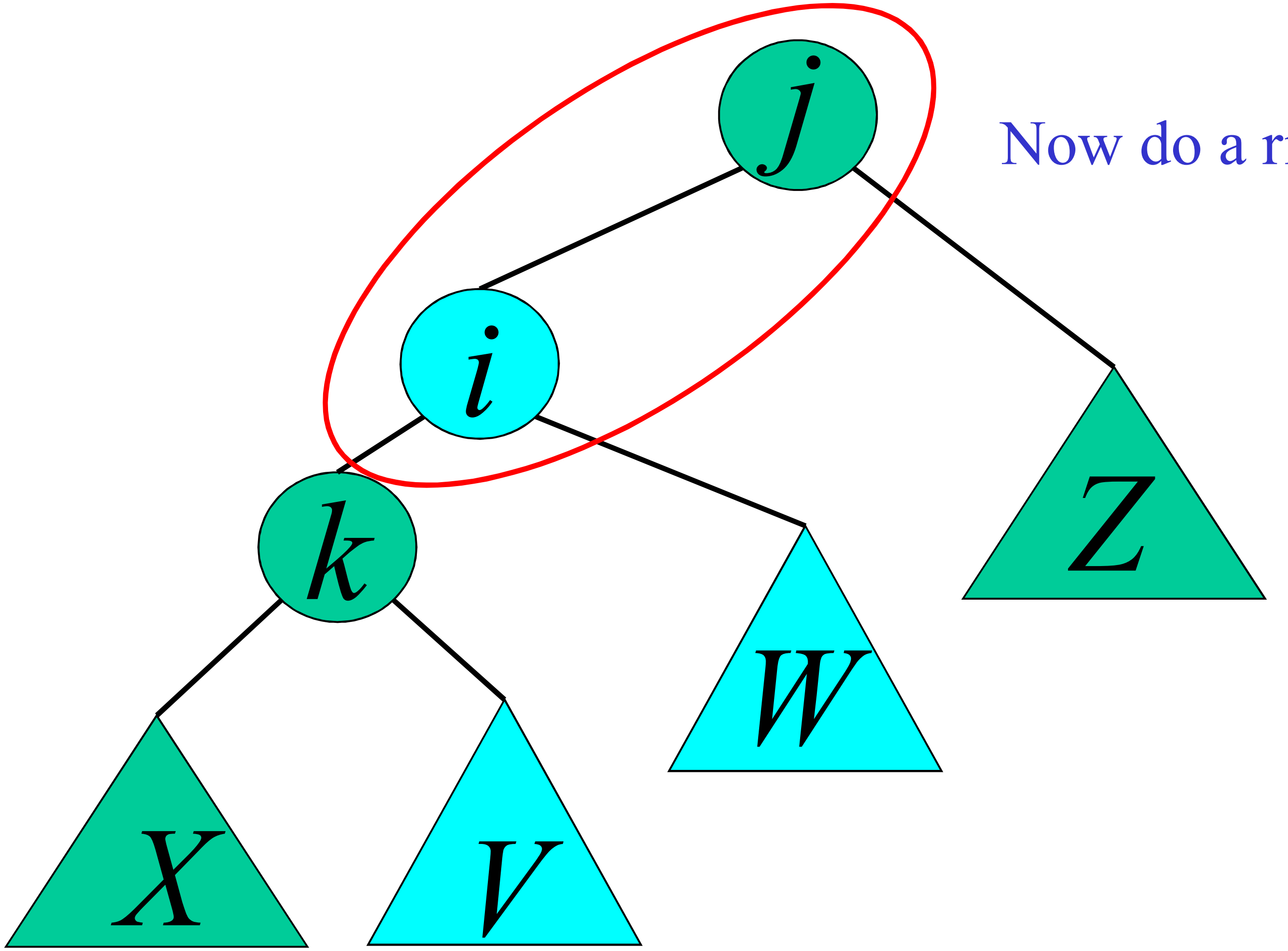
We will do a left-right "double rotation" . . .

Double rotation : first rotation

left rotation complete



Double rotation : second rotation

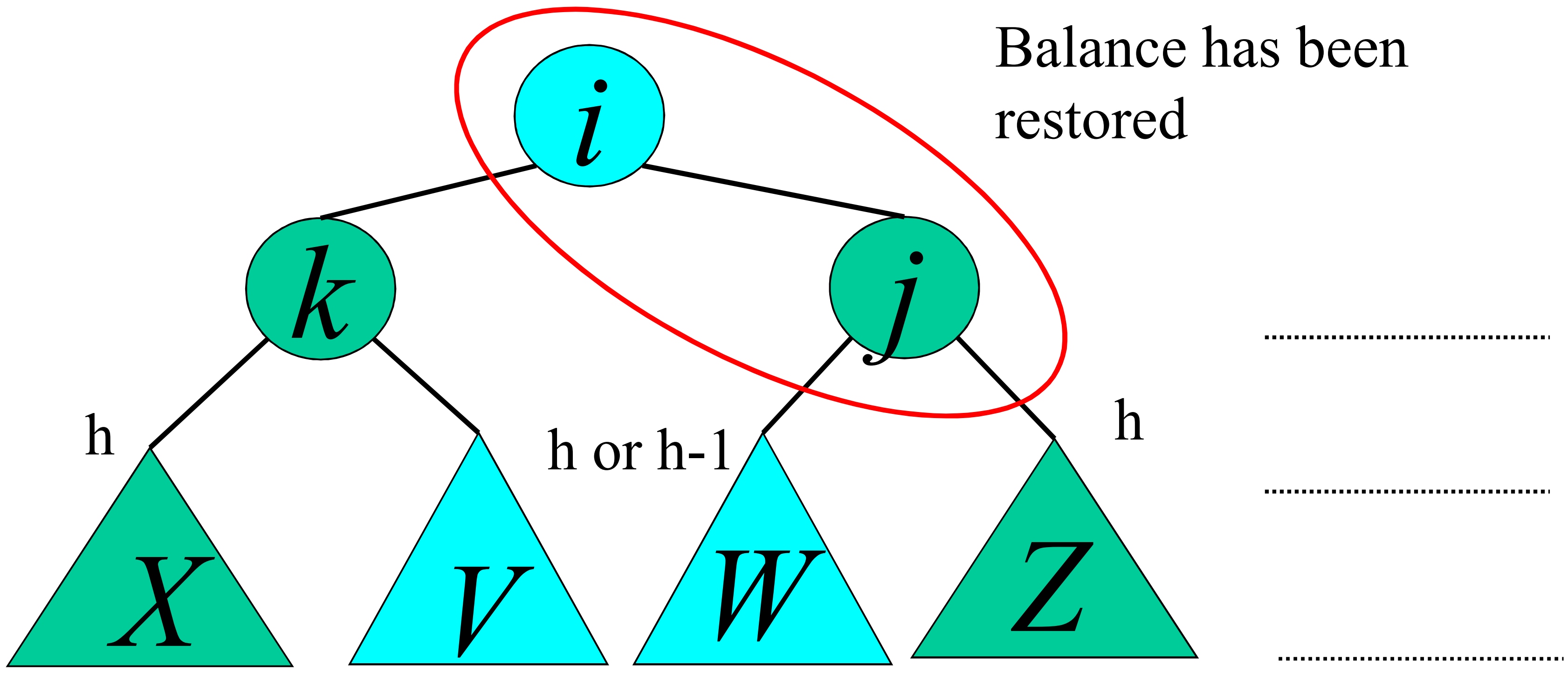


Now do a right rotation

Double rotation : second rotation

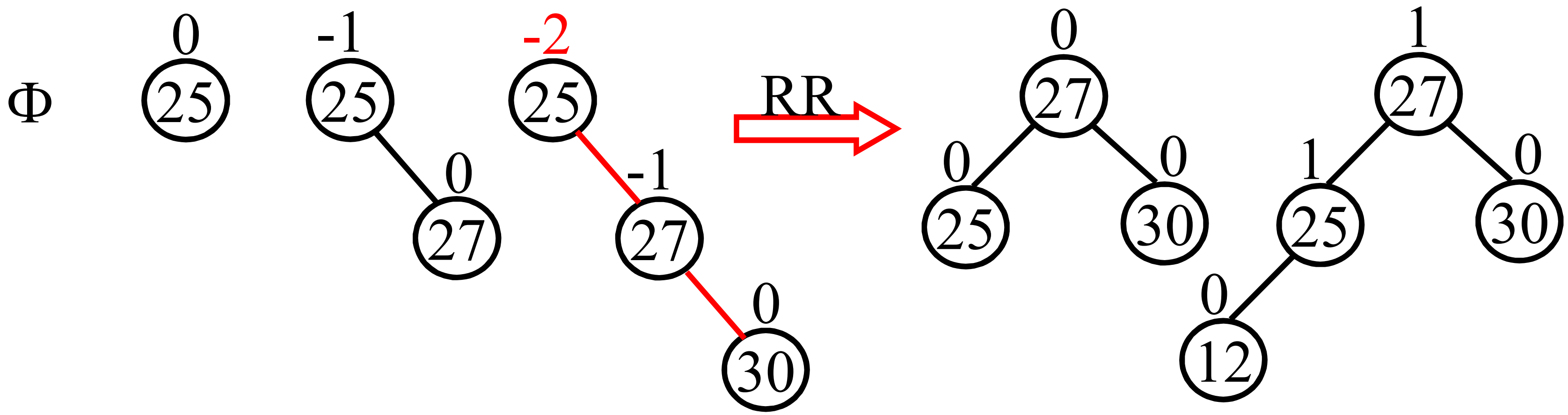
right rotation complete

Balance has been restored



AVL Insertion: an example

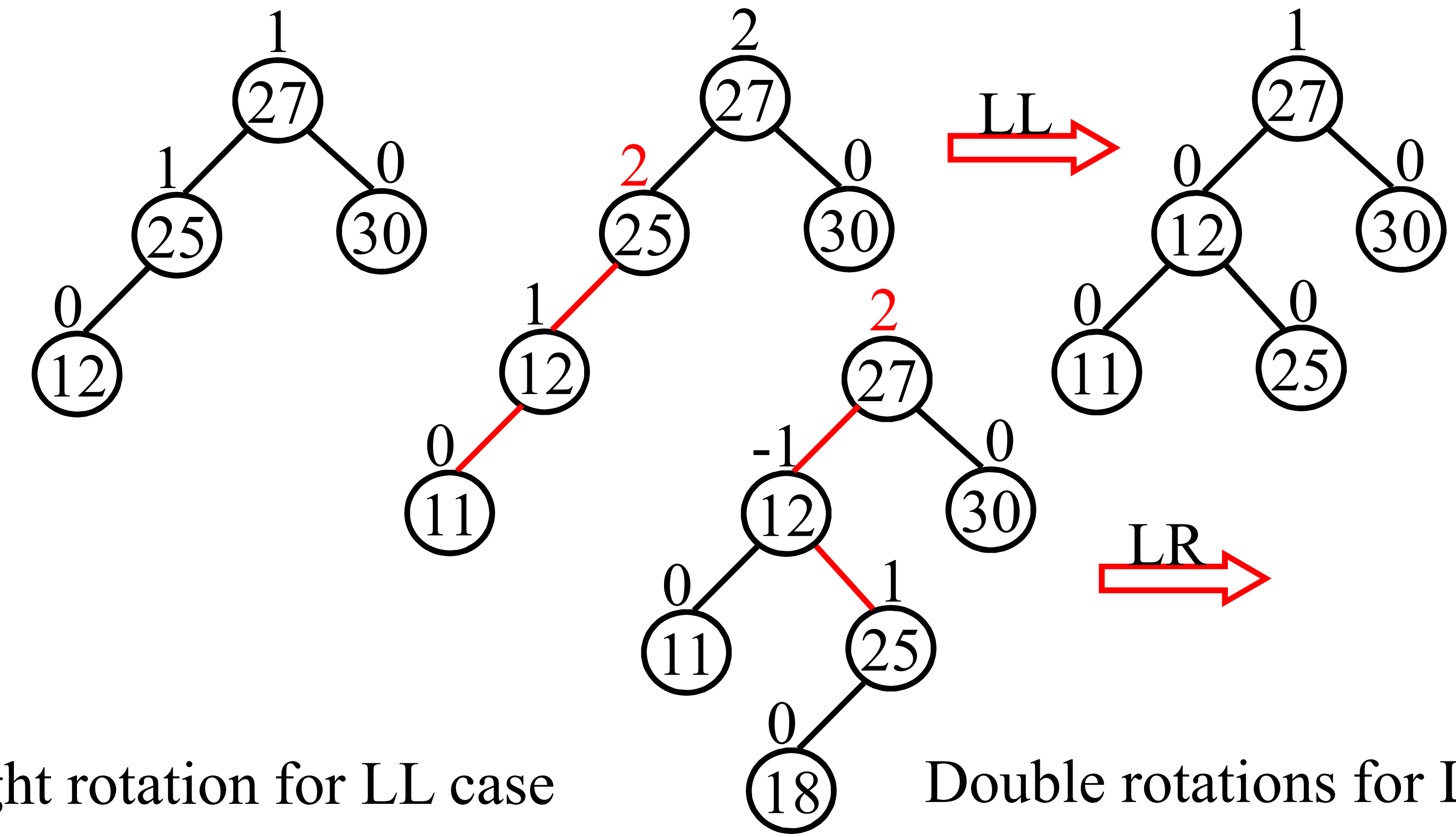
- Insert: 25, 27, 30 , 12 , 11 , 18 , 14 , 20



Left rotation for RR case

AVL Insertion: an example

- Insert: 25, 27, 30 , 12 , 11 , 18 , 14 , 20, 15

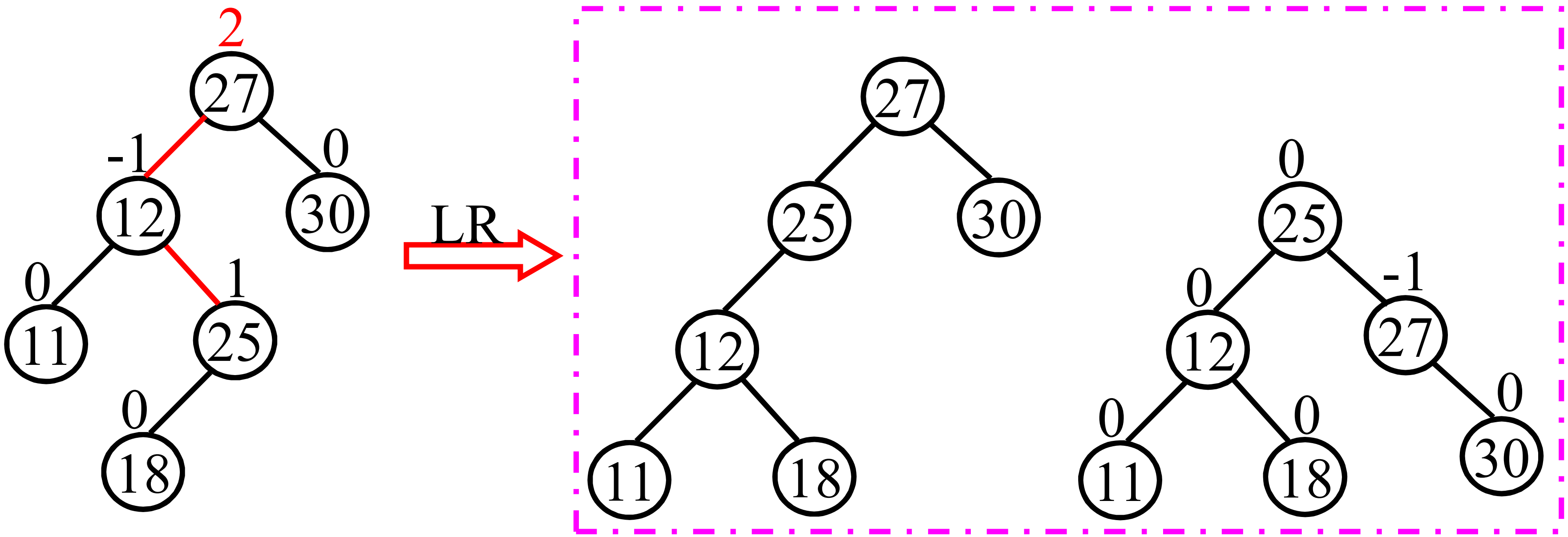


Right rotation for LL case

Double rotations for LR case

AVL Insertion: an example

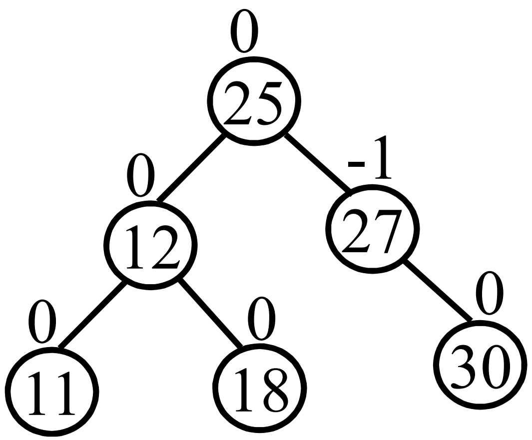
- Insert: 25, 27, 30 , 12 , 11 , 18 , 14 , 20 , 15

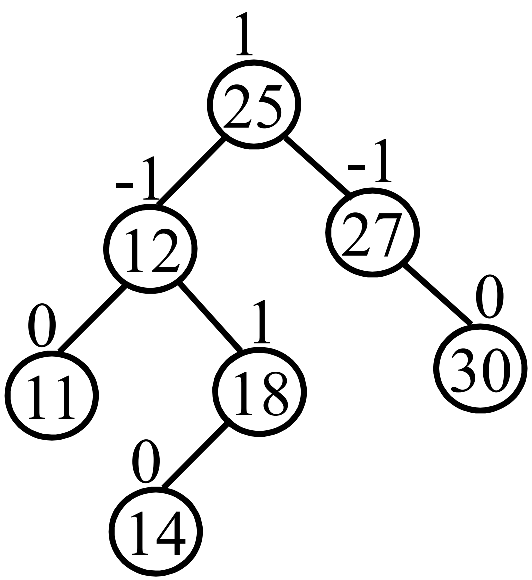


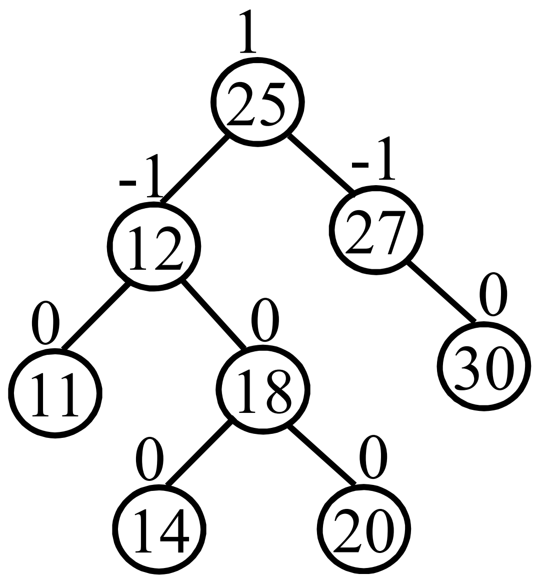
left rotation + right rotation

AVL Insertion: an example

- Insert: 25, 27, 30 , 12 , 11 , 18 , 14 , 20 , 15

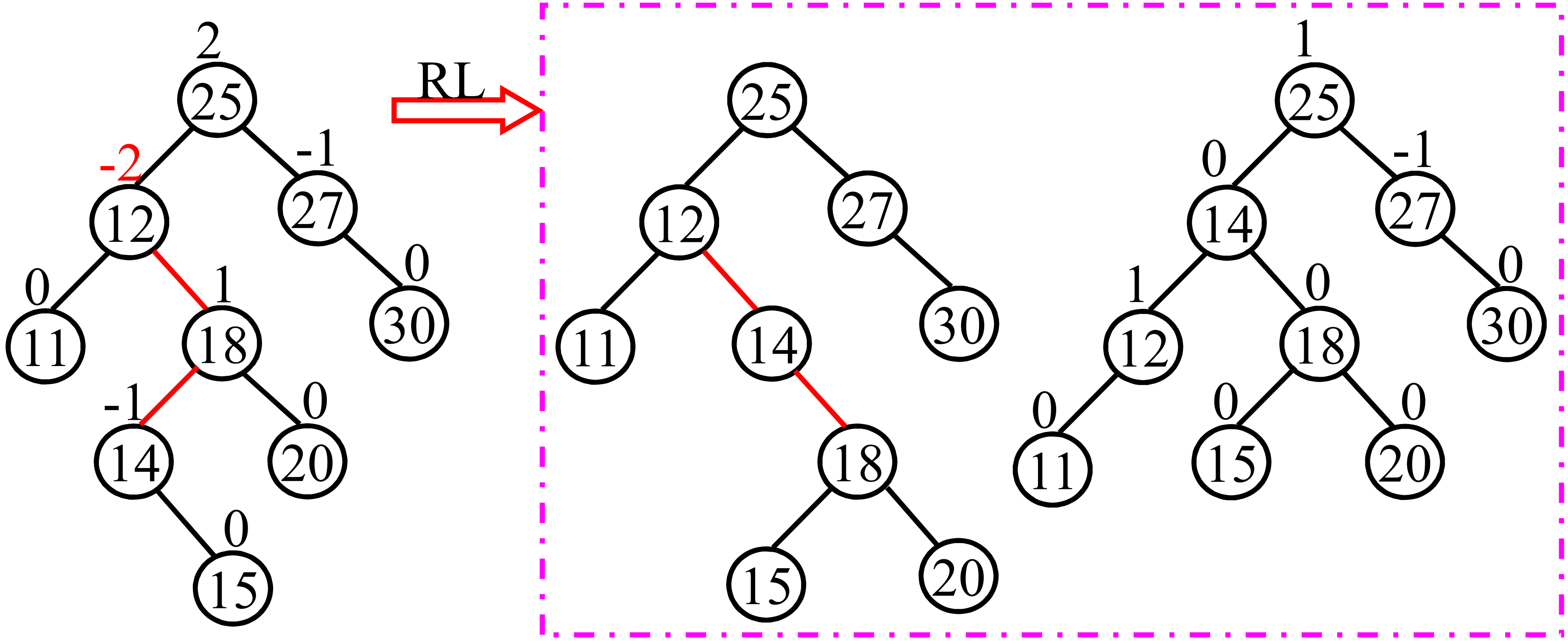






AVL Insertion: an example

- Insert: 25, 27, 30 , 12 , 11 , 18 , 14 , 20 , 15

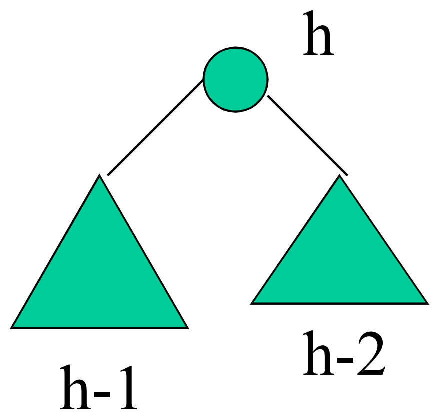


AVL Tree Deletion

- Similar but more complex than insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed.

Height of an AVL Tree

- $N(h)$ = minimum number of nodes in an AVL tree of height h.
- Basis
  - $N(0) = 1$, $N(1) = 2$
- Induction
  - $N(h) = N(h-1) + N(h-2) + 1$
- Solution (recall Fibonacci analysis)
  - $N(h) \geq \phi^h \quad (\phi \approx 1.62)$



Height of an AVL Tree

- $N(h) \geq \phi^h$ $(\phi \approx 1.62)$
- Suppose we have n nodes in an AVL tree of height h.
  - $n \geq N(h)$ (because N(h) was the minimum)
  - $n \geq \phi^h$ hence $\log_\phi n \geq h$ (relatively well balanced tree!!)
  - $h \leq 1.44 \log_2 n$
    (i.e., Find takes O(logn))

Pros and Cons of AVL Trees

Arguments for AVL trees:

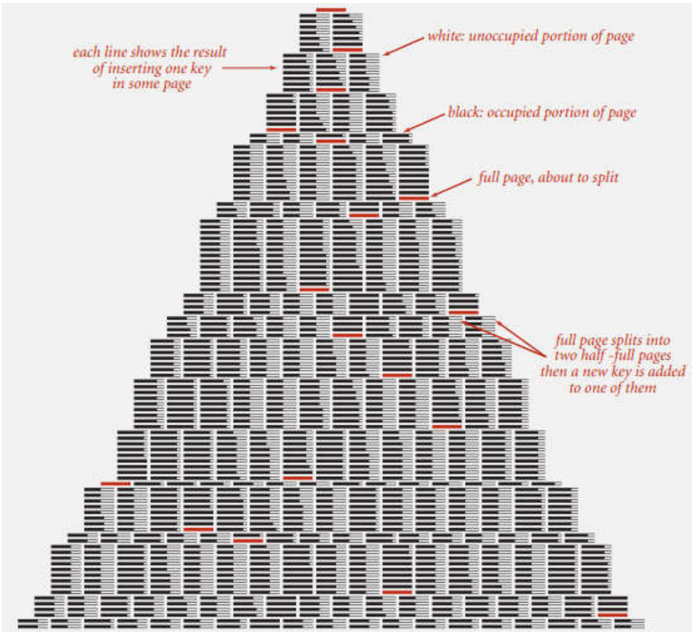
1. Search is O(log N) since AVL trees are always balanced.
2. Insertion and deletions are also O(logn)
3. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees:

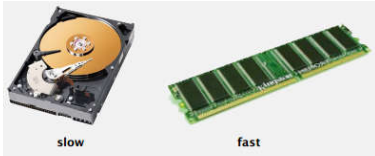
1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).

B- Trees

Invented by R. Bayer in 1970



B-Trees



- B-trees address effectively the major problems encountered when implementing disk-based search trees:
  1. Update and search operations affect only a few disk blocks.
     - The fewer the number of disk blocks affected, the less disk I/O is required.
  3. B-trees keep related records (that is, records with similar key values) on the same disk block, which
     - helps to minimize disk I/O on searches due to locality of reference.
  4. B-trees guarantee that every node in the tree will be full at least to a certain minimum percentage.
     - This improves space efficiency while,
     - reducing the typical number of disk fetches necessary during a search or update operation

B-Trees

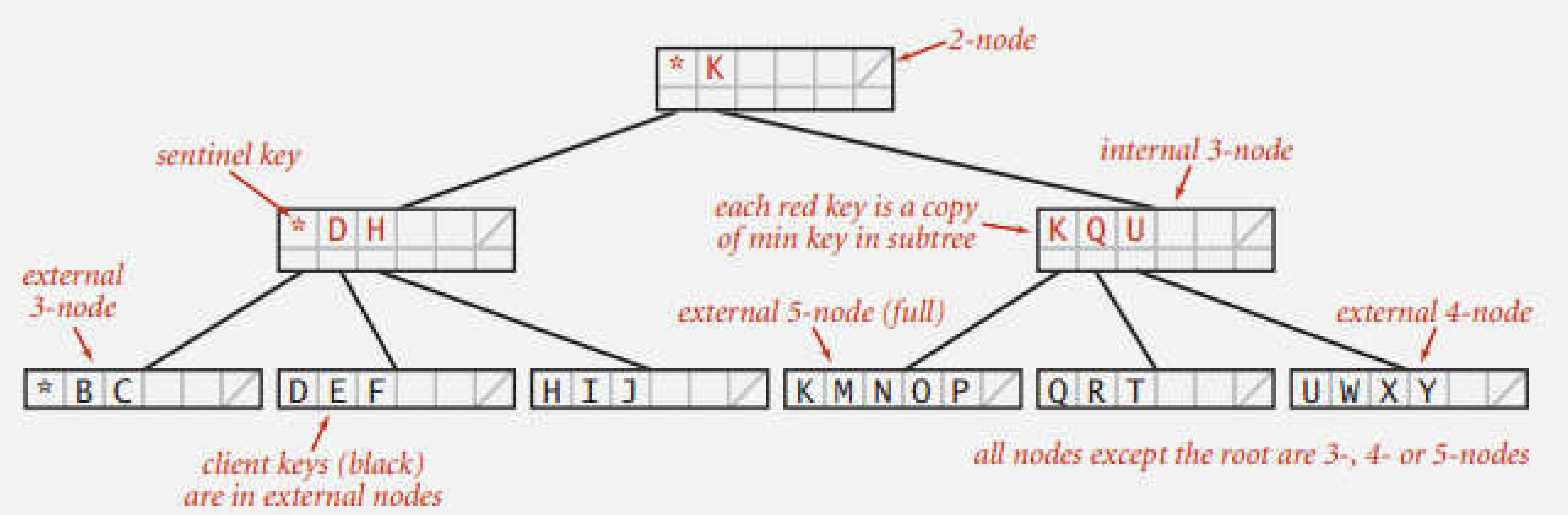
- A B-tree of order M is defined to have the following <u>shape properties</u>:
  - The root is either a leaf or has from two to M children.
  - Each internal node, has between $\lceil M/2 \rceil$ and M children (key-link pairs).

choose M as large as possible so that M links fit in a page, e.g., M = 1024

  - All leaves are at the same level in the tree, so the tree is always height balanced.
  - External nodes contain client keys.
  - Internal nodes contain copies of keys to guide search.

B-tree example

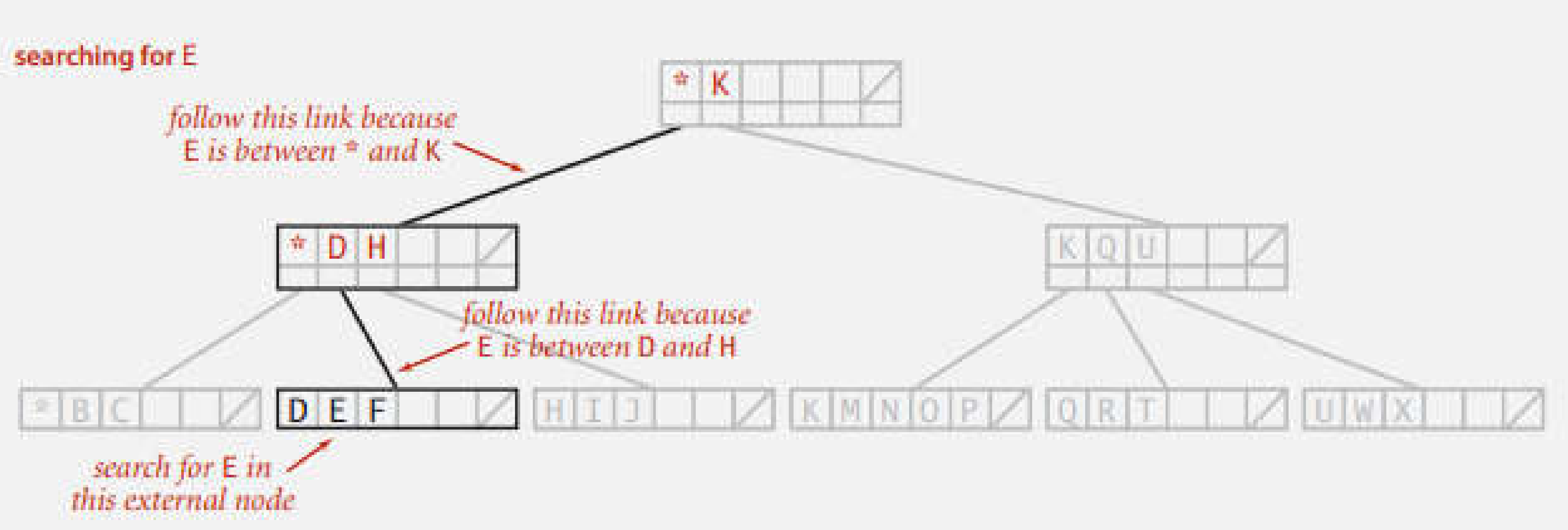
- A B-tree of order six



Anatomy of a B-tree set (M = 6)

Searching in a B-tree

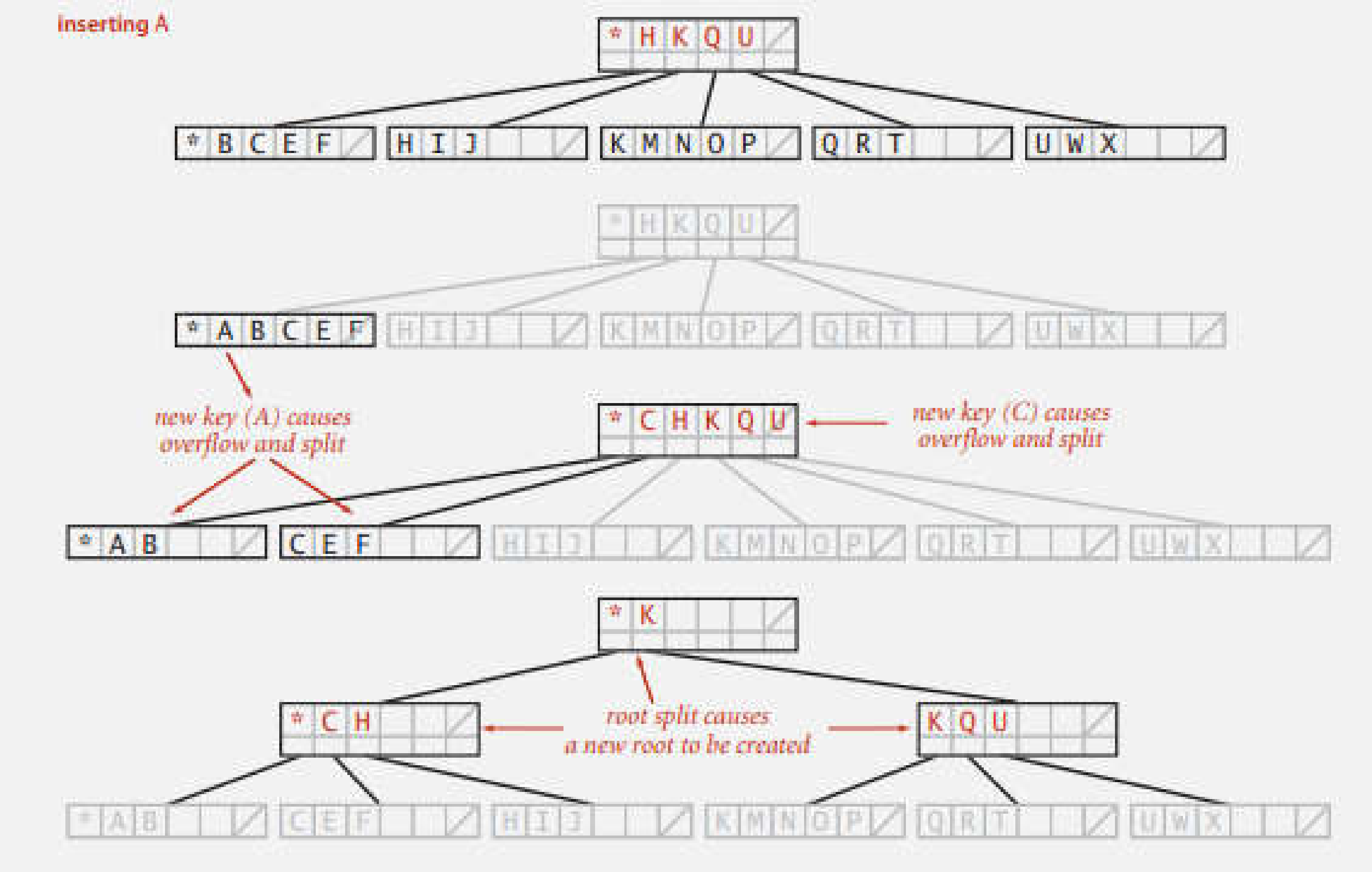
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Searching in a B-tree set (M = 6)

Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with M nodes(key-link pairs) on the way up the tree.



Inserting a new key into a B-tree set

Balance in B-tree

- The height h of B-tree of order m and the number N of keys in the tree
  - First level: 1 node
  - Second level: at least 2 nodes
  - Third level: at least $2\lceil m/2 \rceil$
  - Forth level: at least $2\lceil m/2 \rceil^2$
  - …
  - h level: at least $2\lceil m/2 \rceil^{h-2}$
  - Missed node at $h+1$ level

Balance in B-tree

maxmum height < $\mathbf{\log_{\mathit{m}/2}((\mathit{N}+\mathit{1})/2)}$

So search is $O(\log_{m/2}(N))$

If $m = 199$, $N = 1999999$, then

h is not more then

$$\log_{\lceil 199/2 \rceil}((1999999+1)/2)+1 = \log_{100}1000000+1 = \mathbf{4}$$

Some max. height examples

- For M = 200

  $\log_{M/2}(1M) <= 3$

  $\log_{M/2}(1G) <= 5$

  $\log_{M/2}(1T) <= 7$

  $\log_{M/2}(1P) <= 8$

  $\log_{M/2}(1E) <= 9$

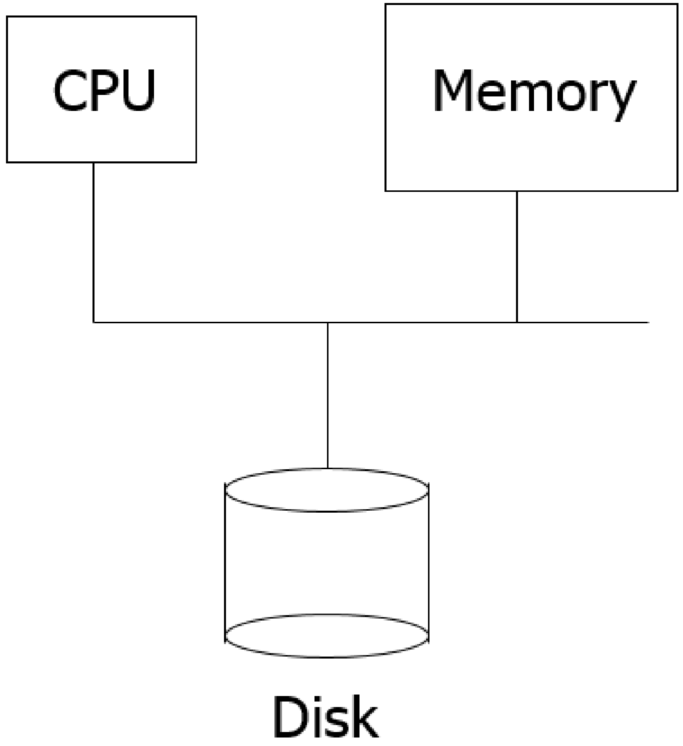
Balance in B-tree

- **Proposition.** A search or an insertion in a B-tree of order M with N keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes.
- **Pf.** All internal nodes (besides root) have between M/2 and M-1 links.

- In practice：
  Number of probes is at most 4. ← M = 1024; N = 62 billion
  $\log_{M/2} N \leq 4$
- Optimization：
  Always keep root page in memory

Model of Computation

- Data stored on disk(s)
- Minimum transfer unit: a page = b bytes or B records (or block)
- N records -> N/B = n pages
- I/O complexity: in number of pages



I/O complexity

- An ideal index has
  - space O(N/B),
  - update overhead O(1) or $O(\log_B(N/B))$ and
  - search complexity O(*a*/B) or $O(\log_B(N/B) + a/B)$ where *a* is the number of records in the answer

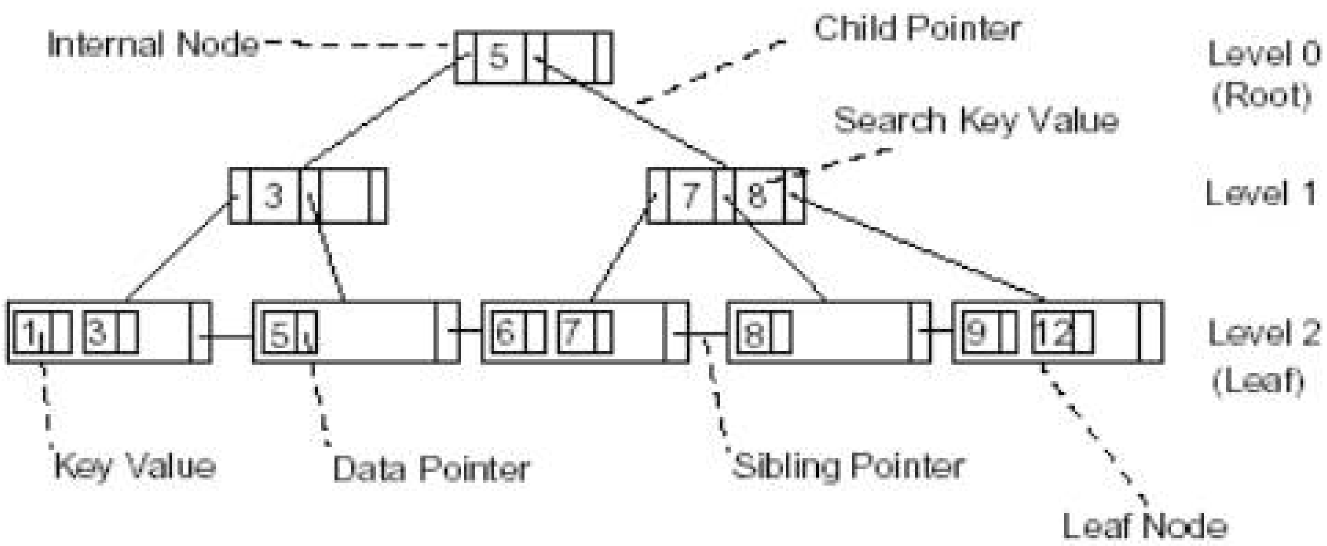
- But, sometimes CPU performance is also important… minimize cache misses -> don't waste CPU cycles

B+ tree

- B+ tree :
  - a variant of the B-tree
  - most commonly used for indexing files
- The most significant difference between the B+ tree and the B-tree
  - the B+ tree stores records only at the leaf nodes.
  - Internal nodes store key values, but these are used solely as placeholders to guide the search.

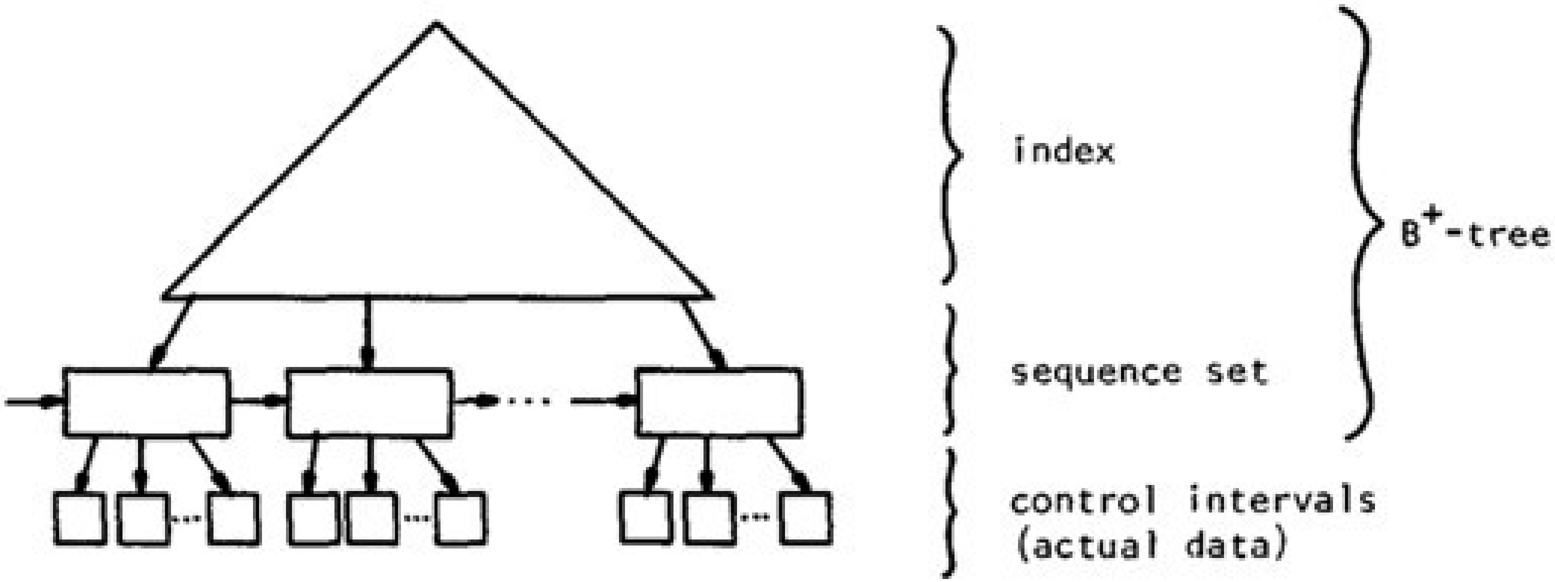
B+ tree

- The B+ tree is essentially a mechanism for managing a sorted array-based list, where the list is broken into chunks.

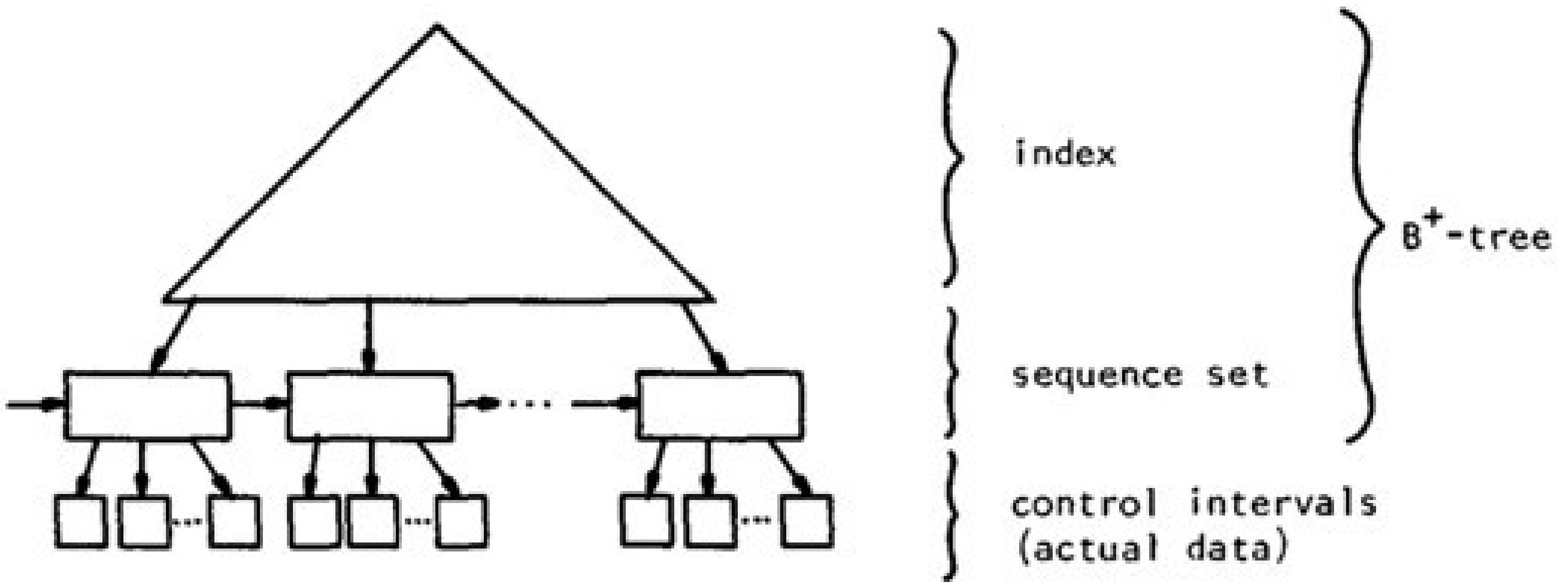


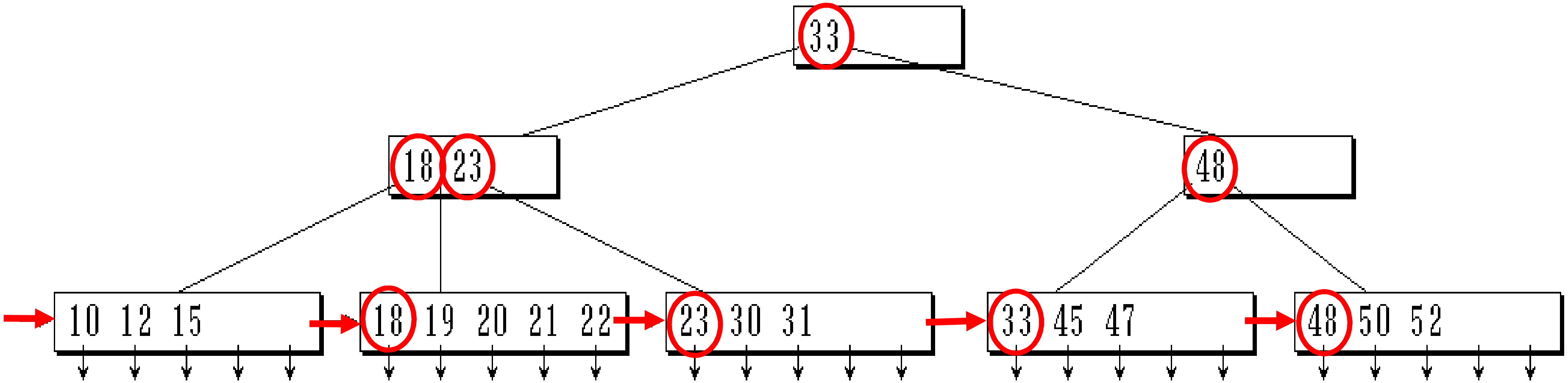
B+ Tree: Properties

- Tree-structured indexes are ideal for *range searches*, also good for *equality searches*.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced;

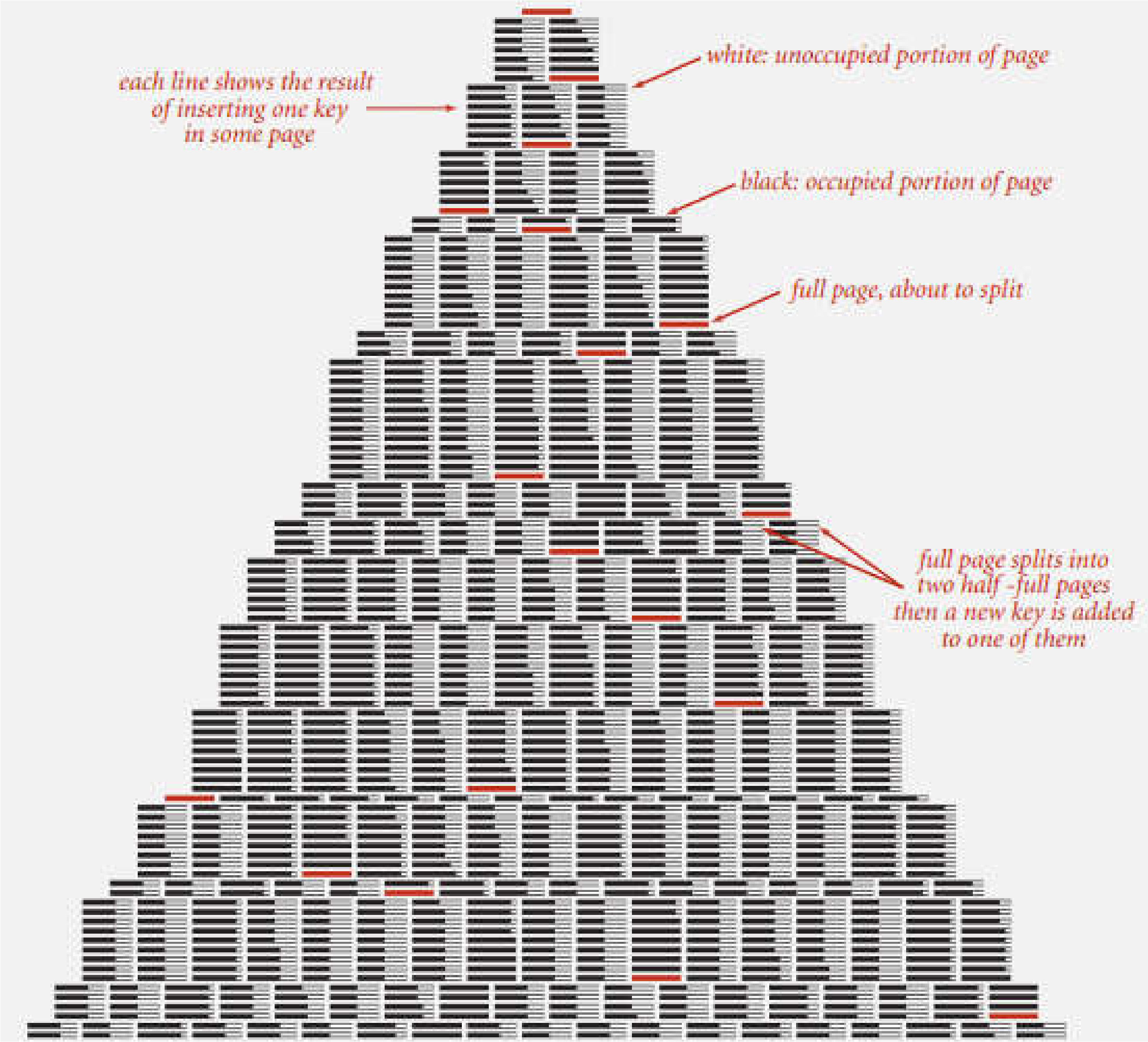


B+ tree: example





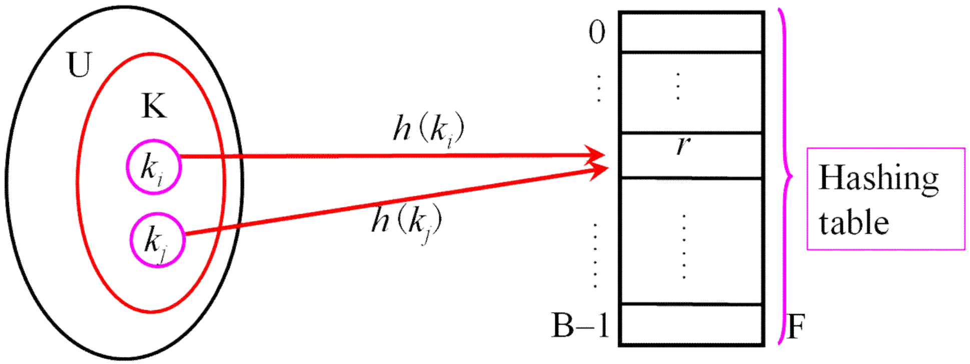
Building a large B tree



HASHING

Hashing

- Hashing:
  - The process of finding a record using some computation to map its key value to a position in the array is called hashing.
- Hash function:
  - The function that maps key values to positions,
  - Denoted by h
- Hash table:
  - The array that holds the records,
  - Denoted by HT.
- Slot:
  - A position in the hash table.

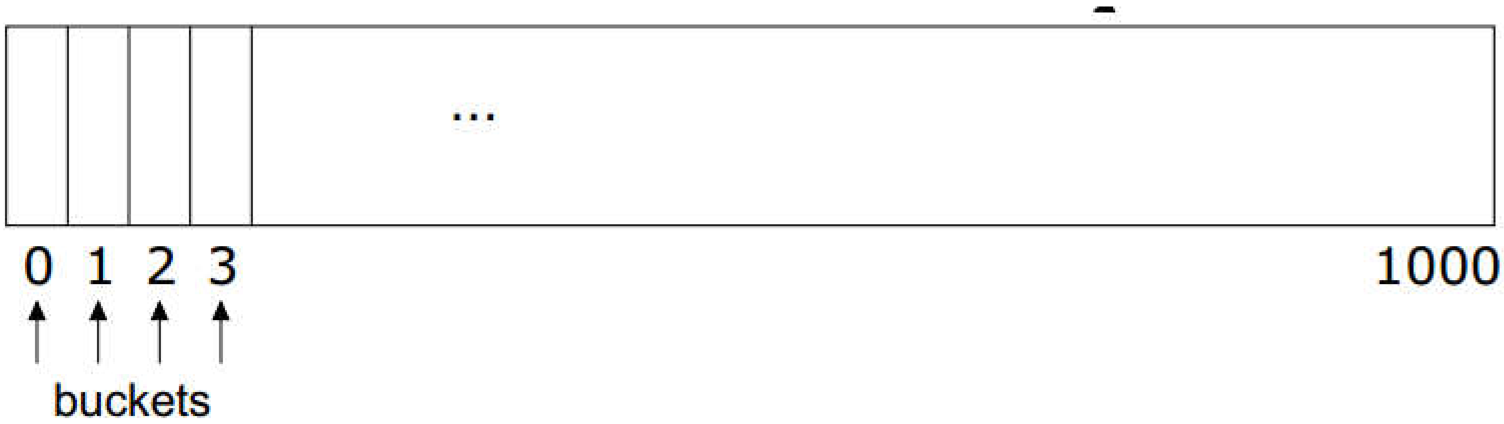


Basic Idea

- Use **hash function** to map keys into positions in a **hash table**
- Ideally
  - If element *e* has key *k* and *h* is hash function, then *e* is stored in position *h(k)* of table
- To search for *e*,
  - compute *h(k)* to locate position.
  - If no element, array does not contain *e*.

Example: Dictionary Student Records

- Dictionary Student Records
  - Keys are ID numbers (951000 - 952000), no more than 1000 students
  - Hash function: h(k) = k-951000 maps ID into distinct table positions 0-1000
  - array table[1001]



Analysis

- Ideal Case
  - O(b) time to initialize hash table (b is the number of positions or buckets in the hash table)
  - O(1) time to perform *insert, remove, search* operations
- Ideal Case is Unrealistic
  - But many applications have key ranges that are too large to have 1-1 mapping between buckets and keys!

Hash Functions

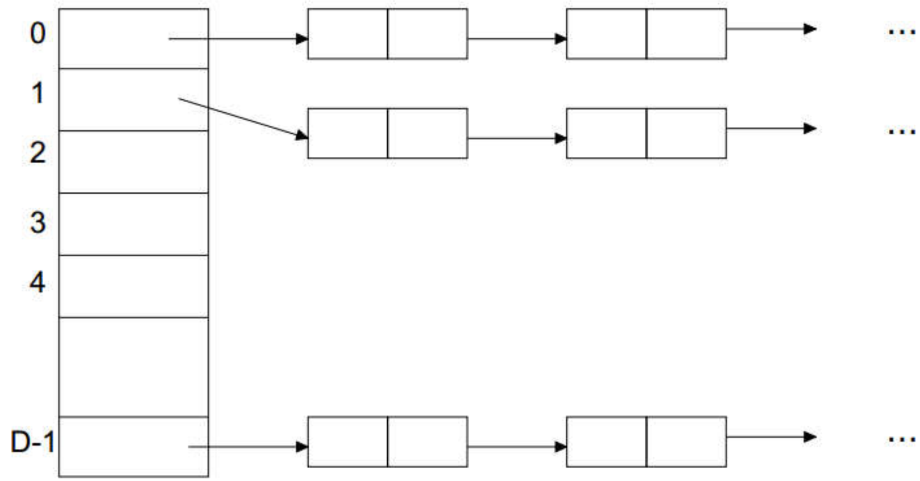
- If key range too large, use hash table with fewer buckets and a hash function which maps multiple keys to same bucket:
  - $h(k_1) = E = h(k_2)$
  - k_1 and k_2 have collision at slot E
- Popular hash functions: hashing by division:
  - $h(k) = k\%D$,
  - Usually, D is the maximum prime number and no more than N
  - N is the number of buckets in hash table
- Example: hash table with 11 buckets
  - $h(k) = k\%11$
  - 80 & 3 (80%11= 3), 40 & 7, 65&10, 58&3 are collisions!

Collision Resolution Policies

- Two classes:
  - (1) Open hashing, a.k.a. separate chaining
  - (2) Closed hashing, a.k.a. open addressing
- Difference has to do with
  - whether collisions are stored outside the table(open hashing) or
  - whether collisions result in storing one of the records at another slot in the table(closed hashing)

Open Hashing

- Each bucket in the hash table is the head of a linked list
- All elements that hash to a particular bucket are placed on that bucket's linked list
- Records within a bucket can be ordered in several ways
  - by order of insertion
  - by key value order, or
  - by frequency of access order



Analysis

- Open hashing is the most appropriate
  - when the hash table is kept in main memory,
  - implemented with a standard in-memory linked list
- We hope that number of elements per bucket roughly equal in size,
  - so that the lists will be short
- If there are n elements in set, then each bucket will
  - have roughly n/D
- If we can estimate n and choose D to be roughly as large, then the average bucket will
  - have only one or two members

Analysis Cont'd

- Average time per dictionary operation:
  - D buckets, *n* elements in dictionary average *n*/D elements per bucket
  - *insert, search, remove* operation take O(1+*n*/D) time each
  - If we can choose D to be about n, constant time
  - Assuming each element is likely to be hashed to any bucket, running time constant, **independent of *n***

Closed Hashing

Associated with closed hashing is **a rehash strategy**:

- If we try to place x in bucket $h(x)$ and find it occupied, find alternative location $h_1(x)$, $h_2(x)$, etc.
  - Try each in order
  - $h(x)$ is called home bucket
- Simplest rehash strategy is called **linear hashing**

$$h_i(x) = (h(x) + i) \% D$$

  - The collision resolution strategy is to generate (probe) a sequence of hash table slots that can hold the record
  - Test each slot until find empty one

Example Linear (Closed) Hashing

D=8, keys *a,b,c,d* have hash values h(a)=3, h(b)=0, h(c)=4, h(d)=3

- Where do we insert d? 3 already filled
- Probe sequence using linear hashing:
  - $h_1(d) = (h(d)+1)\%8 = 4\%8 = 4$
  - $h_2(d) = (h(d)+2)\%8 = 5\%8 = 5$
  - $h_3(d) = (h(d)+3)\%8 = 6\%8 = 6$
  - etc.
  - 7, 0, 1, 2
- Wraps around the beginning of the table!

0	b
1	
2	
3	a
4	c
5	**d**
6	
7	

Operations Using Linear Hashing

Find Item : test for membership

- Examine $h(k)$, $h_1(k)$, $h_2(k)$, …, until we
  - find k or
  - an empty bucket or
  - home bucket
- If no deletions possible, strategy works!
- What if deletions?
  - If we reach empty bucket, cannot be sure that k is not somewhere else and
  - empty bucket was occupied when k was inserted
- Need special placeholder deleted,
  - to distinguish bucket that was never used from one that once held a value
- May need to reorganize table after many deletions

Performance Analysis - Worst Case

- Initialization: O(b), b# of buckets
- Insert and search:
  - O(n), n number of elements in table;
  - all n key values have same home bucket
- No better than linear list for maintaining dictionary!

Performance Analysis - Avg Case

- Distinguish between successful and unsuccessful searches
  - Delete = successful search for record to be deleted
  - Insert = unsuccessful search along its probe sequence
- Expected cost of hashing is a function of how full the table is:

  load factor $\alpha = n/b$
- It has been shown that average costs under linear hashing (probing) are:

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

search hit

$$\sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

search miss / insert

Performance Analysis - Avg Case

- It has been shown that average costs under linear hashing (probing) are:

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

search hit

$$\sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

search miss / insert

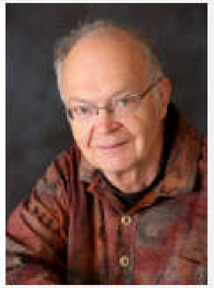
Pf.

[My first analysis of an algorithm, originally done during Summer 1962 in Madison.]

8354

NOTES ON "OPEN" ADDRESSING. D. Knuth 7/22/63

1. Introduction and Definitions. Open addressing is a widely-used technique for keeping "symbol tables." The method was first used in 1954 by Samuel, Amdahl, and Boehme in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solution after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.



Parameters.

- b too large ⇒too many empty array entries.
- b too small ⇒search time blows up.
- Typical choice: $\alpha = n/b \sim ½$

\# probes for search hit is about 3/2

\# probes for search miss is about 5/2

Improved Collision Resolution

- Linear probing: $h_i(x) = (h(x) + i) \% D$
  - all buckets in table will be candidates for inserting a new record before the probe sequence returns to home position
- Linear probing with skipping: $h_i(x) = (h(x) + d_i) \% D$
  - $d_i = 1c, 2c, \ldots$, c is constant other than 1
- Quadratic probing : $h_i(x) = (h(x) + d_i) \% D$
  - $d_i = 1^2, -1^2, 2^2, -2^2, \ldots, q^2, -q^2$ and $q \le B/2$)
- (Pseudo)Random probing: $h_i(x) = (h(x) + r_i) \% D$
  - r_i is the i^{th} value in a random number from 1 to D-1
  - insertions and searches use the same sequence of "random" numbers

Linear probing algorithm(c=1): search

- Storage structure:

```
struct records{
    keytype  key;
    fields other;
};
typedef records
    HsASH[B];
```

```
int  Search(keytype k, HASH F)
{   int locate=first= h(k),rehash=0;
    while((rehash<B)&&
            (F[locate].key!=empty)){
        if(F[locate].key==k)
                return locate;
        else
                rehash=rehash+1;
        locate=(first+rehash)%B
    }
    return -1;
}/*Search*/
```

Linear probing algorithm(c=1) : Insert & Delete

```
void Insert(records R, HASH F)
{   int locate = first=h(k), rehash= 0;
    while((rehash<B)&&
            (F[locate].key!=R.key)) {
       locate=(first+rehash)%B;
       if((F[locate].key==empty)||
            (F[locate].key==deleted))
          F[locate]=R;
       else
          rehash+=1; }
    if(rehash>=B)
          cout<<"hash table is full!";
} /*Insert */
```

```
void Delete(keytype k,HASH F)
{       int locate;
      locate = Search(k,F);
      if( locate != -1)
      F[locate].key = deleted;
}/*Delete*/
```

Hash Functions - Numerical Values

- Example:
  consider: $h(x) = x\%16$
  - poor distribution, not very random
- Better, **mid-square method**
  - if keys are integers in range 0,1, …, K , pick integer C such that DC^2 about equal to K^2, then

$$h(x) = \lfloor x^2/C \rfloor \% D$$

Item	*key*	*key*^{2}	Hash
A	0100	0 010 000	010
I	1100	1 210 000	210
J	1200	1 440 000	440
I0	1160	1 370 400	370
P1	2061	4 310 541	310
P2	2062	4 314 704	314
Q1	2161	4 734 741	734
Q2	2162	4 741 304	741
Q3	2163	4 745 651	745

Hash Functions - Numerical Values

- Better, **mid-square method**
  - extracts middle r bits of x^2, where 2^r=D (a base-D digit), D (# bucket)
  - better, because most or all of bits of key contribute to result

Item	*key*	*key*^{2}	Hash
A	0100	0 010 000	010
I	1100	1 210 000	210
J	1200	1 440 000	440
I0	1160	1 370 400	370
P1	2061	4 310 541	310
P2	2062	4 314 704	314
Q1	2161	4 734 741	734
Q2	2162	4 741 304	741
Q3	2163	4 745 651	745

Performance

- Worst case performance is O(n) for both open hashing and closed hashing.

More practice !

Example

- List of key values: (7, 8 , 30 , 11 , 18 , 9 , 14)
- Hash table: one dimension array
- Hash function: H(key)=(key*3)%7，
  - linear hashing if collision
  - load factor is 0.7。

(a) Work out the hash table

(b) ASL (average searching length):

- hit and miss with the same probability.

Example

- List of key values: (7, 8 , 30 , 11 , 18 , 9 , 14)
- Hash function: H(key)=(key*3)%7，
- load factor α= n/b

  n# of item is 7

  b# of bucket is 10
- load factor is 0.7

```
int  Search(keytype k, HASH F)
{   int locate=first= h(k),rehash=0;
    while((rehash<B)&&
            (F[locate].key!=empty)){
        if(F[locate].key==k)
            return locate;
        else
            rehash=rehash+1;
        locate=(first+rehash)%B
    }
    return -1;
}/*Search*/
```

Solution

(a)Length of the list is 7, load factor is 0.7. So, the size of the array is 10, index : 0~9。

Array index	0	1	2	3	4	5	6	7	8	9
Array element	7	14		8		11	30	18	9	
Order of insert	1	7		2		4	3	5	6	
No comp. for hit	1	2		1		1	1	3	3	
No comp. for miss	3	2	1	2	1	5	4	--	--	--

(b) ASL_{hit} =(1+2+1+1+1+3+3)/7=12/7

ASL_{miss} =(3+2+1+2+1+5+4)/7=18/7