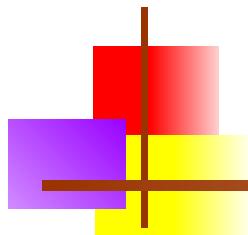
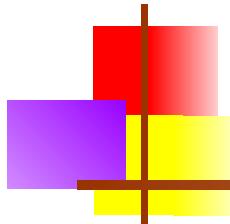
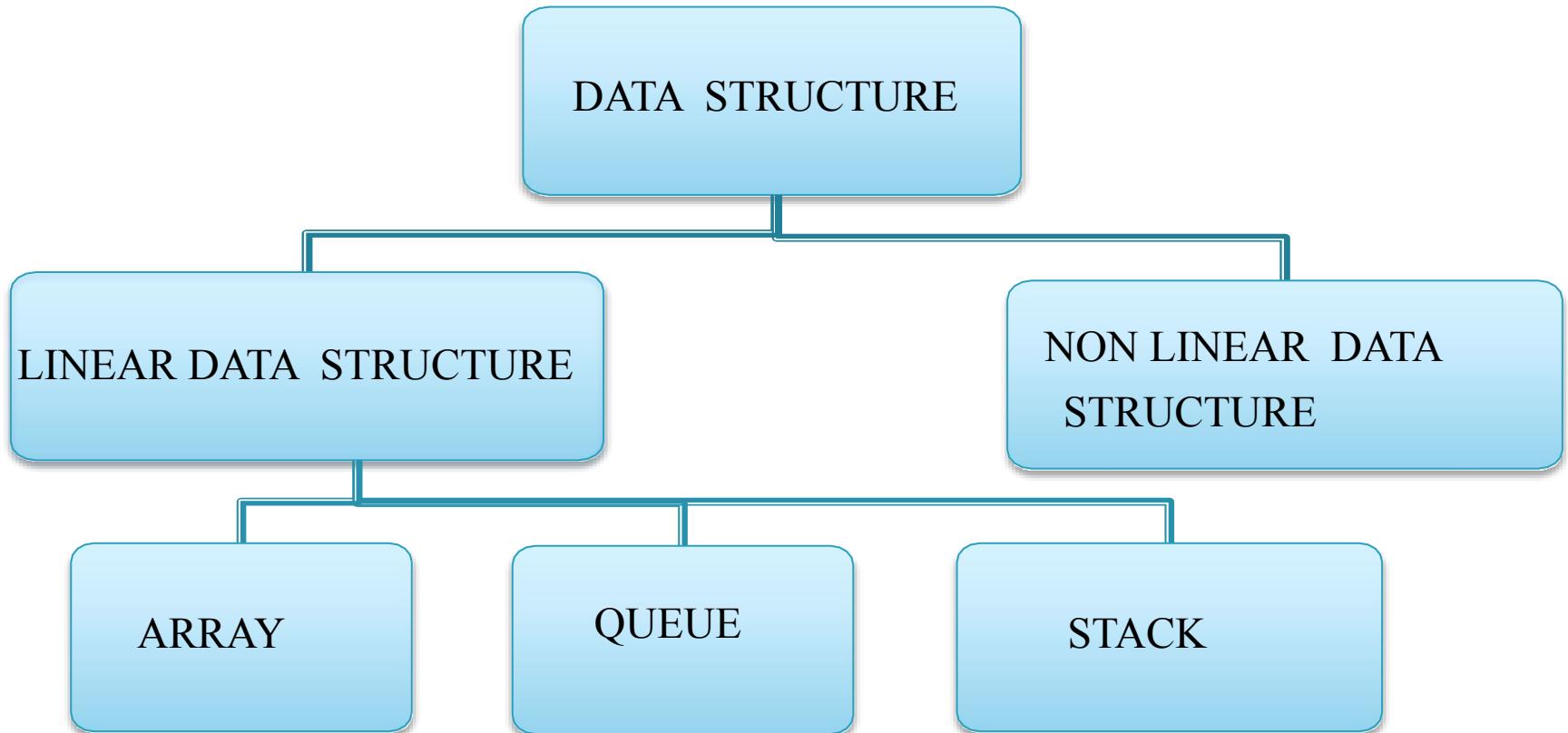


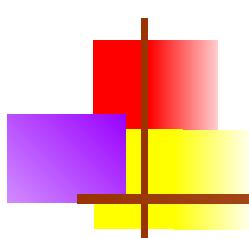
# Stacks and Queues





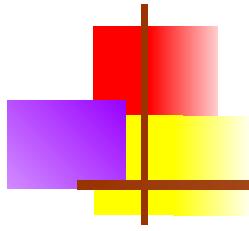
# Stacks and Queues





# What is linear data structure ?

- In **linear data structure**, data is arranged in linear sequence.
- Data **items** can be traversed in a single run.
- In linear data structure **elements** are accessed or placed in **contiguous** (together in sequence) memory locations.



# What is stack



- A **stack** is a last in, first out (**LIFO**) data structure
  - Items are removed from a stack in the reverse order from the way they were inserted
  - The last items we added (pushed) is the first item that gets pulled (popped) off.
  - A stack is a sequence of items that are accessible at only **one end** of the sequence.

# Examples of stack



A stack of  
books



A pile of  
plates

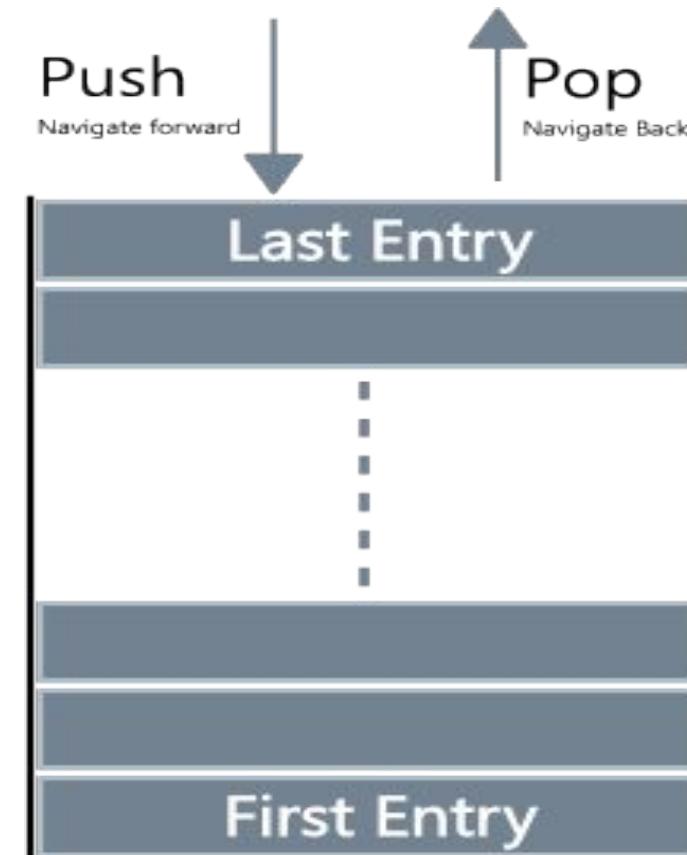


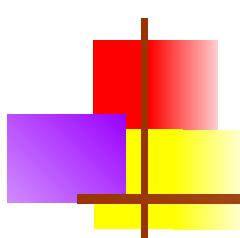
A stack of  
dvd/cd

**Fig: Realtime examples of Stack**

# Operations that can be performed on Stack

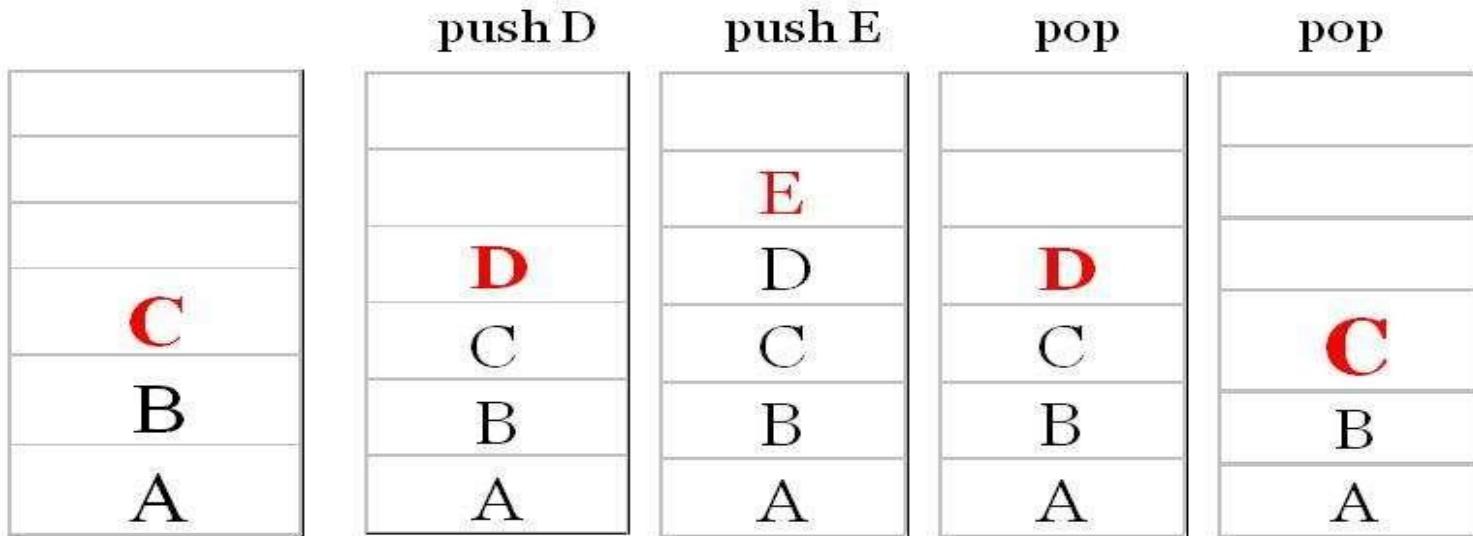
- ❑ Push
- ❑ Pop





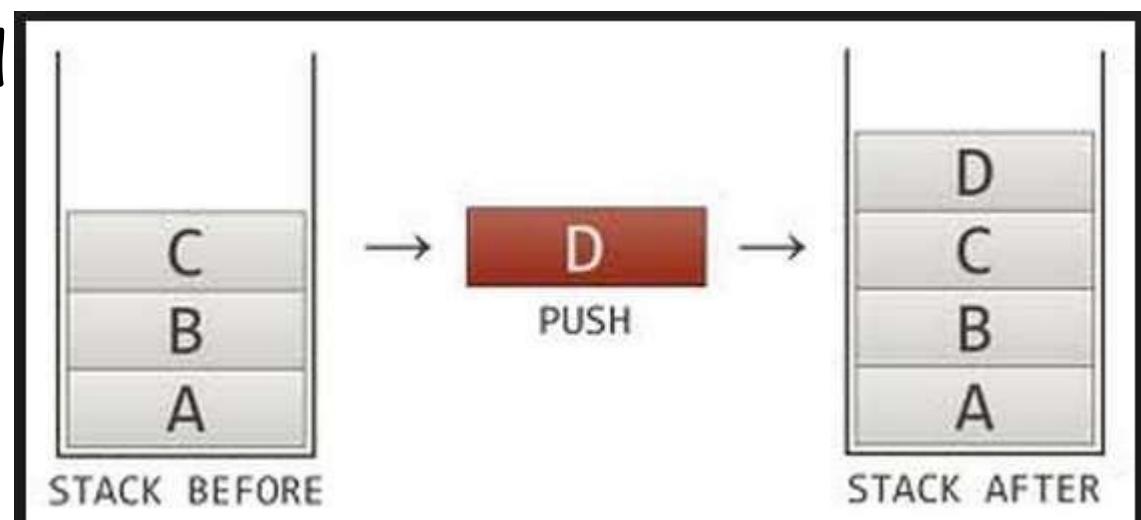
# Operations that can be performed on Stack

- ❑ Push: It is used to insert items into the stack
- ❑ PoP: It is used to delete items from stack
- ❑ Top: It represents the current location of data in stack.



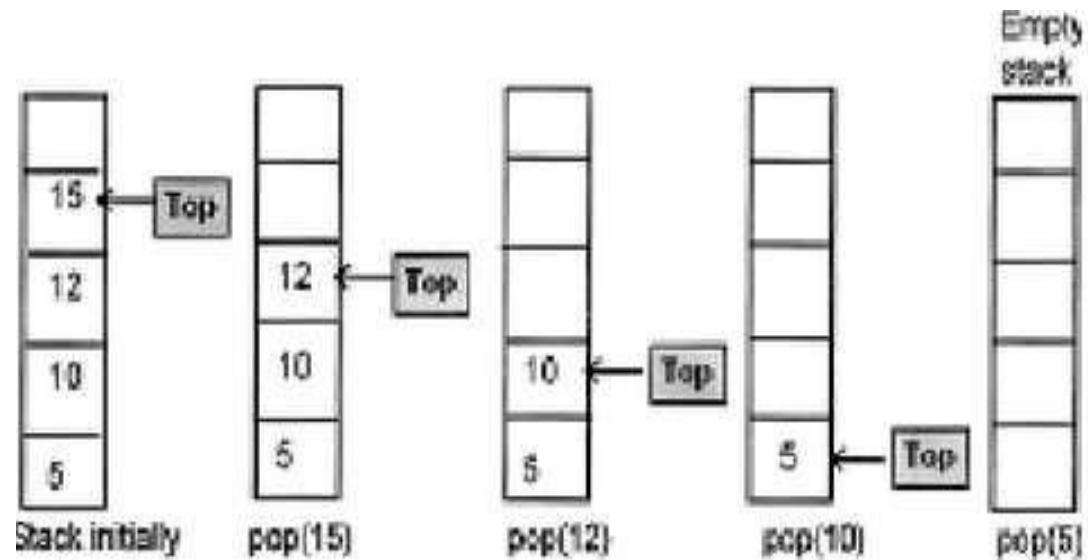
# Algorithm of Insertion in Stack: (Push)

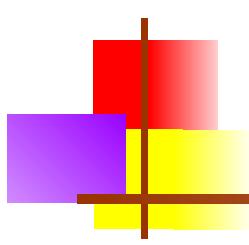
1. **Insertion(a,top,item,max)**
2. If  $\text{top}=\text{max}$  then  
print '**STACK OVERFLOW**'  
exit
- else
3.  **$\text{top}=\text{top}+1$**  end
4.  **$\text{a}[\text{top}]=\text{item}$**
5. Exit



# Algorithm of deletion in Stack: (PoP)

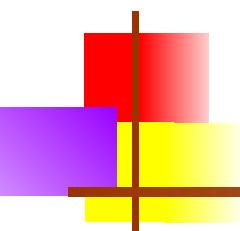
1. **Deletion(a,top,item)**
2. If  $\text{top}=0$  then  
print '**STACK UNDERFLOW**'  
exit
- else
3. **item=a[top]**
- end if
4. **top=top-1**
5. Exit





# Applications of Stacks are:

- ❑ Reversing Strings:
  - ❑ A simple application of stack is reversing strings.
  - ❑ To reverse a string, the characters of strings are pushed onto the stack one by one as the string is read from left to right.
  - ❑ Once all the characters of string are pushed onto stack, they are popped one by one. Since the character last pushed in comes out first, subsequent pop operation results in the reversal of the string.

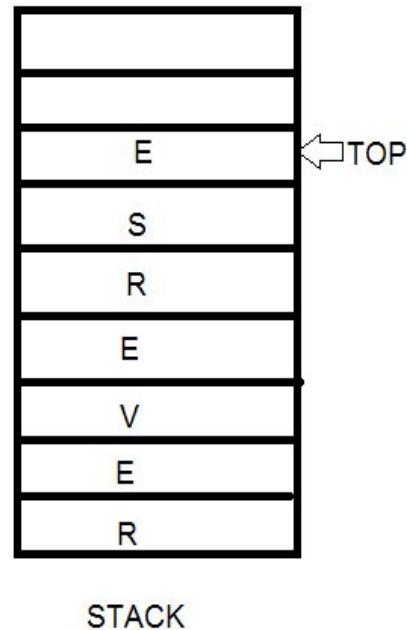


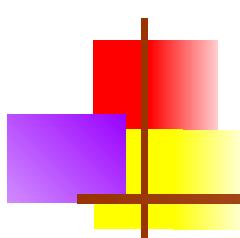
## For example:

- To reverse the string ‘REVERSE’ the string is read from left to right and its characters are pushed. Like:

STRING IS:

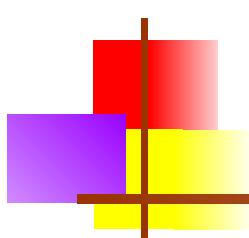
REVERSE





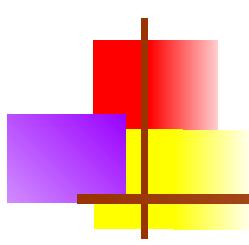
# Applications of Stacks are:

- ❑ Checking the validity of an expression containing nested parenthesis:
  - ❑ Stacks are also used to check whether a given arithmetic expressions containing nested parenthesis is properly parenthesized.
  - ❑ The program for checking the validity of an expression verifies that for each left parenthesis braces or bracket, there is a corresponding closing symbol and symbols are appropriately nested.



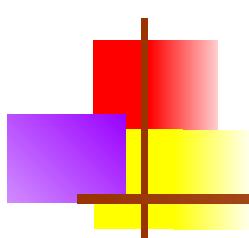
For example:

VALID INPUTS	INVALID INPUTS
{ }	{ ( }
( { [ ] } )	( [ ( ( ) ] )
{ [ ] ( ) }	{ } [ ] )
[ { ( { } [ ] ( {	[ { ) } ( ] } ]
} ) } ]	



# Applications of Stacks are:

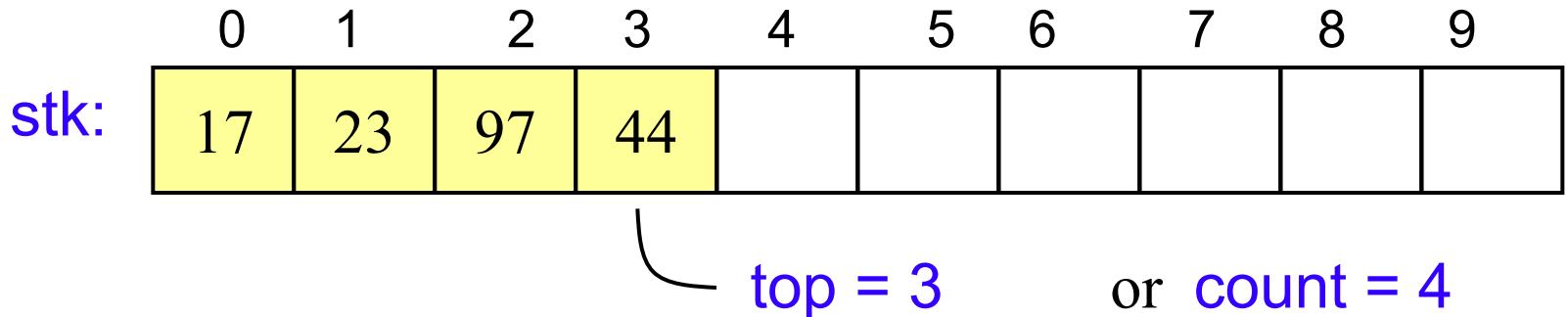
- ❑ Evaluating arithmetic expressions:
  - ❑ INFIX notation:
    - ❑ The general way of writing arithmetic expressions is known as infix notation. E.g,  $(a+b)$
  - ❑ PREFIX notation:
    - ❑ e.g,  $+AB$
  - ❑ POSTFIX notation:
    - ❑ e.g.  $AB+$



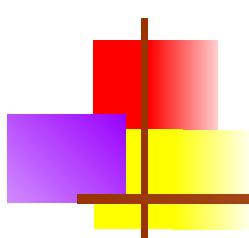
# Array implementation of stacks

- To implement a stack, items are inserted and removed at the same end (called the **top**)
- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end
- To use an array to implement a stack, you need both the array itself and an integer
  - The integer tells you either:
    - Which location is currently the top of the stack, or
    - How many elements are in the stack

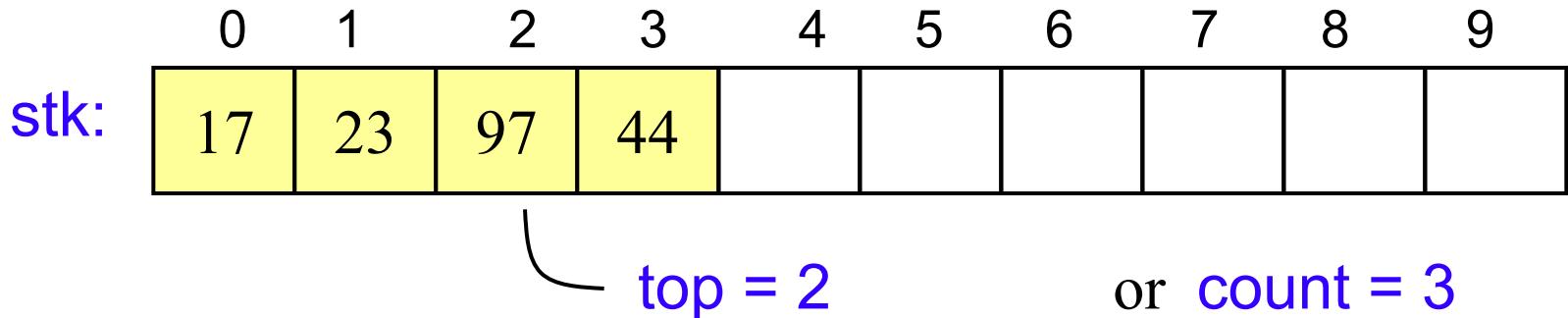
# Pushing and popping



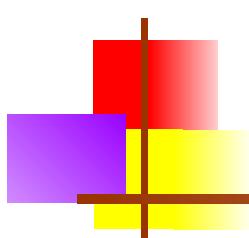
- If the **bottom** of the stack is at location 0, then an empty stack is represented by **top = -1** or **count = 0**
- To add (**push**) an element, either:
  - Increment **top** and store the element in **stk[top]**, or
  - Store the element in **stk[count]** and increment **count**
- To remove (**pop**) an element, either:
  - Get the element from **stk[top]** and decrement **top**, or
  - Decrement **count** and get the element in **stk[count]**



# After popping

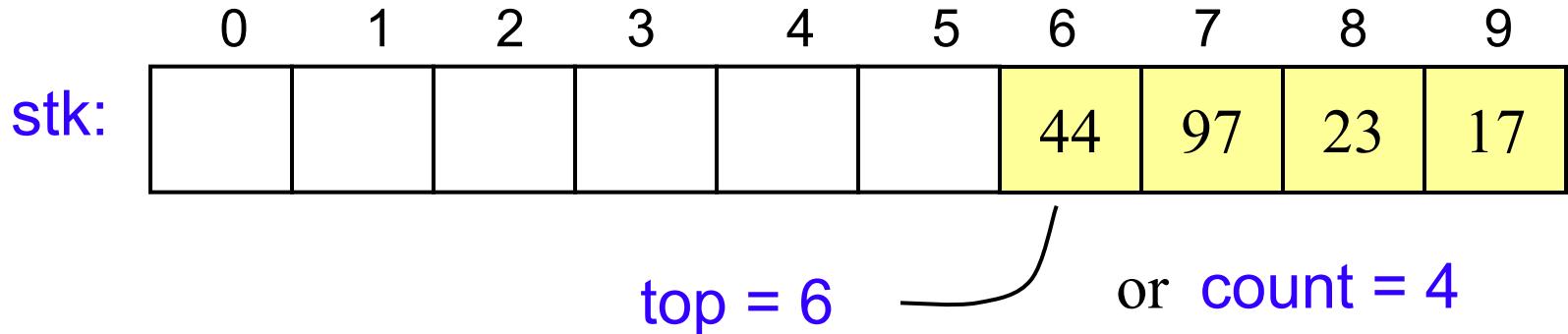


- When you pop an element, do you just leave the “deleted” element sitting in the array?
- The surprising answer is, “*it depends*”
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and the array contains objects, you should set the “deleted” array element to **null**
  - Why? To allow it to be garbage collected!

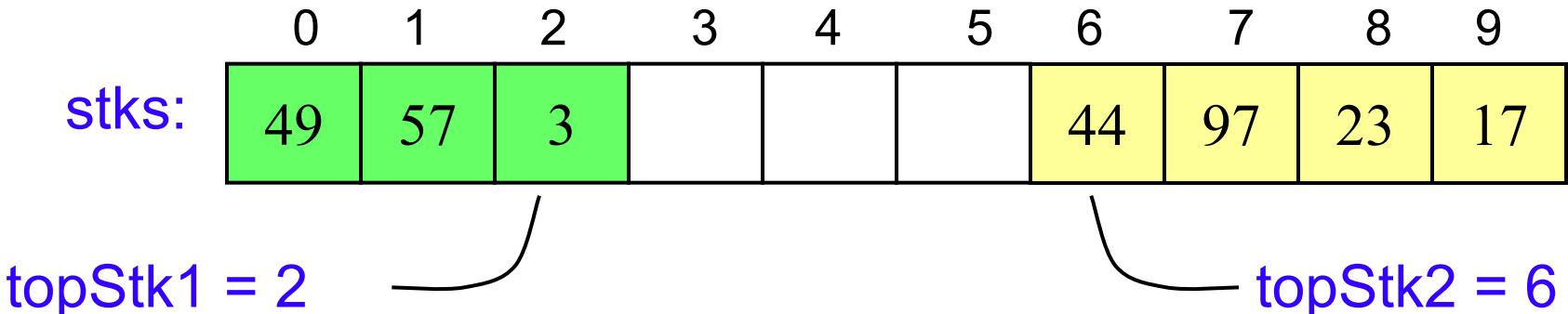


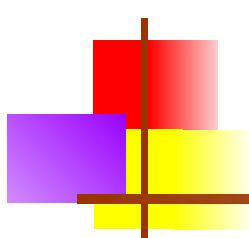
# Sharing space

- Of course, the bottom of the stack could be at the *other* end



- Sometimes this is done to allow two stacks to share the *same storage area*

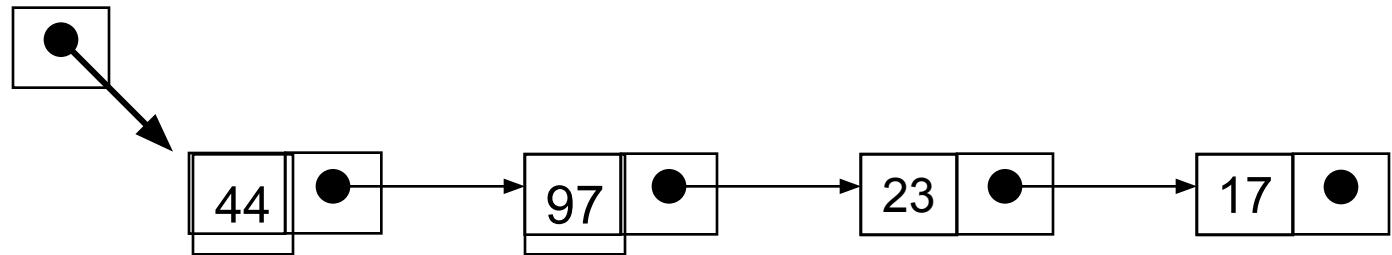




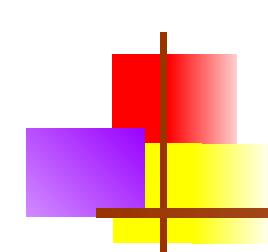
# Linked-list implementation of stacks

- Since all the action happens at the top of a stack, a singly-linked list (SLL) is a fine way to implement it
- The header of the list points to the top of the stack

myStack:

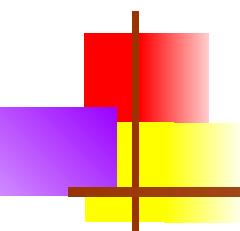


- Pushing is inserting an element at the front of the list
- Popping is removing an element from the front of the list



# Linked-list implementation details

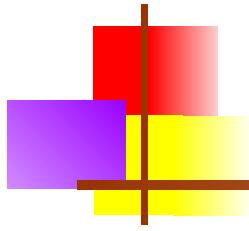
- With a linked-list representation, overflow will not happen (unless you exhaust memory, which is another kind of problem)
- Underflow can happen, and should be handled the same way as for an array implementation
- When a node is popped from a list, and the node references an object, the reference (the pointer in the node) does *not* need to be set to **null**
  - Unlike an array implementation, it really *is* removed--you can no longer get to it from the linked list
  - Hence, garbage collection can occur as appropriate



# Queues

---

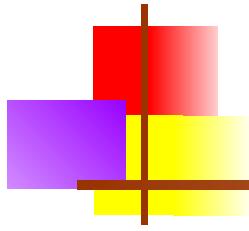
- ❑ Queue is an ADT data structure similar to stack except that the first item to be inserted is the first one to be removed.
- ❑ This mechanism is called First-in-First-Out (FIFO).
- ❑ Placing an item in a queue is called “insertion or enqueue”, which is done at the end of the queue called “rear”.
- ❑ Removing an item from the queue is called “deletion or dequeue”, which is done at the other end of the queue called “front”.
- ❑ Some of the applications are :printer queue, keystroke queue, etc.



# Operations on a Queue

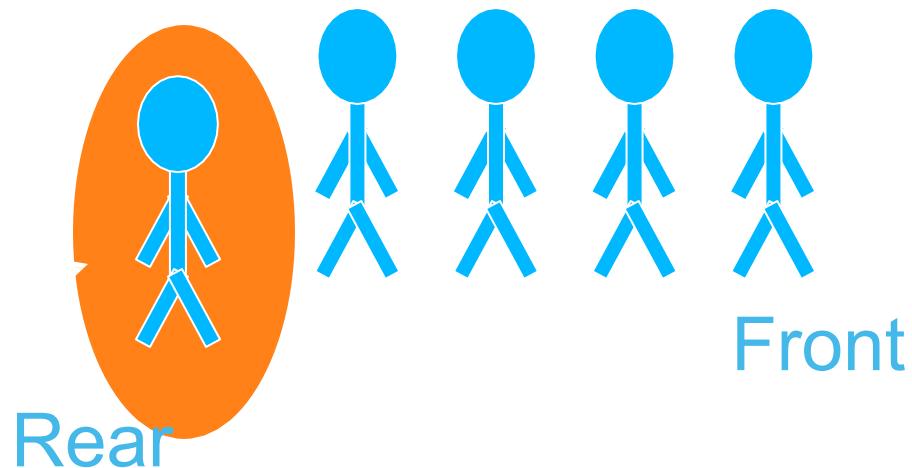
---

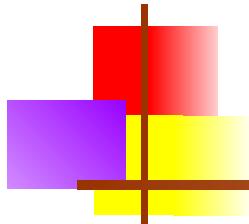
- ❑ To insert an element in queue
- ❑ Delete an element from queue



# The Queue Operation

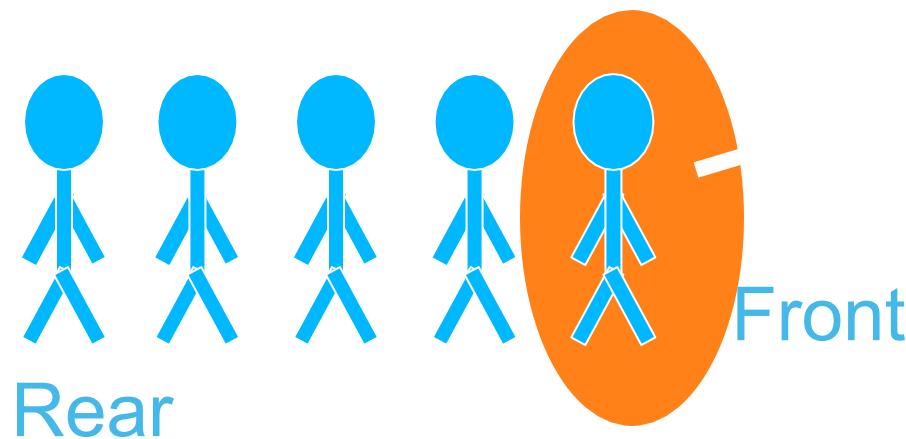
- Placing an item in a queue is called “insertion or **enqueue**”, which is done at the end of the queue called “**rear**”.

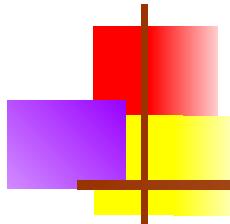




# The Queue Operation

- Removing an item from a queue is called “deletion or **dequeue**”, which is done at the other end of the queue called “**front**”.





# Algorithm Qinsert (Item)

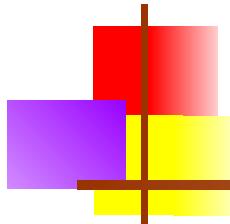
1. If( $\text{rear} = \text{maxsize}-1$  )

print (“queue overflow”) and return

2. Else

$\text{rear} = \text{rear} + 1$  Queue

[ $\text{rear}$ ] = item



# Algorithm Qdelete ()

1. If (front =rear)

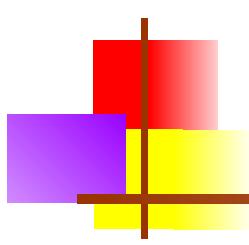
    print “queue empty” and return

2. Else

    Front = front + 1    item

    = queue [front];

Return item



# States of the Queue

---

1. Queue is empty

**FRONT=REAR**

1. Queue is full

**REAR=N**

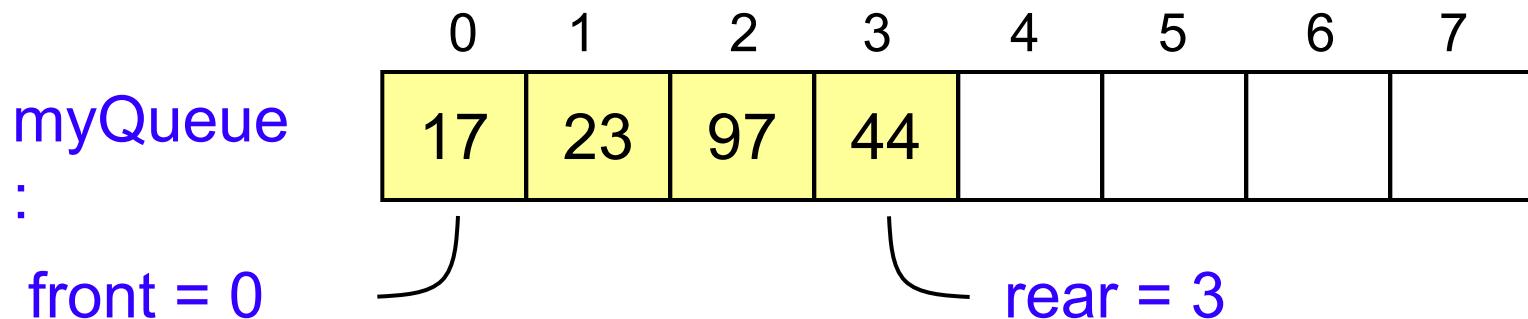
1. Queue contains element  $\geq 1$

**FRONT<REAR**

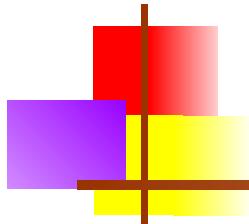
**NO. OF ELEMENT=REAR-FRONT+1**

# Array implementation of queues

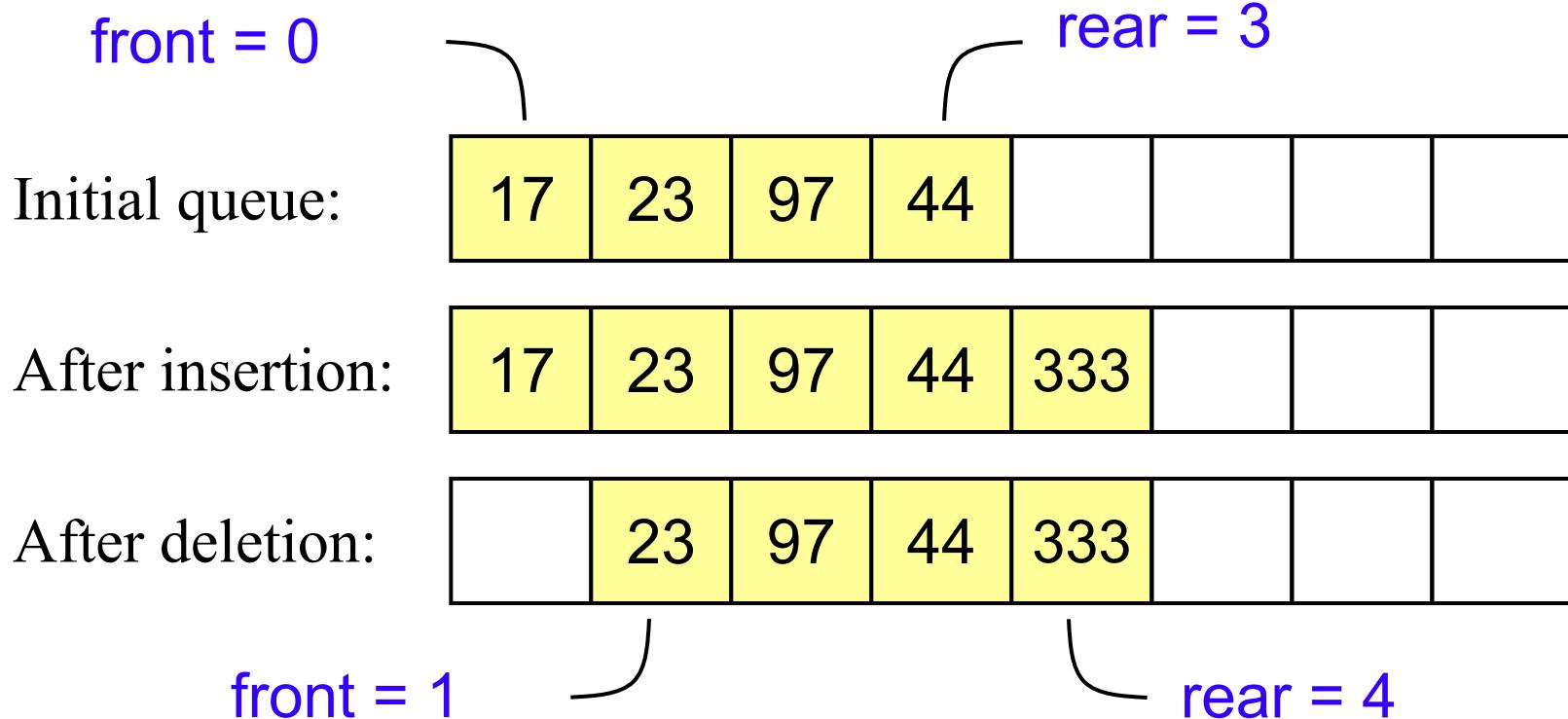
- A **queue** is a first in, first out (**FIFO**) data structure
- This is accomplished by inserting at one end (the **rear**) and deleting from the other (the **front**)



- **To insert:** put new element in location **4**, and set **rear** to **4**
- **To delete:** take element from location **0**, and set **front** to **1**



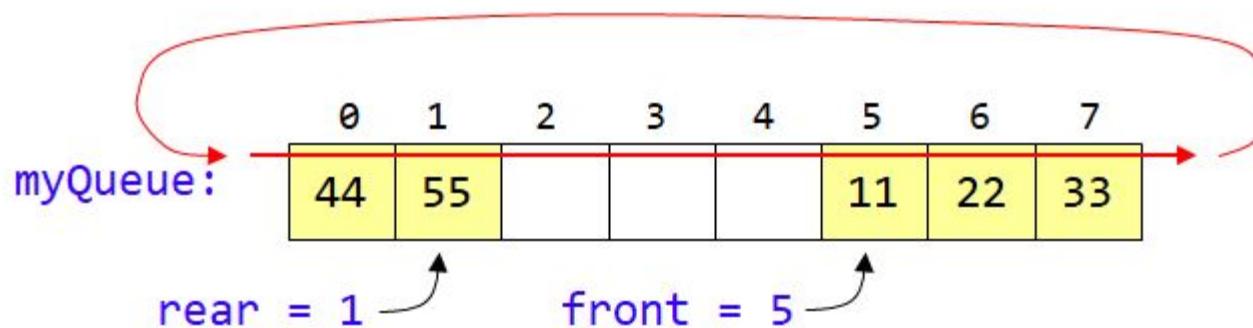
# Array implementation of queues



- Notice how the array contents “crawl” to the right as elements are inserted and deleted
- This will be a problem after a while!

# Circular arrays

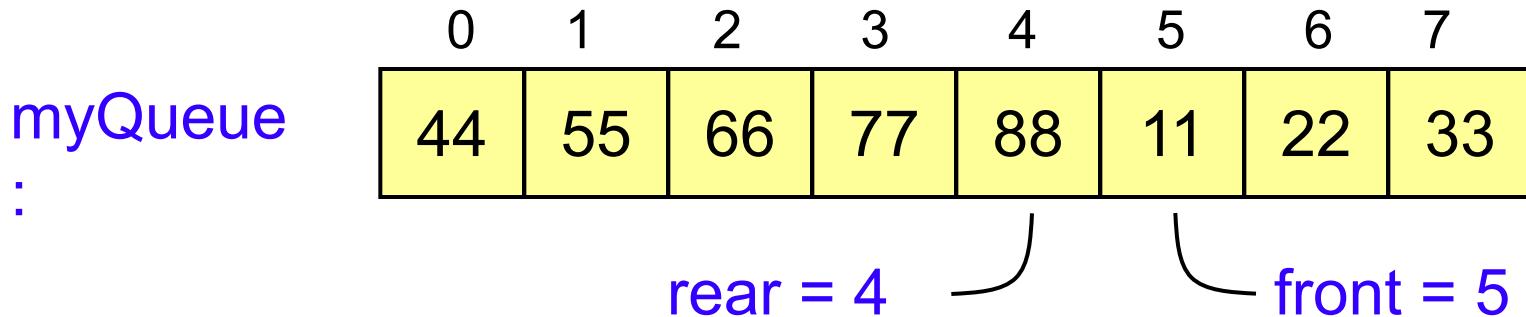
- We can treat the array holding the queue elements as circular (joined at the ends)



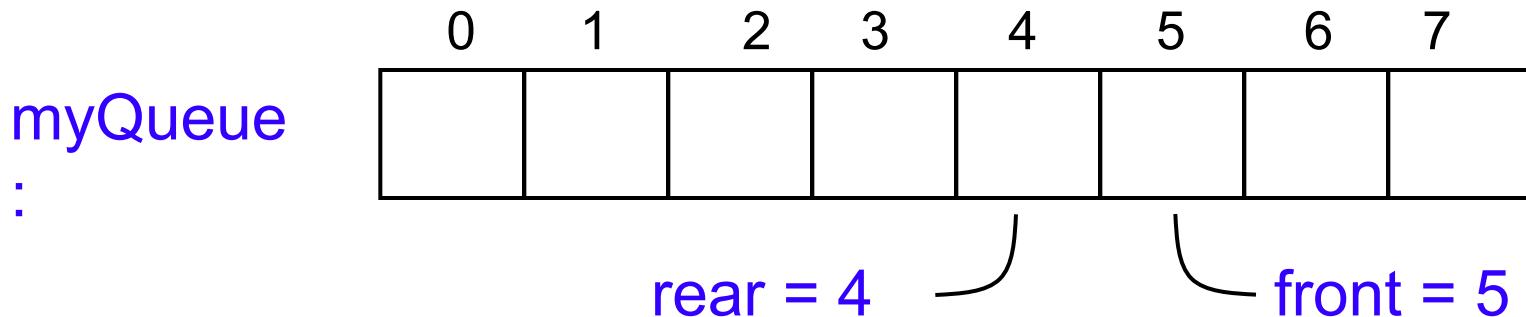
- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order
- Use: `front = (front + 1) % myQueue.length;`  
and: `rear = (rear + 1) % myQueue.length;`

# Full and empty queues

- If the queue were to become completely full, it would look like this:



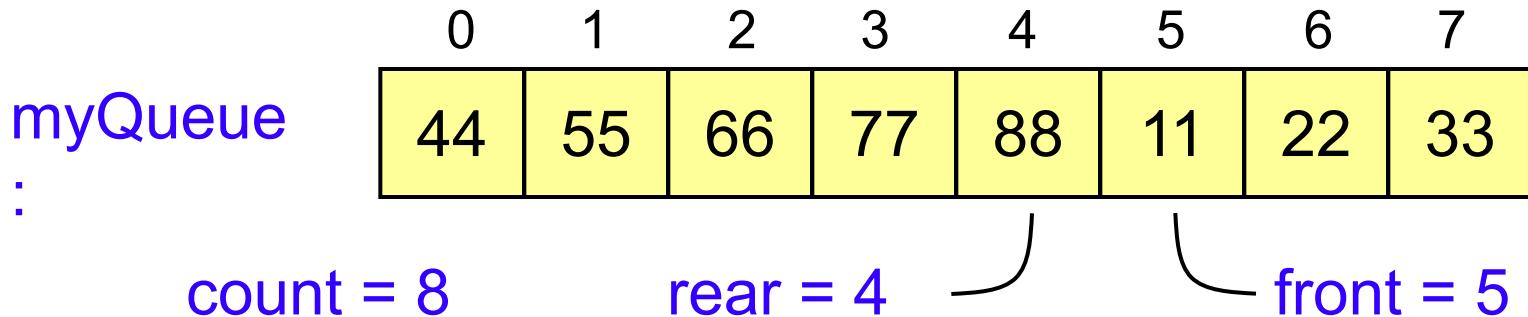
- If we were then to remove all eight elements, making the queue completely empty, it would look like this:



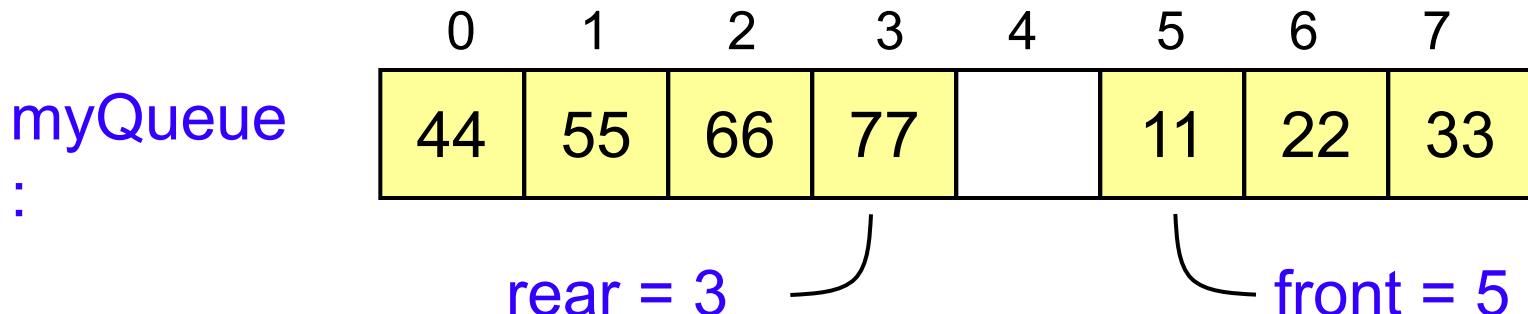
This is a problem!

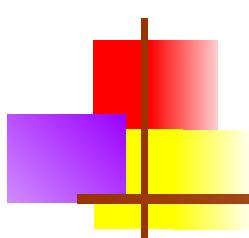
# Full and empty queues: solutions

- **Solution #1:** Keep an additional variable



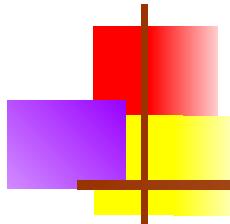
- **Solution #2:** (Slightly more efficient) Keep a gap between elements: consider the queue full when it has  $n-1$  elements





# Linked-list implementation of queues

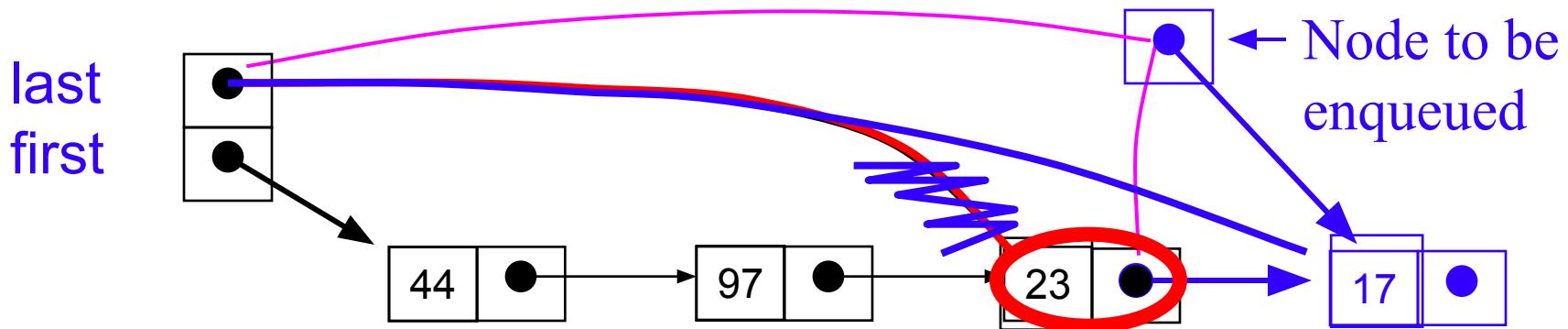
- In a queue, insertions occur at one end, deletions at the other end
- Operations at the front of a singly-linked list (SLL) are  $O(1)$ , but at the other end they are  $O(n)$ 
  - Because you have to find the last element each time
- BUT: there is a simple way to use a singly-linked list to implement both insertions and deletions in  $O(1)$  time
  - You always need a pointer to the first thing in the list
  - You can keep an additional pointer to the *last* thing in the list



# SLL implementation of queues

- In an SLL you can easily find the successor of a node, but not its predecessor
  - Remember, pointers (references) are one-way
- If you know where the *last* node in a list is, it's hard to remove that node, but it's easy to add a node after it
- Hence,
  - Use the *first* element in an SLL as the *front* of the queue
  - Use the *last* element in an SLL as the *rear* of the queue
  - Keep pointers to *both* the front and the rear of the SLL

# Enqueueing a node



To **enqueue** (add) a node:

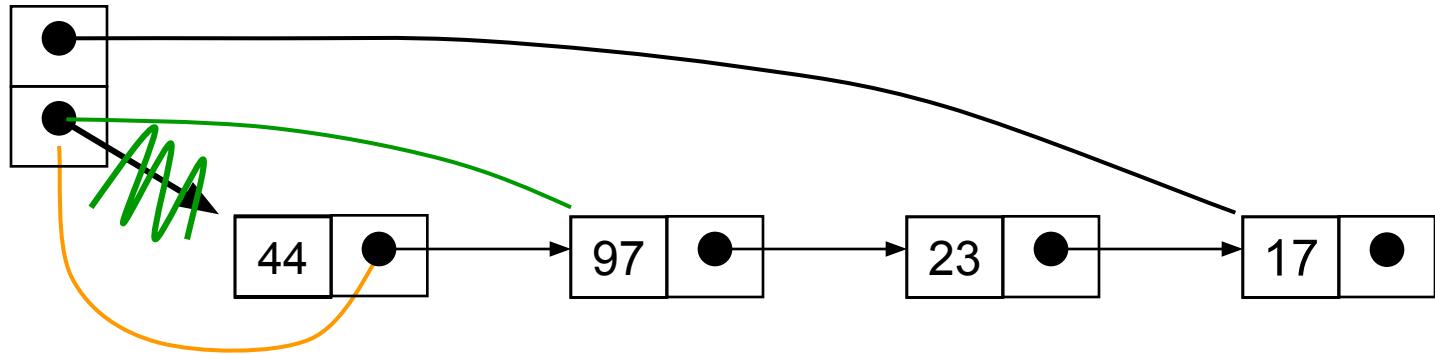
Find the current last node

Change it to point to the new last node

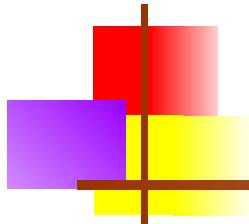
Change the **last** pointer in the list header

# Dequeuing a node

last  
first

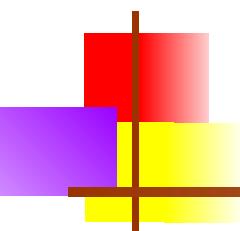


- To **dequeue** (remove) a node:
  - Copy the pointer from the first node into the header



# Queue implementation details

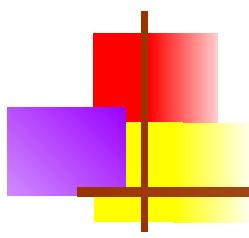
- With an array implementation:
  - you can have both overflow and underflow
  - you should set deleted elements to **null**
- With a linked-list implementation:
  - you can have underflow
  - overflow is a global out-of-memory condition
  - there is no reason to set deleted elements to **null**



# Deques

---

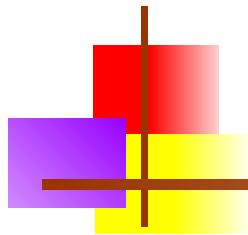
- A **deque** is a double-ended queue
- Insertions *and* deletions can occur at *either* end
- Implementation is similar to that for queues
- Deques are not heavily used
- You should know what a deque is, but we won't explore them much further

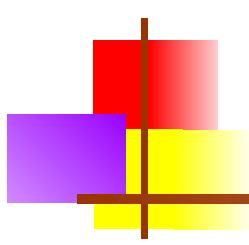


# The End

---

# Searching and Sorting

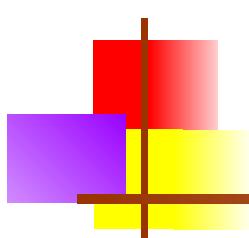




# What is Searching?

---

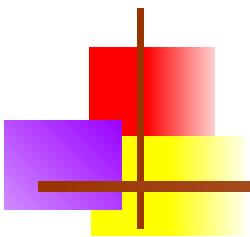
- ❑ In computer science, **searching** is the process of finding an item with **specified properties** from a collection of items.
- ❑ The items may be sorted as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs, or may be the elements in other search place.
- ❑ The definition of a searching is the process of **looking** for something or someone.



# Why do we need searching?

---

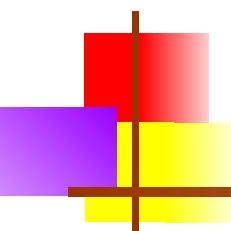
- ❑ Searching is one of the core computer science algorithms.
- ❑ We know that today's computers store a lot of information.
- ❑ To retrieve information proficiently we need very efficient searching algorithms.
- ❑ **Type of Searching**
  - ❑ Linear Search
  - ❑ Binary Search



# Linear Search

---

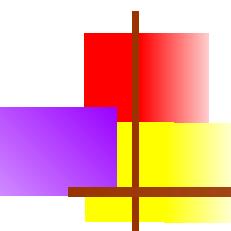




# Linear Search

---

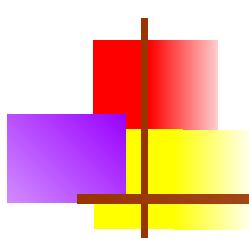
- ❑ The linear search is a sequential search, which uses a loop to step through an array, starting with the first element.
- ❑ It compares each elements with the value being searched for, and stops when either the value is found or the end of the array is encountered.
- ❑ If the value being searched is not in the array, the algorithm will unsuccessfully search to the end of the array.



# Linear Search

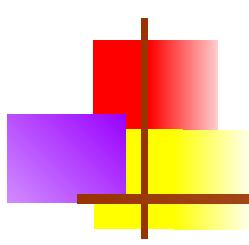
---

- ❑ Since the array elements are stored in linear order searching the element in the linear order make it easy and efficient.
  
- ❑ The search may be successful or unsuccessfully. That is, if the required element is found them the search is successful otherwise it is unsuccessfully.



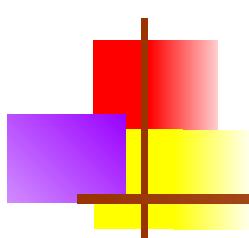
# Unordered Linear/ Sequential search

```
□ int unorderedlinearsearch (int A[], int n, int  
    data)  
□ {  
□   for (int i=0; i<n; i++)  
□   {  
□     □ if(A[i] == data)  
□       □ return i;  
□   }  
□   return -1;  
□ }
```



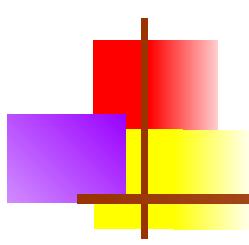
# Advantages of Linear Search

- ❑ If the first number in the directory is the number you were searching for, then lucky you!!
- ❑ Since you have found it on the very first page, now its not important for you that how many pages are there in the dictionary.
- ❑ The linear search is simple- it is very easy to understand and implement.
- ❑ It does not require the data in the array to be stored in any particular order.
- ❑ So it does not depends on number of elements in the directory. Hence constant time  $O(1)$ .



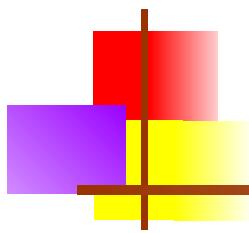
# Disadvantages of Linear Search

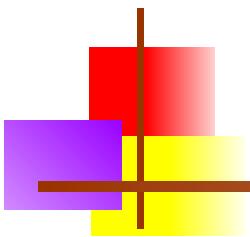
- ❑ It may happen that the number you are searching for is the last number of directory or if it is not in the directory at all.
- ❑ In that case you have to search the whole directory.
- ❑ Now number of elements will matter to you. If there are 500 pages, you have to search 500; if it has 1000 you have to search 1000.
- ❑ your search time is proportional to number of elements in the directory.
- ❑ it has very less efficiency because it takes lots of comparisons to find a particular record in big files
- ❑ The performance of the algorithm scales linearly with the size of the input
- ❑ Linear search is slower than other searching algorithms



# Analysis of Linear Search

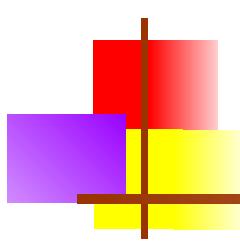
- In the **best case**, the target value is in the first element of the array. So the search takes constant amount of time.
- In the **worst case**, the target value is in the last element of the array. So the search takes an amount of time proportional to the length of the array,i.e.,  $O(n)$ .
- In the **average case**, the target value is somewhere in the array. So on average, the target value will be in the middle of the array. So the search takes an amount of time proportional to the length of the array, i.e.,  $O(n)$ .
-





# Binary Search





# Binary Search

- The general term for a smart search through sorted data is a binary search
  - 1. The initial search is the whole array
  - 2. Look at the data value in the **middle** of the search region.
  - 3. If you've found your target, **stop**
  - 4. If your target is **less than** the middle data value, the new search region is the **lower half** of the data.
  - 5. If your target is **greater than** the middle data value, the new search region is the **higher half** of the data.
  - 6. Continue from **Step 2.**



# Binary Search

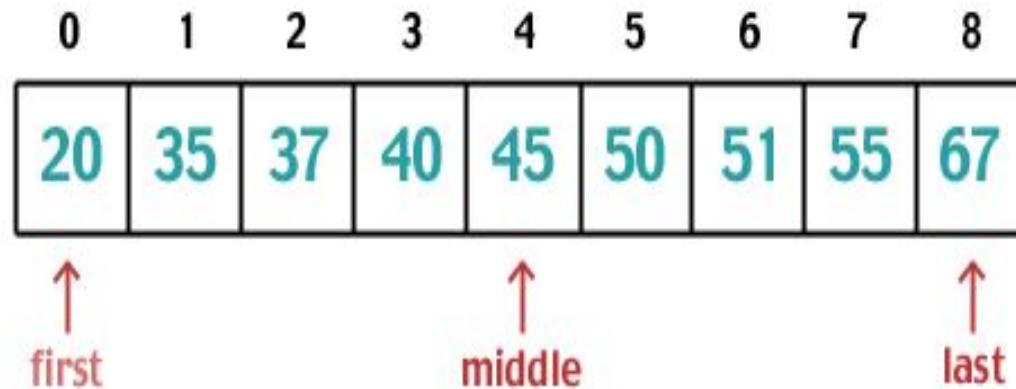
Given Array

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

Searching element-37

# Binary Search

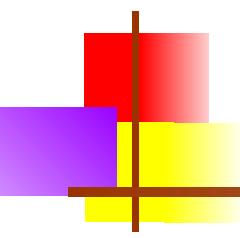
Step-2 Calculate middle=(low+high)/2=(0+8)/2=4



If  $37 == \text{array}[\text{middle}] \rightarrow \text{return middle}$

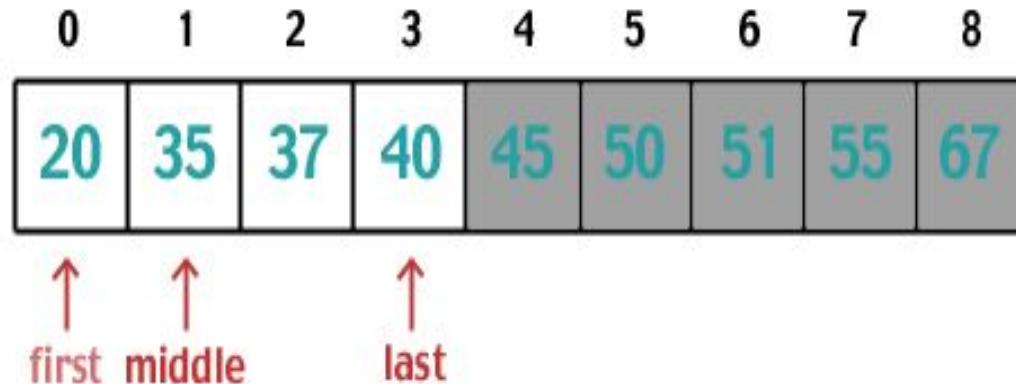
Else if  $37 < \text{array}[\text{middle}] \rightarrow \text{high} = \text{middle} - 1$

Else if  $37 > \text{array}[\text{middle}] \rightarrow \text{low} = \text{middle} + 1$



# Binary Search

Repeat Step-2 Calculate middle=(low+high)/2=(0+3)/2=1



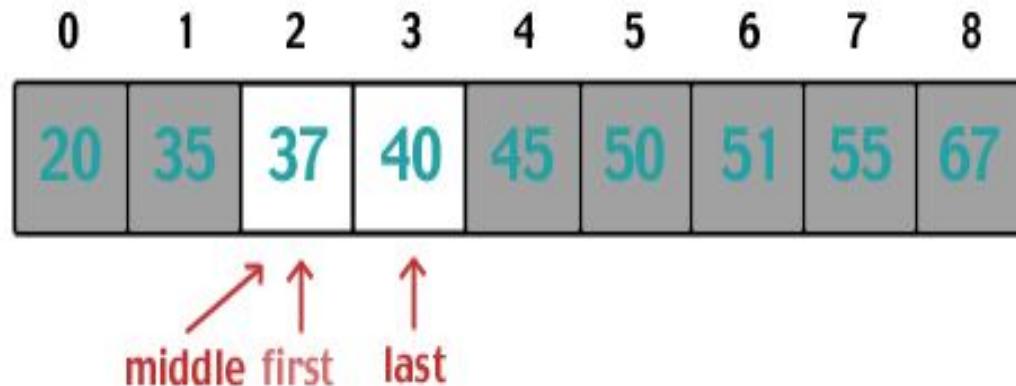
If  $37 == \text{array}[\text{middle}] \rightarrow \text{return } \text{middle}$

Else if  $37 < \text{array}[\text{middle}] \rightarrow \text{high} = \text{middle} - 1$

Else if  $37 > \text{array}[\text{middle}] \rightarrow \text{low} = \text{middle} + 1$

# Binary Search

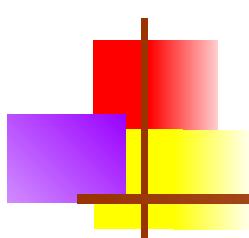
Repeat Step-2 Calculate middle=(low+high)/2=(2+3)/2=2



If  $37 == \text{array}[\text{middle}] \rightarrow \text{return middle}$

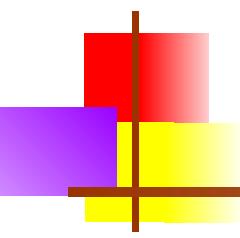
Else if  $37 < \text{array}[\text{middle}] \rightarrow \text{high} = \text{middle} - 1$

Else if  $37 > \text{array}[\text{middle}] \rightarrow \text{low} = \text{middle} + 1$



# Binary Search Performance

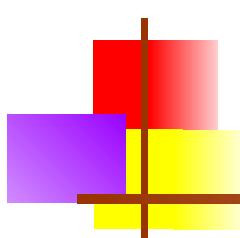
- ❑ Successful search
  - ❑ Best Case-1 Comparison
  - ❑ Worst Case- $\log N$  comparison
- ❑ Unsuccessful search
  - ❑ Best case=Worst case- $\log N$  comparisons
- ❑ Since the portion of an array to search is cut into half after every comparison, we compute how many times the array can be divided into halves.
- ❑ After  $K$  comparisons, there will be  $N/2^K$  elements in the list. We solve for  $K$  when  $N/2^K = 1$  deriving  $K = \log_2 N$



# Binary Search

```
 // C program to implement iterative Binary Search
> #include <stdio.h>
>
> // A iterative binary search function. It returns
> // location of x in given array arr[l..r] if present,
> // otherwise -1
> int binarySearch(int arr[], int l, int r, int x)
> {
>     while (l <= r) {
>         int m = l + (r - l) / 2;
>
>         // Check if x is present at mid
>         if (arr[m] == x)
>             return m;
>
>         // If x greater, ignore left half
>         if (arr[m] < x)
>             l = m + 1;
>
>         // If x is smaller, ignore right half
>         else
>             r = m - 1;
>     }
>
>     // If we reach here, then element was
>     // not present
>     return -1;
> }
>
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present"
                           " in array")
                  : printf("Element is present at "
                           "index %d",
                           result);
    return 0;
}
```

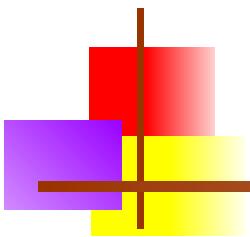
Source:<https://www.geeksforgeeks.org/binary-search/>



# Important Differences

---

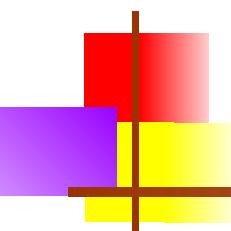
- ❑ Input data needs to be sorted in Binary Search and not in Linear Search
- ❑ Linear search does the sequential access whereas Binary search access data randomly.
- ❑ Time complexity of linear search - $O(n)$  , Binary search has time complexity  $O(\log n)$ .
- ❑ Linear search performs equality comparisons and Binary search performs ordering comparisons



# Bubble Sort

---

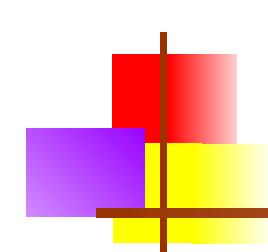




# Bubble Sort

---

- ❑ Bubble sort is a simple sorting algorithm
- ❑ This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order
- ❑ This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$  where n is the number of items.

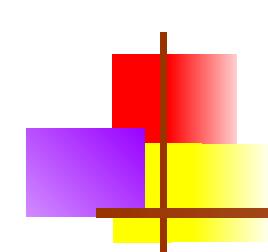


# How Bubble Sort Works ?

- We take an unsorted array for our example. Bubble sort takes  $O(n^2)$  time so we're keeping it short and precise.



- Bubble sort starts with very first two elements, comparing them to check which one is greater



# How Bubble Sort Works ?



In this case, value 33 is greater than 14, so it is already in sorted locations.  
Next, we compare 33 with 27.



We find that 27 is smaller than 33 and these two values must be swapped.

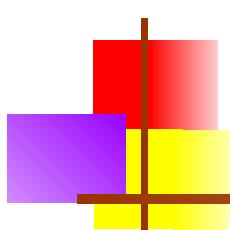


The new array should look like this –



Next we compare 33 and 35. We find that both are in already sorted positions.





# How Bubble Sort Works ?

Then we move to the next two values, 35 and 10.



We know then that 10 is smaller than 35. Hence they are not sorted.



We swap these values. We find that we have reached the end of the array.  
After one iteration, the array should look like this –



To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –



Notice that after each iteration, at least one value moves at the end.



# How Bubble Sort Works ?

And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

## Algorithm

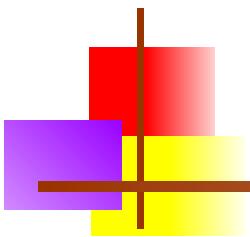
We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

    for all elements of list
        if list[i] > list[i+1]
            swap(list[i], list[i+1])
        end if
    end for

    return list

end BubbleSort
```



# Insertion Sort

---



# Insertion Sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

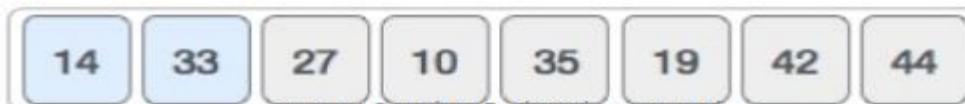
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$ , where **n** is the number of items.

## How Insertion Sort Works?

We take an unsorted array for our example.



Insertion sort compares the first two elements.



# Insertion Sort

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



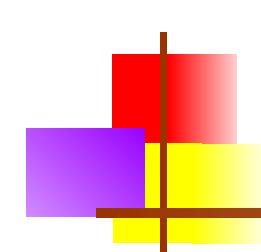
And finds that 33 is not in the correct position.



It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



# Insertion Sort



These values are not in a sorted order.



So we swap them.



However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.

# Insertion Sort



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

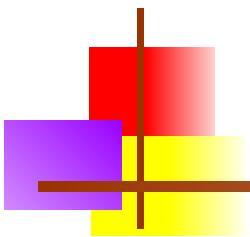


This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

## Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

- ```
Step 1 - If it is the first element, it is already sorted. return 1;  
Step 2 - Pick next element  
Step 3 - Compare with all elements in the sorted sub-list  
Step 4 - Shift all the elements in the sorted sub-list that is greater than the  
        value to be sorted  
Step 5 - Insert the value  
Step 6 - Repeat until list is sorted
```



# Selection Sort

---



# Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

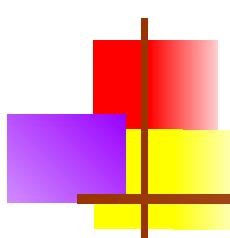
This algorithm is not suitable for large data sets as its average and worst case complexities are of  $O(n^2)$ , where  $n$  is the number of items.

## How Selection Sort Works?

Consider the following depicted array as an example.



For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



# Selection Sort



So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



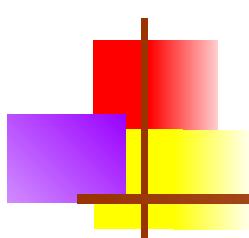
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.

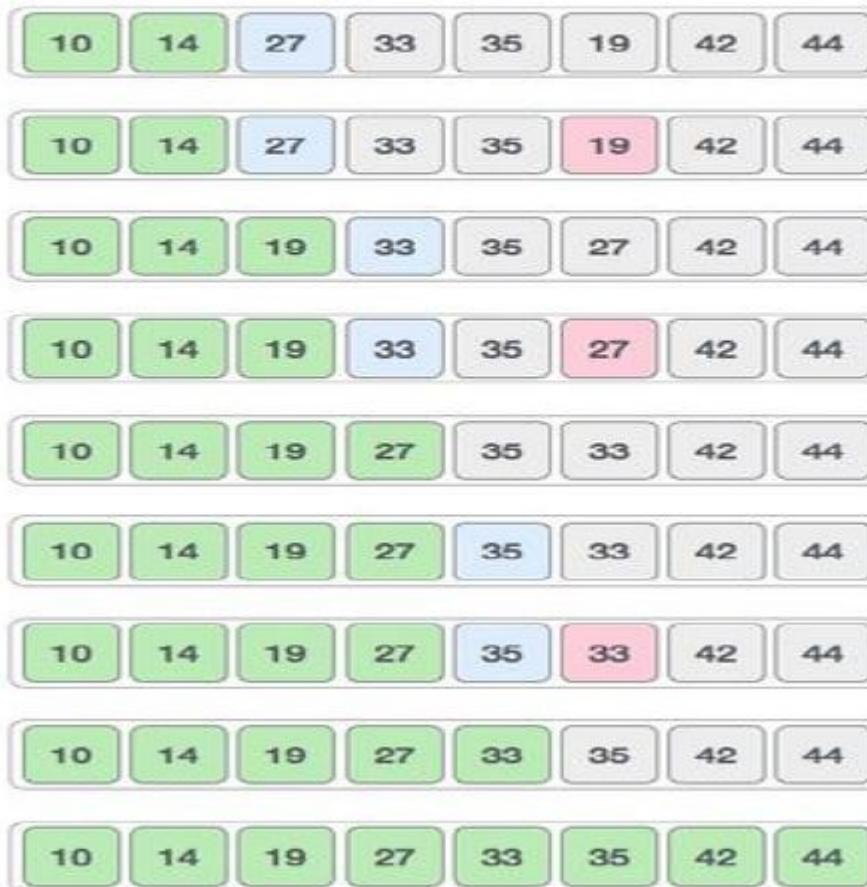


# Selection Sort



The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process –



# Selection Sort

## Algorithm

```
Step 1 - Set MIN to location 0
Step 2 - Search the minimum element in the list
Step 3 - Swap with value at location MIN
Step 4 - Increment MIN to point to next element
Step 5 - Repeat until list is sorted
```

## Pseudocode

```
procedure selection sort
    list : array of items
    n     : size of list

    for i = 1 to n - 1
        /* set current element as minimum*/
        min = i

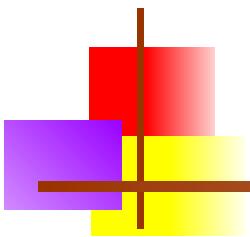
        /* check the element to be minimum */

        for j = i+1 to n
            if list[j] < list[min] then
                min = j;
            end if
        end for

        /* swap the minimum element with the current element*/
        if indexMin != i then
            swap list[min] and list[i]
        end if

    end for

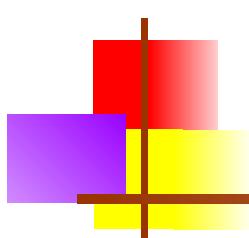
end procedure
```



# Special Types of Trees

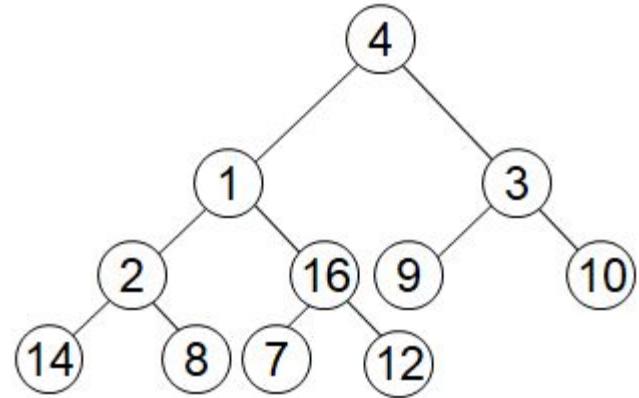
---





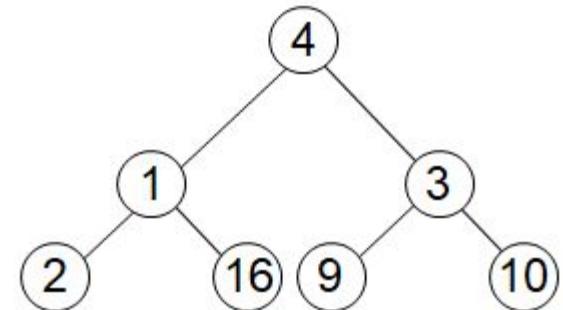
# Full and Complete Binary Trees

- **Full Binary Tree:** a binary tree in which each node is either a leaf or has degree exactly 2.



Full binary tree

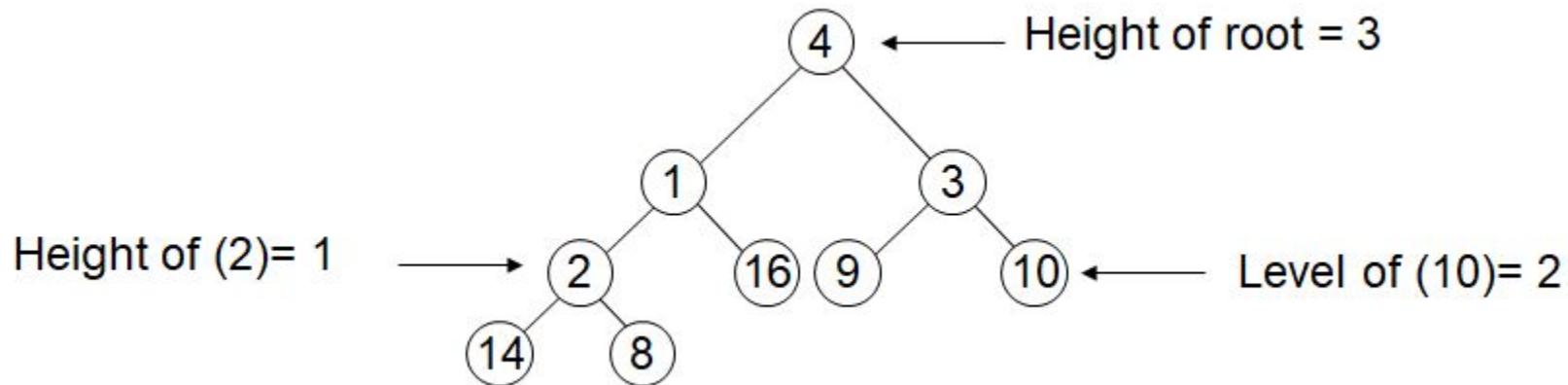
- **Complete Binary Tree:** a binary tree in which all leaves are on the same level and all internal nodes have degree 2.



Complete binary tree

# Definitions

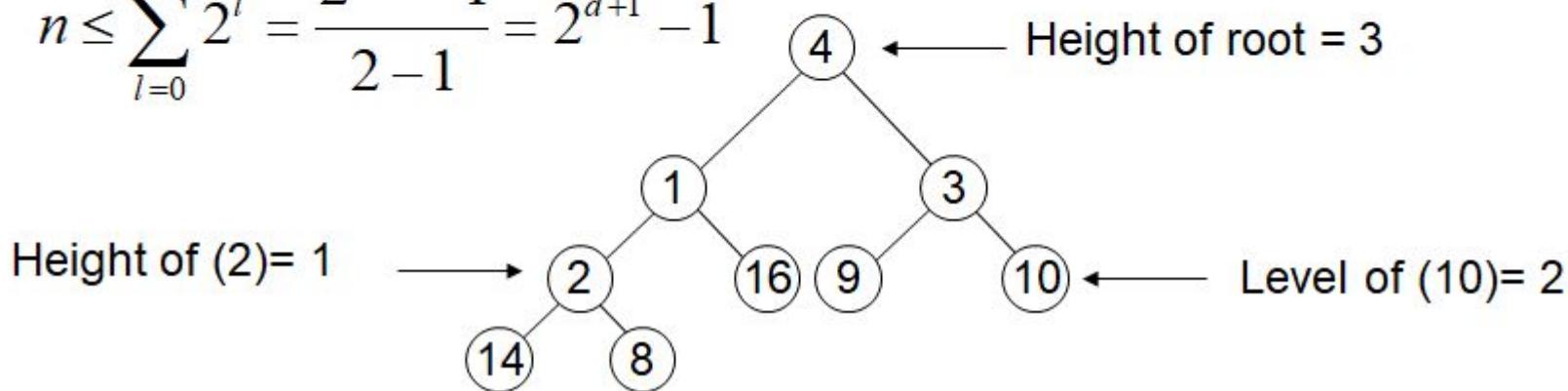
- **Height of a node:** the number of edges on the longest simple path from the node down to a leaf
- **Level of a node:** the length of a path from the root to the node
- **Height of tree:** height of root node

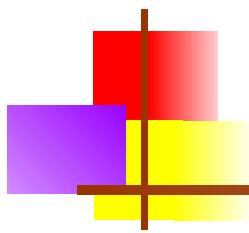


# Useful Properties

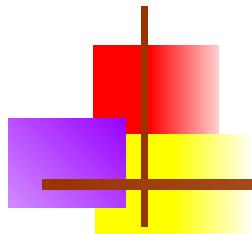
- There are **at most**  $2^l$  nodes at level (or depth)  $l$  of a binary tree
- A binary tree with **height**  $d$  has **at most**  $2^{d+1} - 1$  nodes
- A binary tree with  $n$  nodes has **height at least**  $\lfloor \lg n \rfloor$   
(see Ex 6.1-2, page 129)

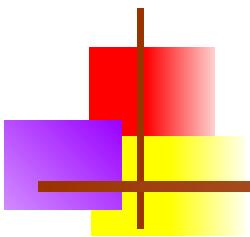
$$n \leq \sum_{l=0}^d 2^l = \frac{2^{d+1} - 1}{2 - 1} = 2^{d+1} - 1$$





# Heap Sort





# Heap

---

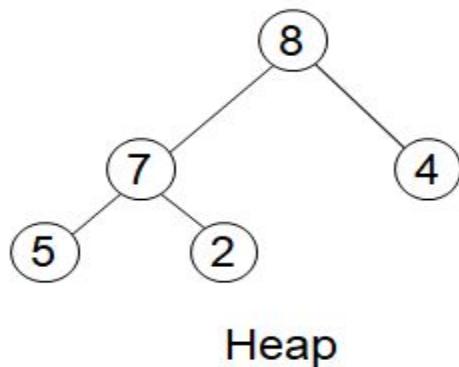


# The Heap Data Structure

A heap is a nearly complete binary tree with the following two properties:

- **Structural Property:** all levels are full, except possibly the last one, which is filled from left to right
- **Order (heap) property:** for any node x

$$\text{Parent}(x) \geq x$$

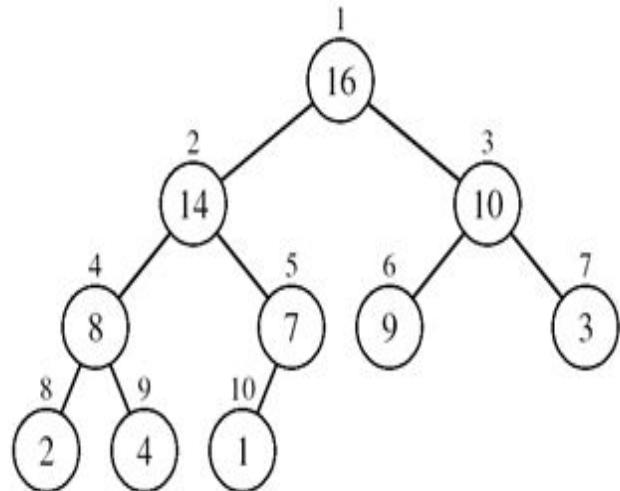
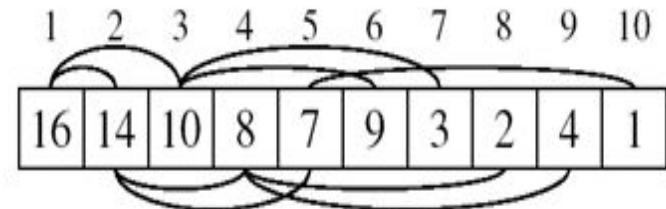


From the heap property, it follows that:  
“The root is the maximum element of the heap!”

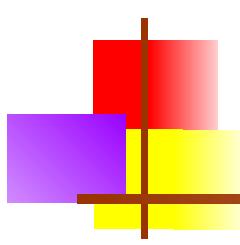
A heap is a binary tree that is filled in order

# Array Representation of Heaps

- A heap can be stored as an array  $A$ 
  - Root of tree is  $A[1]$
  - Left child of  $A[i] = A[2i]$
  - Right child of  $A[i] = A[2i + 1]$
  - Parent of  $A[i] = A[i/2]$
  - Heapsize[A]  $\leq$  length[A]
- The elements in the subarray  $A[(n/2+1) .. n]$  are leaves



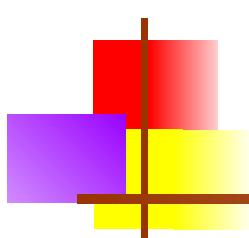
]



# Heap Types

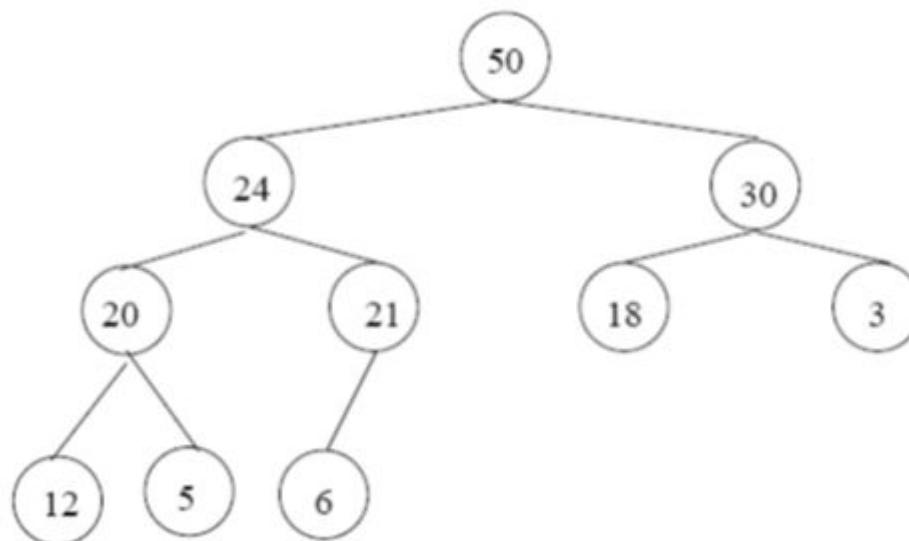
---

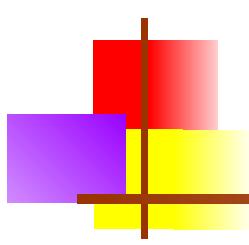
- **Max-heaps** (largest element at root), have the *max-heap property*:
  - for all nodes  $i$ , excluding the root:
$$A[\text{PARENT}(i)] \geq A[i]$$
- 
- **Min-heaps** (smallest element at root), have the *min-heap property*:
  - for all nodes  $i$ , excluding the root:
$$A[\text{PARENT}(i)] \leq A[i]$$



# Adding/ Deleting Nodes

- ❑ New nodes are always inserted at the bottom level (left to right)
- ❑ Nodes are removed from the bottom level (right to left)

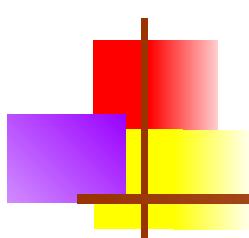




# Operations on Heaps

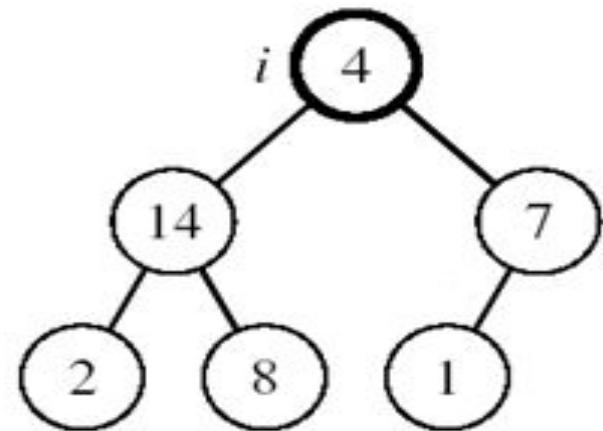
---

- ❑ Maintain/ Restore the max-heap property
  - ❑ MAX-heapify
- ❑ Create a max-heap from an unordered array
  - ❑ Build-MAX-Heap
- ❑ Sort an array in Place
  - ❑ HEAPSORT
- ❑ Priority queues



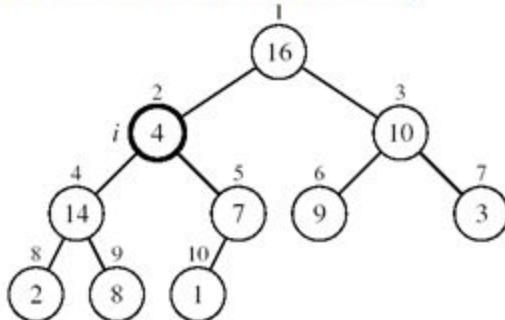
# Maintaining the Heap Property

- ❑ Suppose a node is smaller than a child
  - ❑ Left and Right subtrees of  $i$  are max-heaps
- ❑ To eliminate the violation:
  - ❑ Exchange with larger child
  - ❑ Move down the tree
  - ❑ Continue until node is not smaller than children



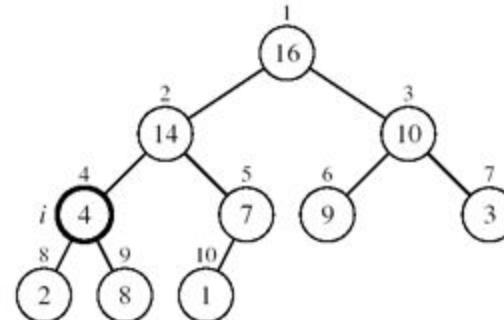
# Example

MAX-HEAPIFY(A, 2, 10)



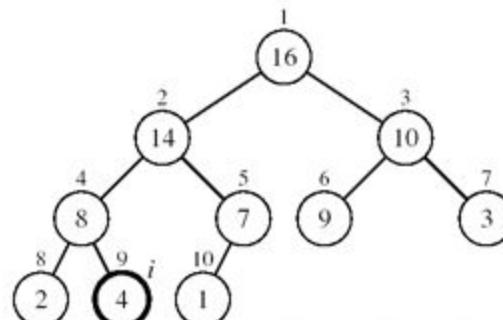
A[2] violates the heap property

$A[2] \leftrightarrow A[4]$



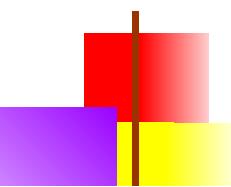
A[4] violates the heap property

$A[4] \leftrightarrow A[9]$



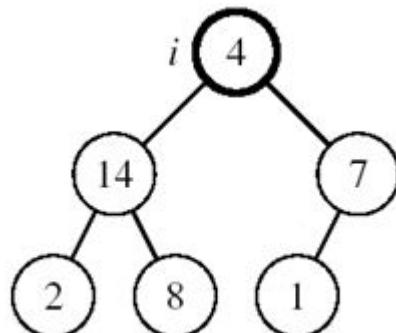
Heap property restored

A<sub>i</sub>



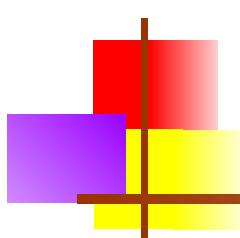
# Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of  $i$  are max-heaps
  - $A[i]$  may be smaller than its children



*Alg:* MAX-HEAPIFY( $A, i, n$ )

1.  $l \leftarrow \text{LEFT}(i)$
2.  $r \leftarrow \text{RIGHT}(i)$
3. **if**  $l \leq n$  and  $A[l] > A[i]$
4.   **then**  $\text{largest} \leftarrow l$
5.   **else**  $\text{largest} \leftarrow i$
6. **if**  $r \leq n$  and  $A[r] > A[\text{largest}]$
7.   **then**  $\text{largest} \leftarrow r$
8. **if**  $\text{largest} \neq i$
9.   **then** exchange  $A[i] \leftrightarrow A[\text{largest}]$
10.      MAX-HEAPIFY( $A, \text{largest}, n$ )



# MAX-HEAPIFY Running Time

- ❑ Intuitively:

It traces a path from the root to a leaf (longest path length:  $d$ )  
At each level, it makes exactly 2 comparisons  
Total number of comparisons is  $2d$   
Running time is  $O(d)$  or  $O(\lg n)$

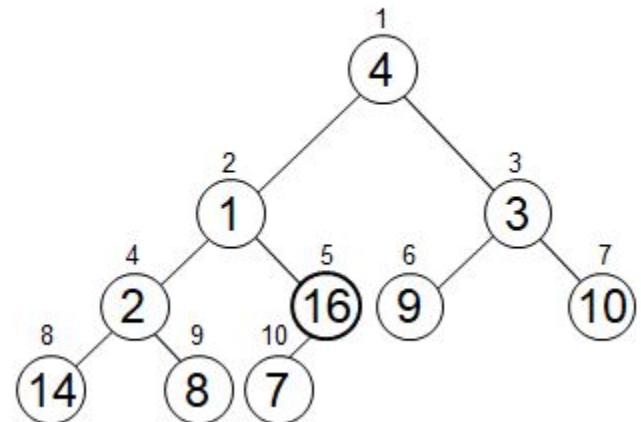
- ❑ Running time of MAX-HEAPIFY is  $O(\lg n)$
- ❑ Can be written in terms of the height of the heap, as being  $O(h)$ 
  - ❑ Since the height of the heap is  $\lg n$

# Building a Heap

- Convert an array  $A[1 \dots n]$  into a max-heap ( $n = \text{length}[A]$ )
- The elements in the subarray  $A[(n/2+1) \dots n]$  are leaves
- Apply MAX-HEAPIFY on elements between 1 and  $n/2$

*Alg:* BUILD-MAX-HEAP( $A$ )

- $n = \text{length}[A]$
- for**  $i \leftarrow \lfloor n/2 \rfloor$  **downto** 1
- do** MAX-HEAPIFY( $A, i, n$ )



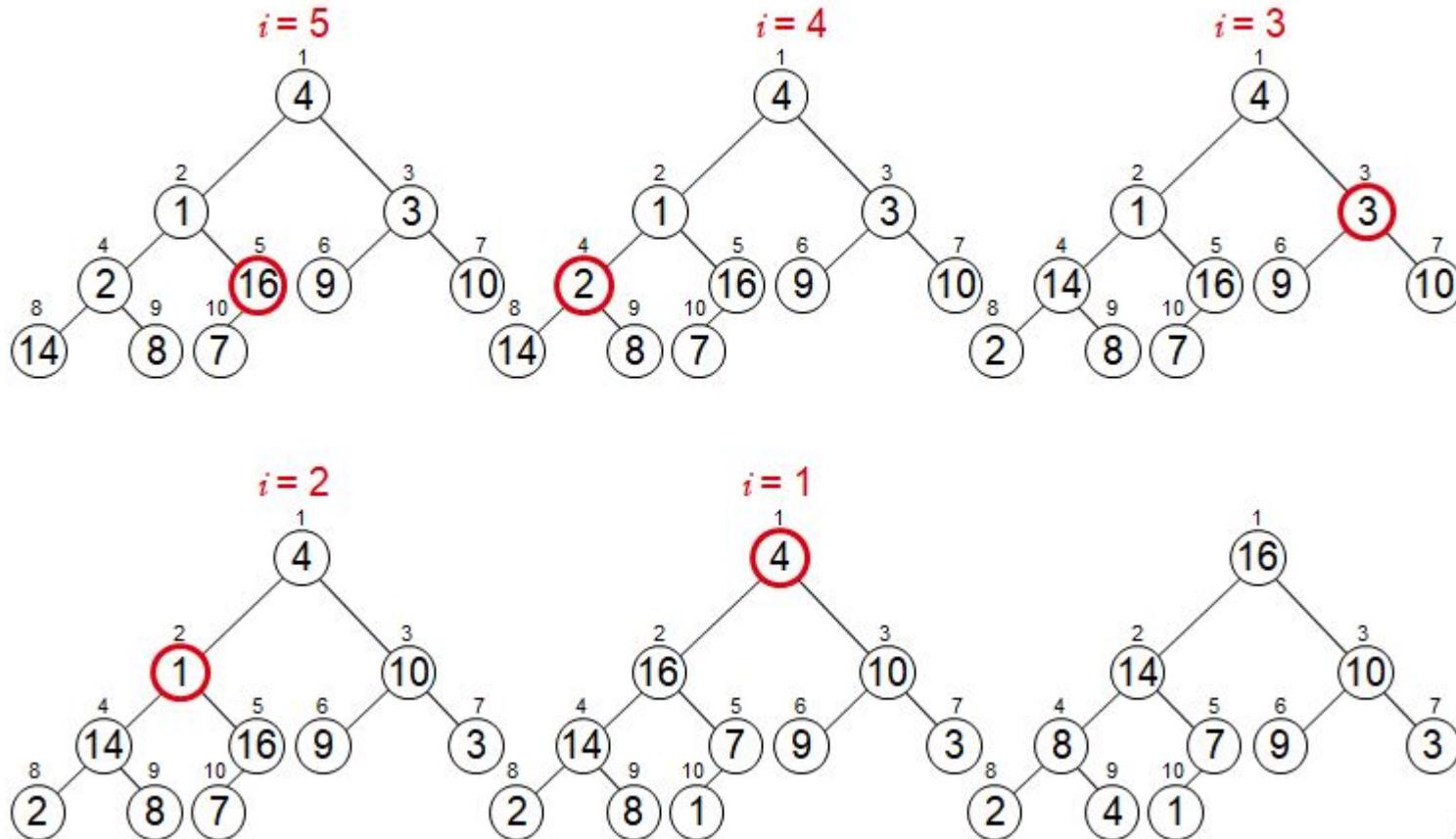
A: 

|   |   |   |   |    |   |    |    |   |   |
|---|---|---|---|----|---|----|----|---|---|
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|----|---|----|----|---|---|

# Example

A: 

|   |   |   |   |    |   |    |    |   |   |
|---|---|---|---|----|---|----|----|---|---|
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|----|---|----|----|---|---|



Ac

# Running Time of BUILD MAX HEAP

*Alg:* BUILD-MAX-HEAP( $A$ )

1.  $n = \text{length}[A]$
  2. **for**  $i \leftarrow \lfloor n/2 \rfloor$  **downto** 1
  3.     **do**  $\text{MAX-HEAPIFY}(A, i, n)$
- $O(\lg n)$  }  $O(n)$

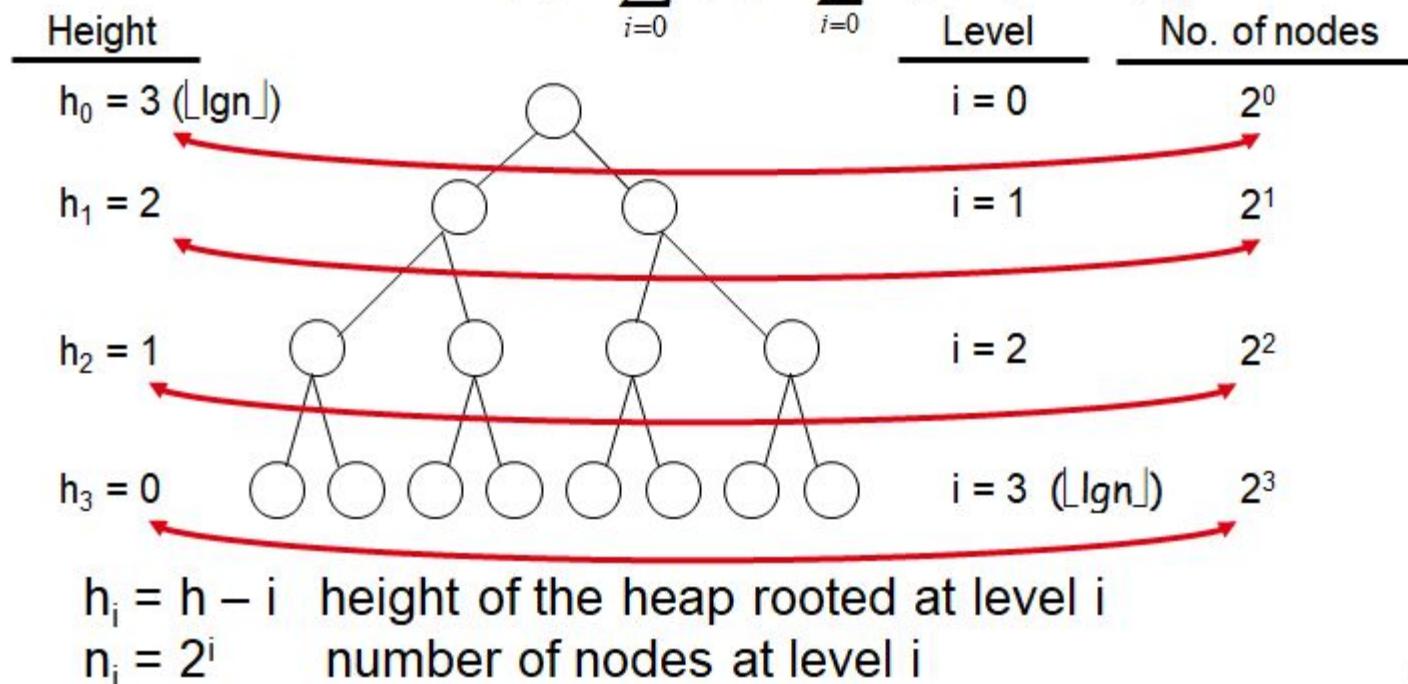
⇒ Running time:  $O(n \lg n)$

- This is not an asymptotically tight upper bound

# Running Time of BUILD MAX HEAP

HEAPIFY takes  $O(h)$  ⇒ the cost of HEAPIFY on a node  $i$  is proportional to the height of the node  $i$  in the tree

$$\Rightarrow T(n) = \sum_{i=0}^h n_i h_i = \sum_{i=0}^h 2^i (h-i) = O(n)$$



# Running Time of BUILD MAX HEAP

$$T(n) = \sum_{i=0}^h n_i h_i \quad \text{Cost of HEAPIFY at level } i * \text{number of nodes at that level}$$

$$= \sum_{i=0}^h 2^i (h-i) \quad \text{Replace the values of } n_i \text{ and } h_i \text{ computed before}$$

$$= \sum_{i=0}^h \frac{h-i}{2^{h-i}} 2^h \quad \text{Multiply by } 2^h \text{ both at the nominator and denominator and write } 2^i \text{ as } \frac{1}{2^{-i}}$$

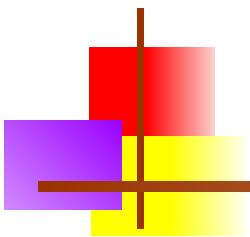
$$= 2^h \sum_{k=0}^h \frac{k}{2^k} \quad \text{Change variables: } k = h - i$$

$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k} \quad \text{The sum above is smaller than the sum of all elements to } \infty \text{ and } h = \lg n$$

$$= O(n) \quad \text{The sum above is smaller than 2}$$

Running time of BUILD-MAX-HEAP:  $T(n) = O(n)$

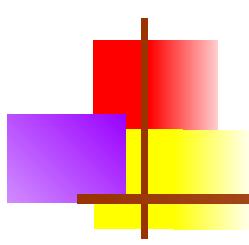
A



# Heap Sort

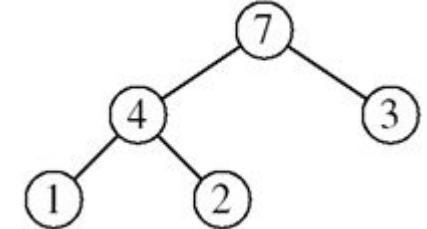
---



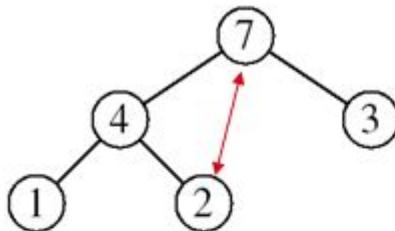


# Heapsort

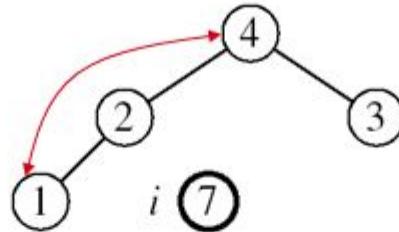
- Goal:
  - sort an array using heap representation
- Idea:
  - Build a max-heap from the array
  - Swap the root (the maximum element) with the last element in the array
  - “Discard” this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains



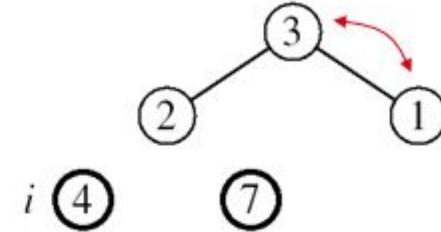
# Example: $A=[7, 4, 3, 1, 2]$



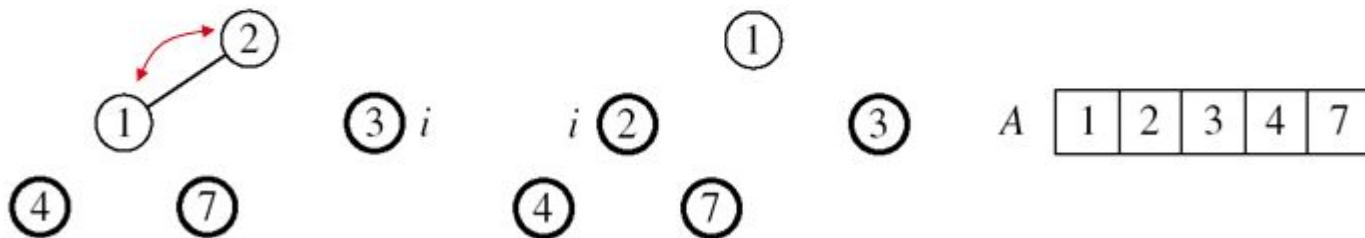
MAX-HEAPIFY( $A$ , 1, 4)



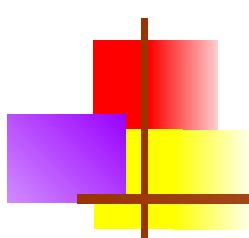
MAX-HEAPIFY( $A$ , 1, 3)



MAX-HEAPIFY( $A$ , 1, 2)



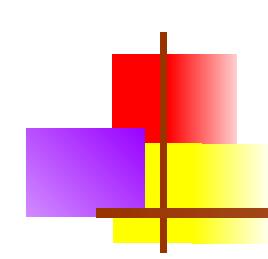
MAX-HEAPIFY( $A$ , 1, 1)



# Alg: HEAPSORT( $A$ )

1.  $\text{BUILD-MAX-HEAP}(A)$   $O(n)$
  2.  $\text{for } i \leftarrow \text{length}[A] \text{ downto } 2$
  3.     do exchange  $A[1] \leftrightarrow A[i]$
  4.      $\text{MAX-HEAPIFY}(A, 1, i - 1)$   $O(\lg n)$
- }  $n-1$  times

- Running time:  $O(n \lg n)$  --- Can be shown to be  $\Theta(n \lg n)$



# Priority Queues

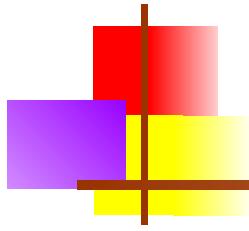
---

## Properties

- Each element is associated with a value (priority)
- The key with the highest (or lowest) priority is extracted first
- Major operations

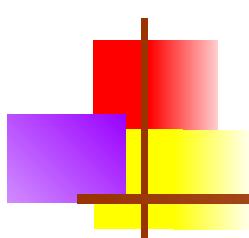
*Remove* an element from the queue

*Insert* an element in the queue



# Operations on Priority Queues

- ❑ Max-priority queues support the following operations:
  - ❑ INSERT( $S, x$ ): inserts element  $x$  into set  $S$
  - ❑ EXTRACT-MAX( $S$ ): removes and returns element of  $S$  with largest key
  - ❑ MAXIMUM( $S$ ): returns element of  $S$  with largest key
  - ❑ INCREASE-KEY( $S, x, k$ ): increases value of element  $x$ 's key to  $k$  (Assume  $k \geq x$ 's current key value)



# HEAP-MAXIMUM

Goal:

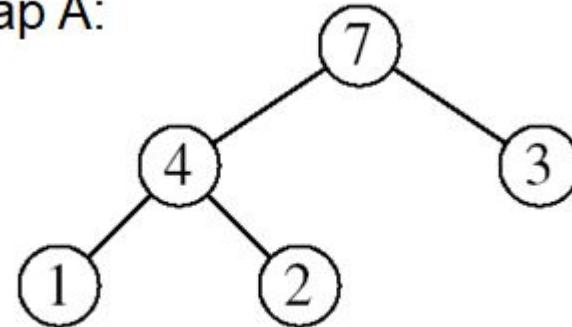
- Return the largest element of the heap

Running time:  $O(1)$

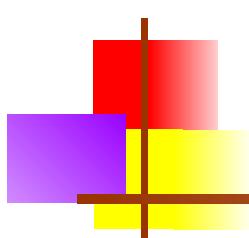
Alg: HEAP-MAXIMUM( $A$ )

1. **return  $A[1]$**

Heap A:



Heap-Maximum( $A$ ) returns 7



# HEAP-EXTRACT-MAX

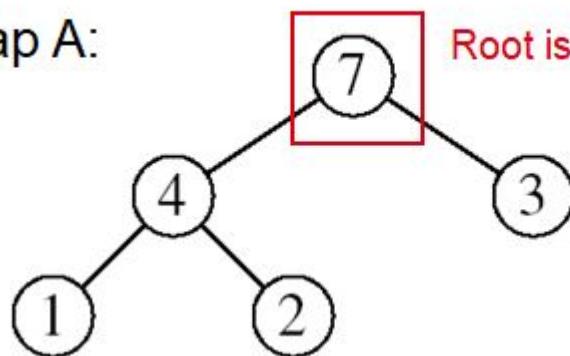
Goal:

- ❑ Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap)

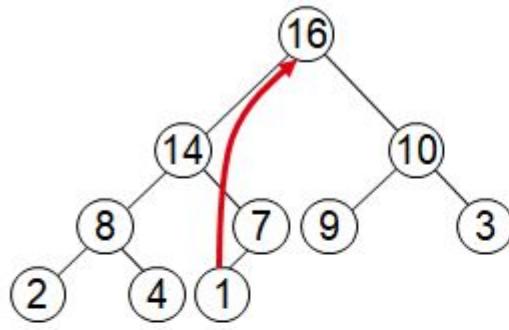
Idea:

- ❑ Exchange the root element with the last
- ❑ Decrease the size of the heap by 1 element
- ❑ Call MAX-HEAPIFY on the new root, on a heap of size  $n-1$

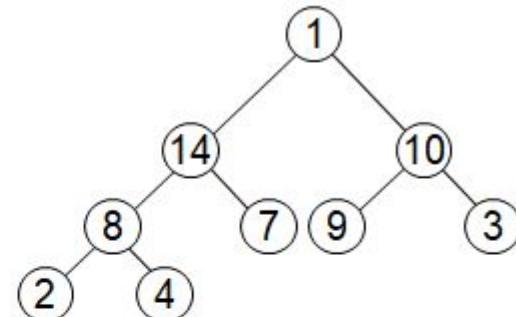
Heap A:  
Root is the largest element



# Example: HEAP-EXTRACT-MAX

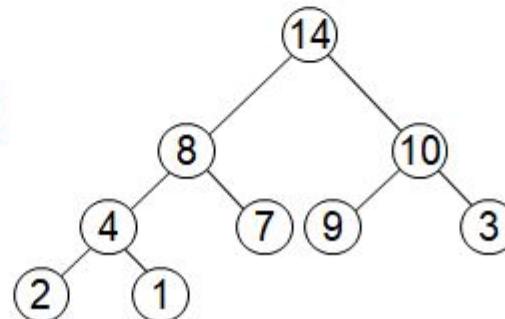


max = 16



Heap size decreased with 1

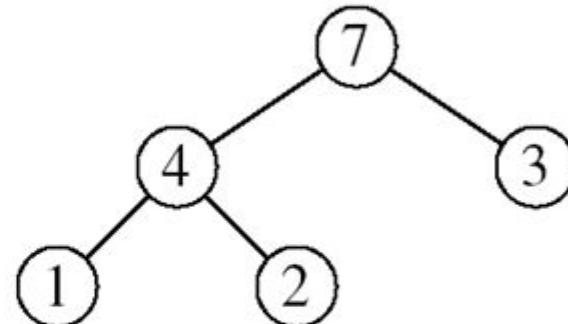
Call MAX-HEAPIFY( $A, 1, n-1$ )



# HEAP-EXTRACT-MAX

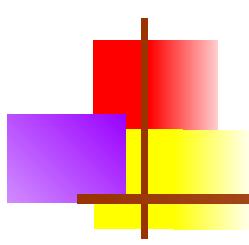
*Alg:* HEAP-EXTRACT-MAX( $A$ ,  $n$ )

1. **if**  $n < 1$
2.   **then error** “heap underflow”
3.  $\max \leftarrow A[1]$
4.  $A[1] \leftarrow A[n]$
5. MAX-HEAPIFY( $A$ , 1,  $n-1$ )                       $\triangleright$  remakes heap
6. **return**  $\max$



Running time:  $O(\lg n)$

A

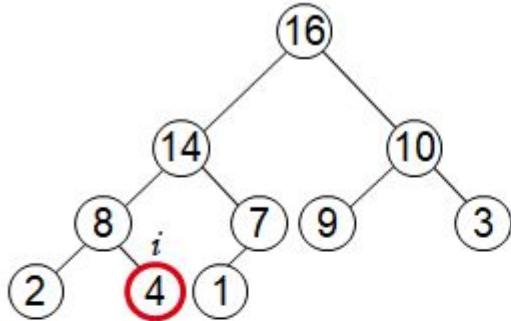


# HEAP-INCREASE-KEY

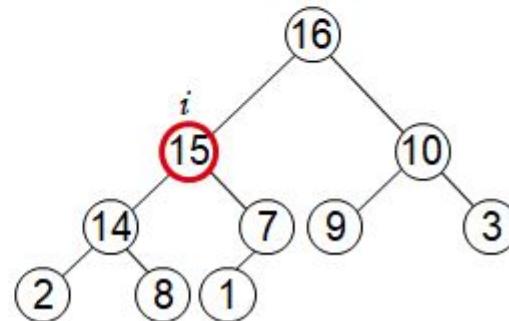
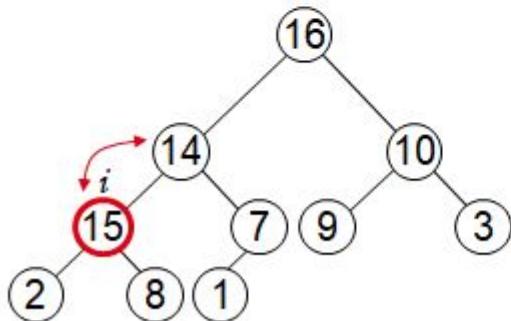
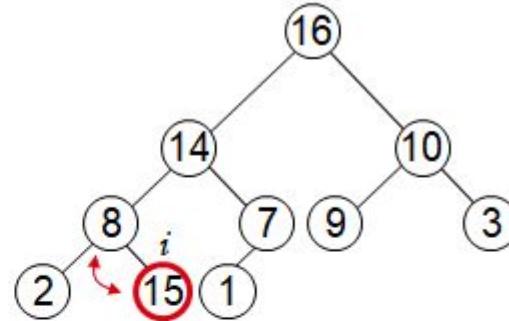
---

- Goal:
  - Increases the key of an element  $i$  in the heap
- Idea:
  - Increment the key of  $A[i]$  to its new value
  - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key

# Example: HEAP-INCREASE-KEY



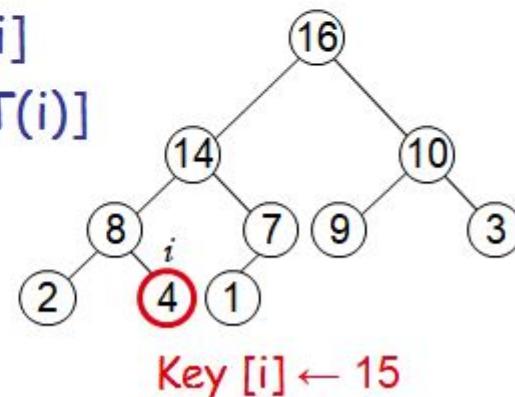
$\text{Key}[i] \leftarrow 15$



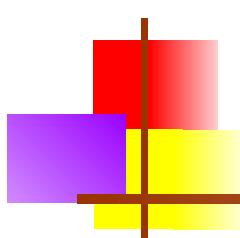
# HEAP-INCREASE-KEY

*Alg:* HEAP-INCREASE-KEY( $A$ ,  $i$ , key)

1. **if** key <  $A[i]$
  2.   **then error** “new key is smaller than current key”
  3.    $A[i] \leftarrow \text{key}$
  4.   **while**  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$
  5.     **do exchange**  $A[i] \leftrightarrow A[\text{PARENT}(i)]$
  6.      $i \leftarrow \text{PARENT}(i)$
- Running time:  $O(\lg n)$

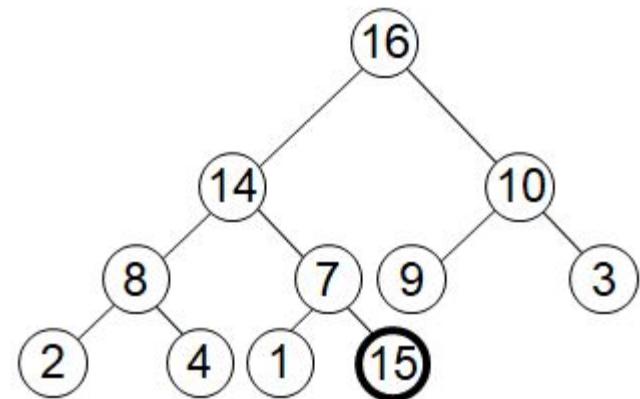
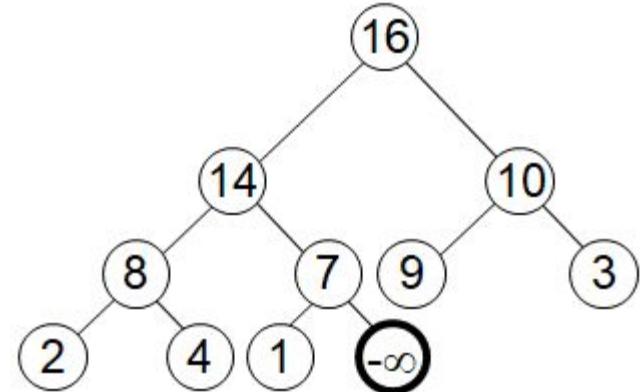


A<sub>i</sub>



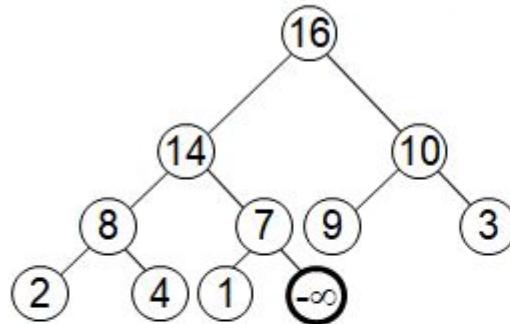
# MAX-HEAP-INSERT

- ❑ Goal:
  - ❑ Inserts a new element into a max-heap
- ❑ Idea:
  - ❑ Expand the max-heap with a new element whose key is  $-\infty$
  - ❑ Calls  
HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property

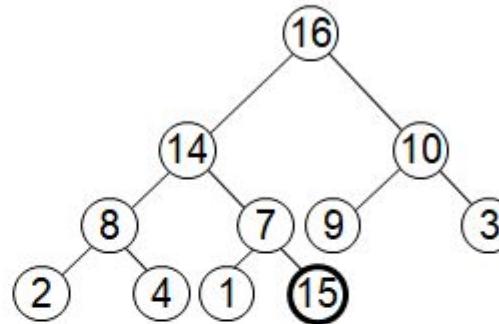


# Example: MAX-HEAP-INSERT

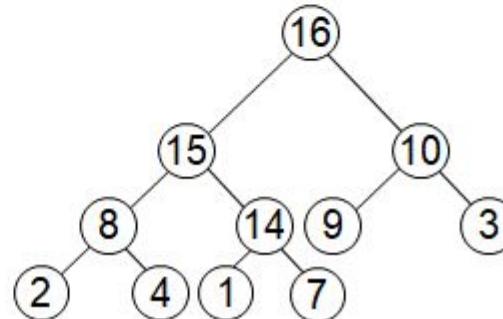
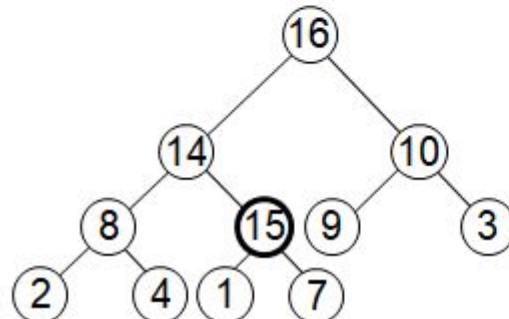
Insert value 15:  
- Start by inserting  $-\infty$



Increase the key to 15  
Call HEAP-INCREASE-KEY on  $A[11] = 15$



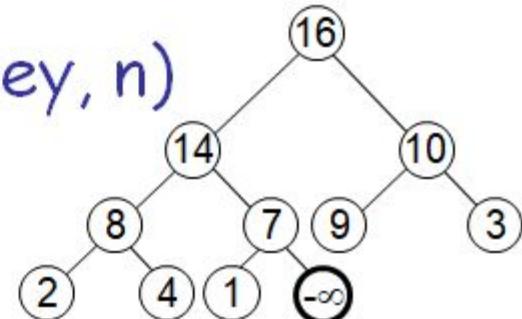
The restored heap containing  
the newly added element



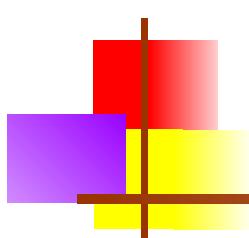
# MAX-HEAP-INSERT

*Alg:* MAX-HEAP-INSERT( $A$ , key,  $n$ )

1.  $\text{heap-size}[A] \leftarrow n + 1$
2.  $A[n + 1] \leftarrow -\infty$
3. HEAP-INCREASE-KEY( $A$ ,  $n + 1$ , key)



Running time:  $O(\lg n)$



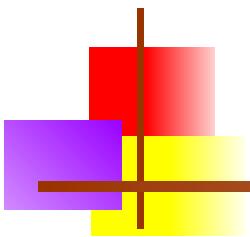
# Summary

- We can perform the following operations on heaps:

|                     |           |
|---------------------|-----------|
| – MAX-HEAPIFY       | $O(lgn)$  |
| – BUILD-MAX-HEAP    | $O(n)$    |
| – HEAP-SORT         | $O(nlgn)$ |
| – MAX-HEAP-INSERT   | $O(lgn)$  |
| – HEAP-EXTRACT-MAX  | $O(lgn)$  |
| – HEAP-INCREASE-KEY | $O(lgn)$  |
| – HEAP-MAXIMUM      | $O(1)$    |

*Average  
 $O(lgn)$*

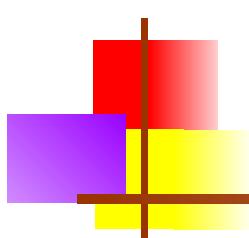
Ac



# Divide and Conquer

---

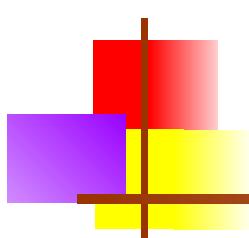




# Divide and Conquer

---

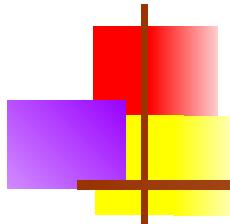
- **Divide** the problem into a number of subproblems
- **Conquer** the subproblems (solve them)
- **Combine** the subproblem solutions to get the solution to the original problem
- Note: often the “conquer” step is done recursively



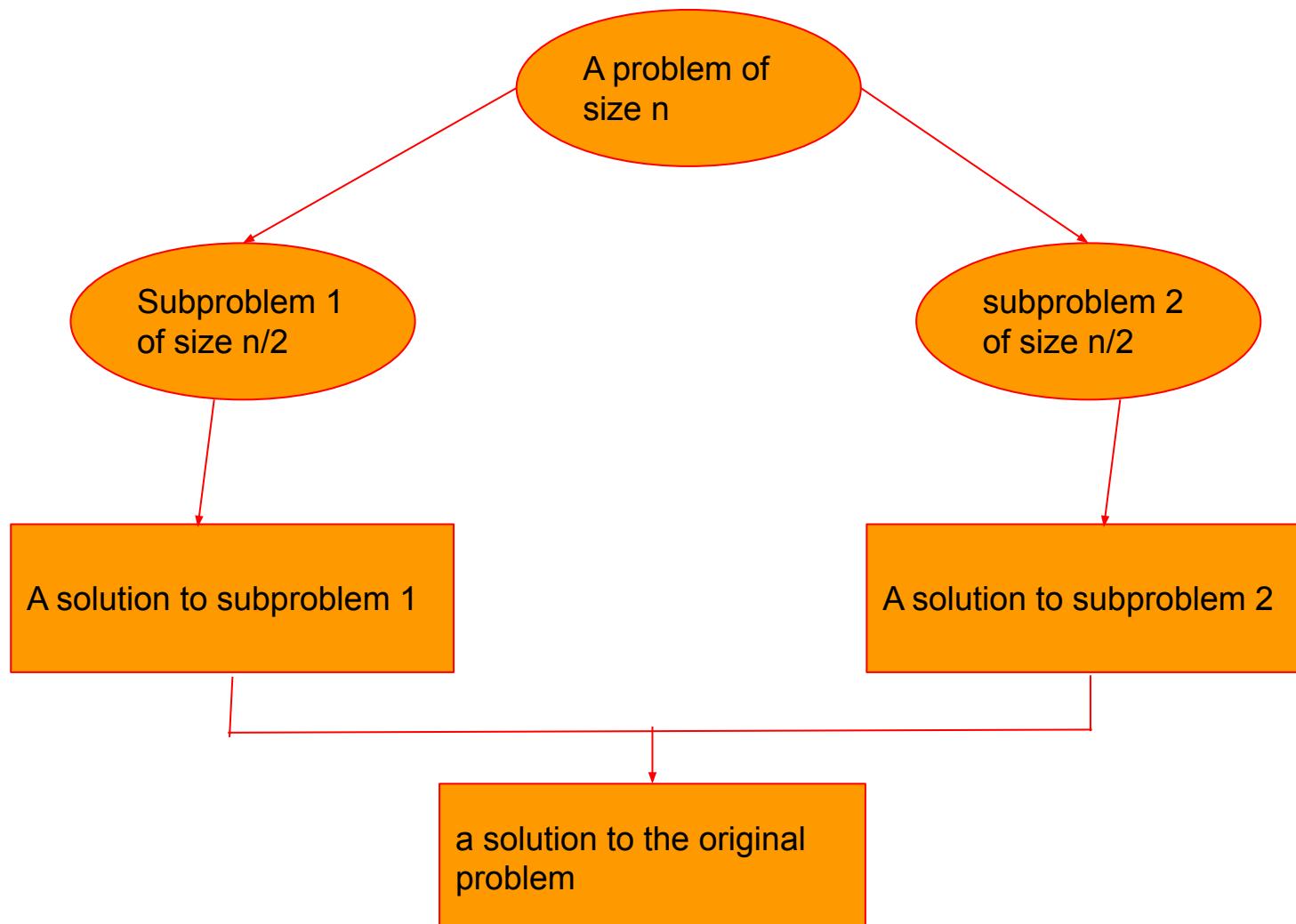
# Divide and Conquer

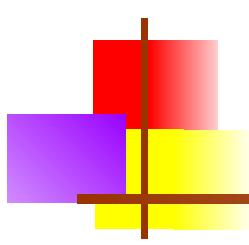
---

- ❑ A general methodology for using recursion to design efficiently algorithms
- ❑ It solves a problem by:
  - ❑ Dividing the data into parts
  - ❑ Finding sub solutions for each of the parts
  - ❑ Constructing the final answer from the sub solutions



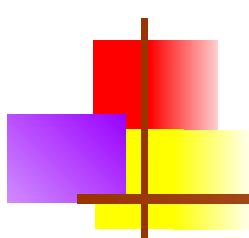
# Divide and Conquer





# Divide and Conquer Algo.

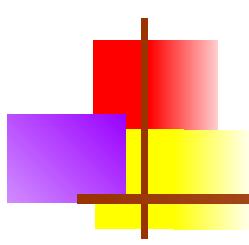
- ❑ **Divide** the problem into a number of subproblems
  - ❑ Subproblems must of same type
  - ❑ Subproblems do not need to overlap
- ❑ **Conquer** by solving sub-problems recursively. If the sub-problems are small enough, solve them in brute force fashion
- ❑ **Combine** the solutions of sub-problems into a solution of the original problem



# Divide and Conquer

---

- ❑ For divide-and-conquer algorithms the running time is mainly affected by 3 criteria:
  - ❑ The number of sub-instance into which a problem is split.
  - ❑ The ratio of initial problem size to sub-problem size.
  - ❑ The number of steps required to divide the initial instance and to combine sub solutions.



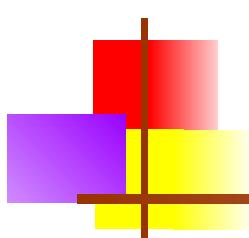
# Analyzing Divide and Conquer Algo.

- When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence equation which describes the overall running time on a problem of size  $n$  in terms of the running time on smaller inputs.
- For divide-and-conquer algorithms, we get recurrences that look like:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$


$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

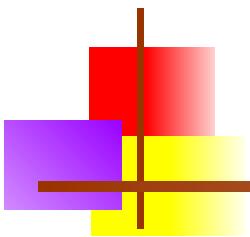
- ❑ where  $a$ =the number of subproblems we break the problem into
- ❑  $n/b$ =the size of the subproblems (in terms of  $n$ )
- ❑  $D(n)$  is the time to divide the problem of size  $n$  into the subproblems
- ❑  $C(n)$  is the time to combine the subproblem solutions to get the answer for the problem of size  $n$



# Example: Divide and Conquer

---

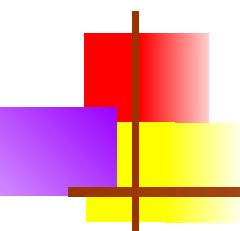
- Binary Search
- Heap Construction
- Tower of Hanoi
- Exponentiation
  - Fibonacci Sequence
- Quick sort
- Merge Sort
- Multiplying large integers
- Matrix Multiplication
- Closest Pairs



# Merge Sort

---



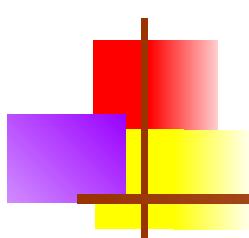


# Merge Sort

---

Recursive in structure

- ❑ **Divide** the problem into subproblems that are similar to the original but smaller in size
- ❑ **Conquer** the subproblems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
- ❑ **Combine** the solutions to create a solution to the original problem



# An Example: Merge Sort

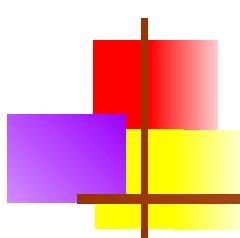
---

**Sorting Problem:** Sort a sequence of  $n$  element into non-decreasing order.

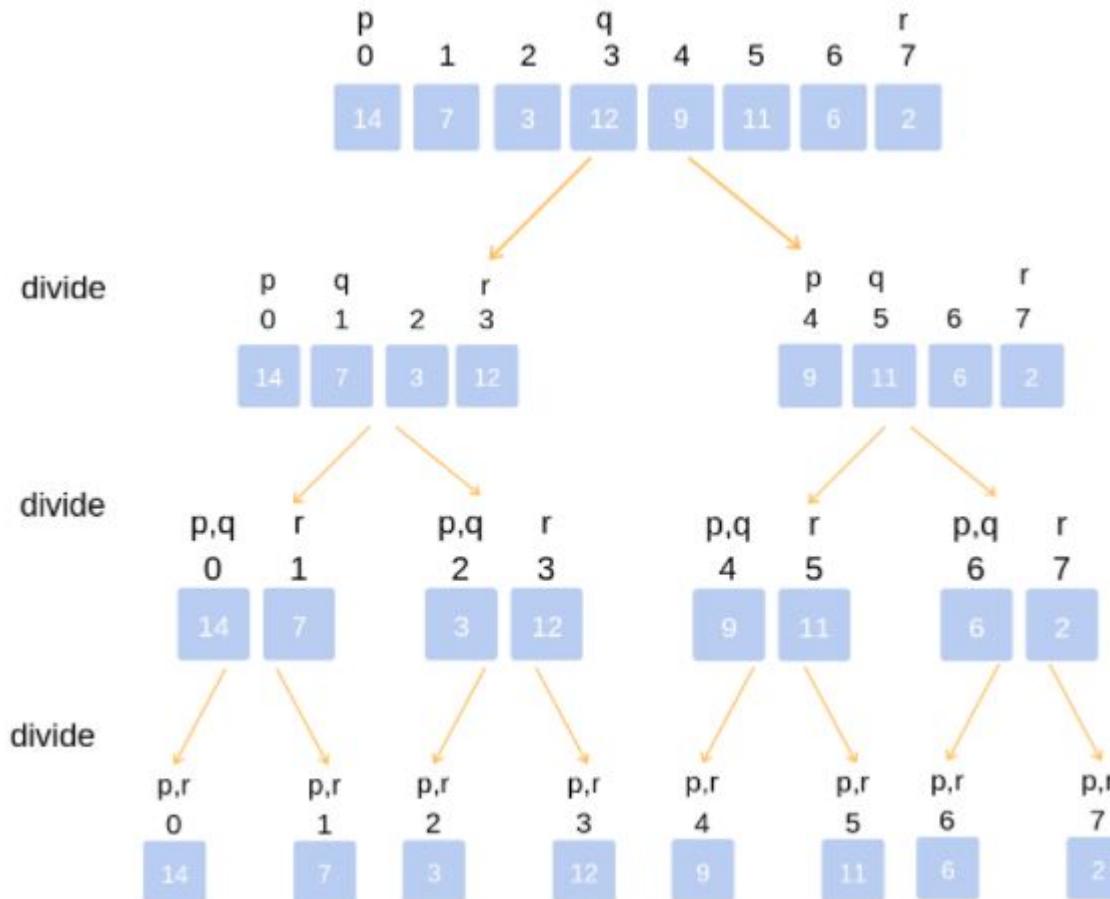
**Divide:** Divide the  $n$ -element sequence to be sorted into two subsequences of  $n/2$  elements each

**Conquer:** Sort the two subsequences recursively using merge sort

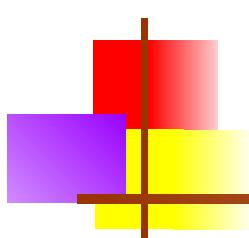
**Combine:** Merge the two sorted subsequences to produce the sorted.



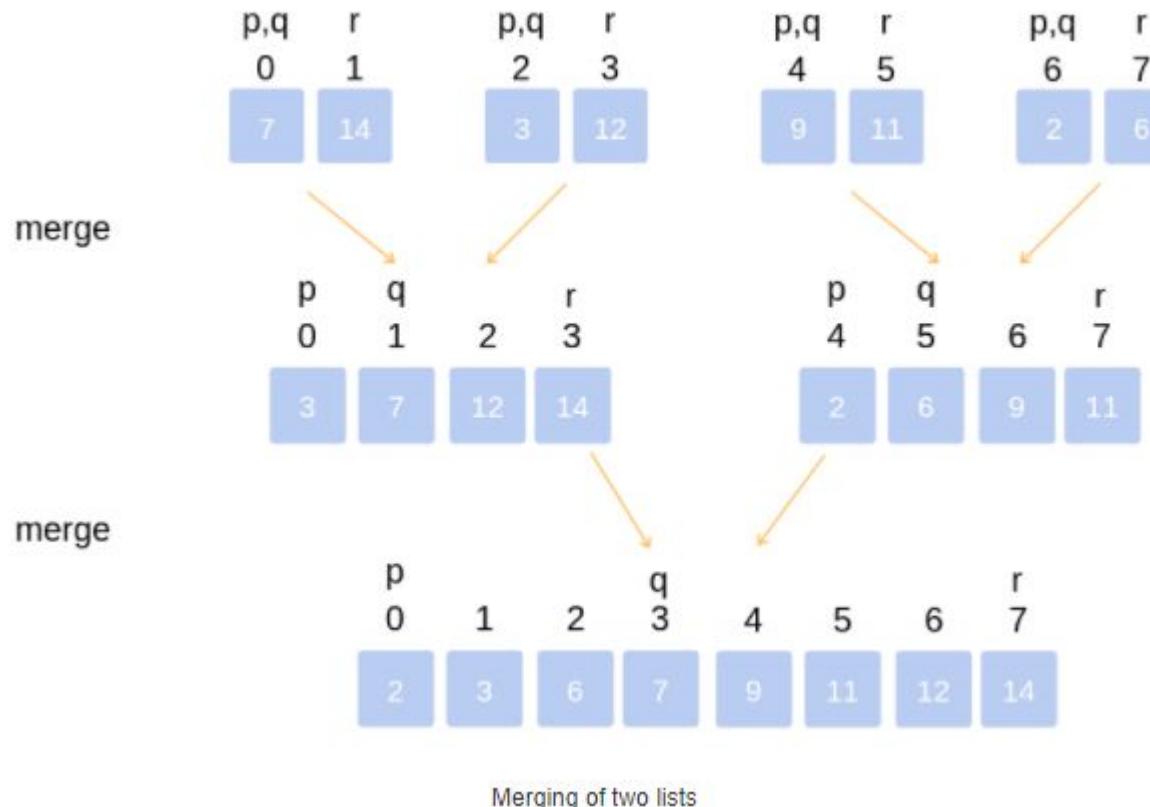
# An Example: Merge Sort

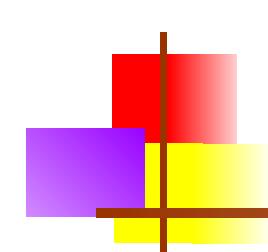


Top-down Implementation



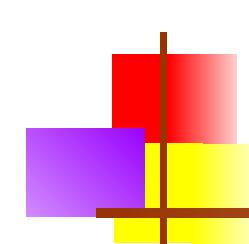
# An Example: Merge Sort





# Implementation Of Merge Sort

```
void mergeSort(int *Arr, int start, int end) {  
    if(start < end) {  
        int mid = (start + end) / 2;  
        mergeSort(Arr, start, mid);  
        mergeSort(Arr, mid+1, end);  
        merge(Arr, start, mid, end);  
    }  
}
```



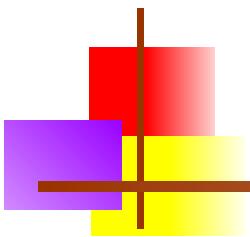
# Implementation Of Merge Sort

```
void merge(int *Arr, int start, int mid, int end) {  
    // create a temp array  
    int temp[end - start + 1];  
    // crawlers for both intervals and for temp  
    int i = start, j = mid+1, k = 0;  
    // traverse both arrays and in each iteration add smaller of both elements in temp  
    while(i <= mid && j <= end) {  
        if(Arr[i] <= Arr[j]) {  
            temp[k] = Arr[i];  
            k += 1; i += 1;  
        }  
        else {  
            temp[k] = Arr[j];  
            k += 1; j += 1;  
        }  
    }  
    // add elements left in the first interval  
    while(i <= mid) {  
        temp[k] = Arr[i];  
        k += 1; i += 1;}  
    // add elements left in the second interval  
    while(j <= end) {  
        temp[k] = Arr[j];  
        k += 1; j += 1; }  
    // copy temp to original interval  
    for(i = start; i <= end; i += 1) {  
        Arr[i] = temp[i - start]}  
}
```

# Merge Sort

how does merge sort stack up?

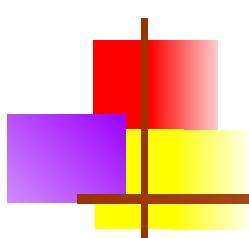
|                          |               |
|--------------------------|---------------|
| time complexity          | $O(n \log n)$ |
| space complexity         | out-of-place  |
| stability                | stable        |
| internal/external?       | external      |
| recursive/non-recursive? | recursive     |
| comparison sort?         | comparison    |



# QuickSort

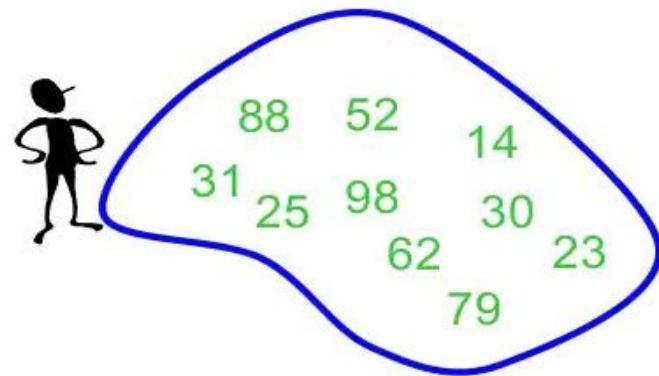
---





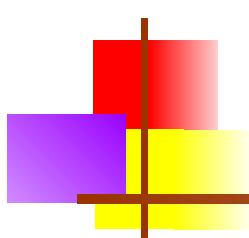
# Quick Sort

## Quick Sort



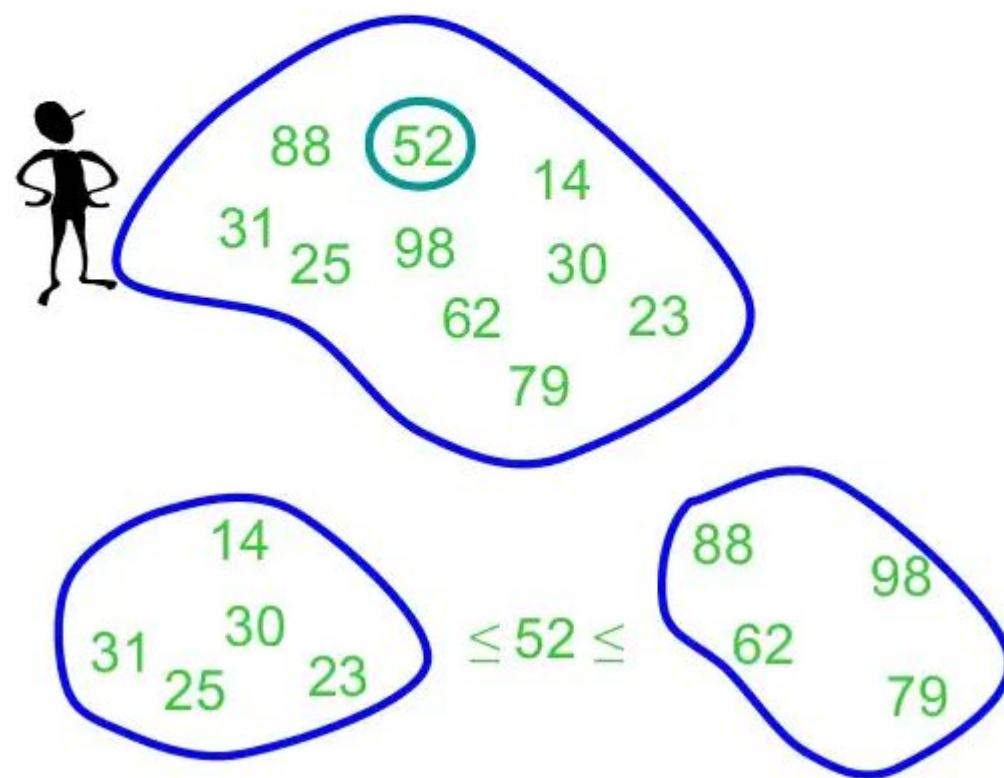
Divide and Conquer

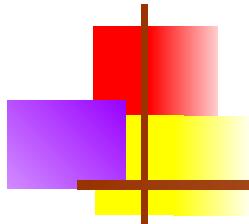




# Quick Sort

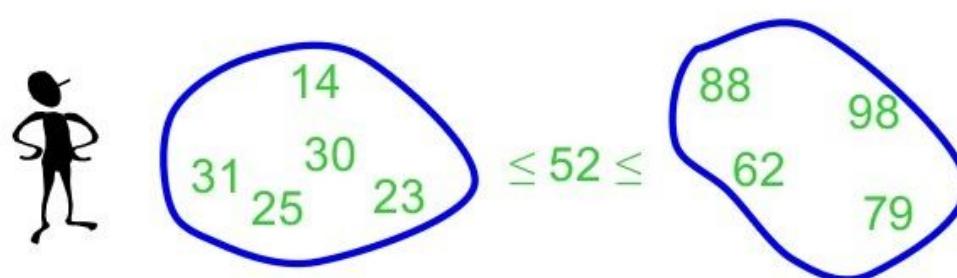
Partition set into two using randomly chosen pivot





# Quick Sort

## Quick Sort



sort the first half.

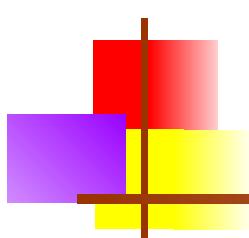


14, 23, 25, 30, 31

sort the second half.

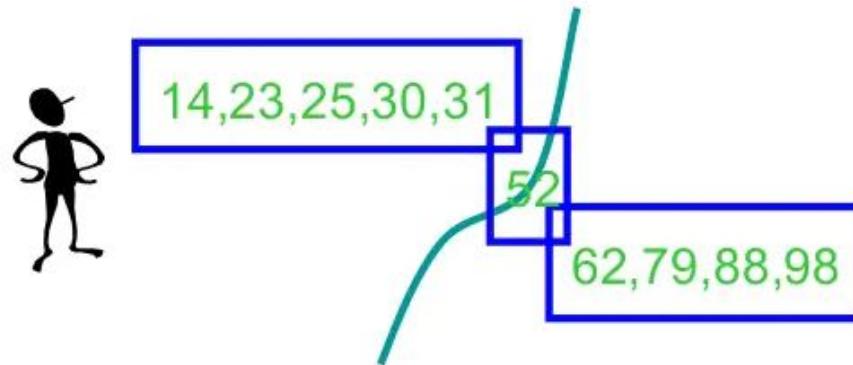


62, 79, 98, 88



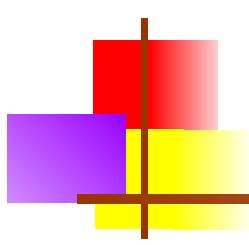
# Quick Sort

## Quick Sort



Glue pieces together.

14,23,25,30,31,52,62,79,88,98



# Quick Sort: Design

- ❑ Follows the **divide-and-conquer** paradigm
- ❑ **Divide:** Partition (separate) the array  $A[p..r]$  into two (possibly nonempty) subarrays  $A[p..q-1]$  and  $A[q+1..r]$ .
  - ❑ Each element in  $A[p..q-1] \leq A[q]$
  - ❑  $A[q] \leq$  each element in  $A[q+1,r]$
  - ❑ Index  $q$  is computed as part of the partitioning procedure.
- ❑ **Conquer:** Sort the two subarrays  $A[p..q-1]$  &  $A[q+1..r]$  by recursive calls to quicksort.
- ❑ **Combine:** Since the subarrays are sorted in place-no work is needed to combine them.

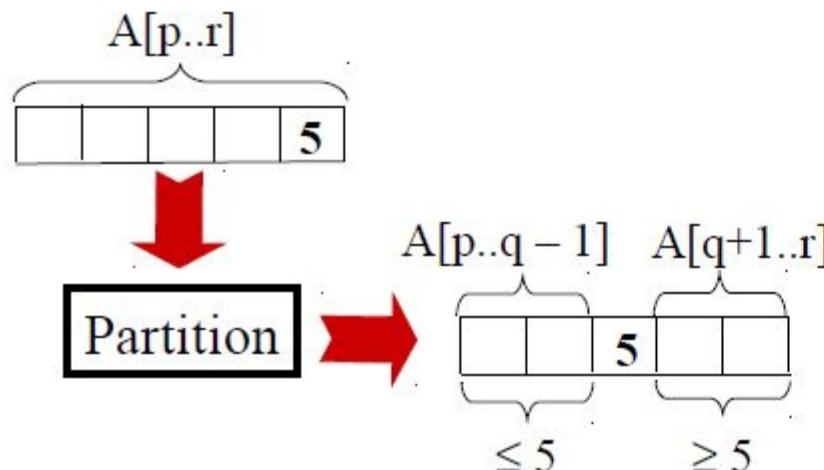
# Quick Sort: Pseudocode

```
Quicksort(A, p, r)
```

```
  if p < r then
```

```
    q := Partition(A, p, r);  
    Quicksort(A, p, q - 1);  
    Quicksort(A, q + 1, r)
```

```
  fi
```



```
Partition(A, p, r)
```

```
  x := A[r],
```

```
  i := p - 1;
```

```
  for j := p to r - 1 do
```

```
    if A[j] ≤ x then
```

```
      i := i + 1;
```

```
      A[i] ↔ A[j]
```

```
    fi
```

```
  od;
```

```
  A[i + 1] ↔ A[r];
```

```
  return i + 1
```

# Quick Sort: Example

initially:

$p$   
2 5 8 3 9 4 1 7 10  $r$   
 $i \ j$

note: pivot ( $x$ ) = 6

next iteration:

2 5 8 3 9 4 1 7 10 6  
i j

next iteration:

2 5 8 3 9 4 1 7 10 6  
i j

next iteration:

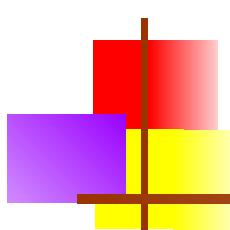
2 5 8 3 9 4 1 7 10 6  
i j

next iteration:

2 5 3 8 9 4 1 7 10 6  
i j

Partition( $A, p, r$ )

```
x, i := A[r], p - 1;  
for j := p to r - 1 do  
  if A[j] ≤ x then  
    i := i + 1;  
    A[i] ↔ A[j]  
  fi  
od;  
A[i + 1] ↔ A[r];  
return i + 1
```



# Quick Sort: Example

next iteration:

2 5 3 8 9 4 1 7 10 6  
i j

next iteration:

2 5 3 8 9 4 1 7 10 6  
i j

next iteration:

2 5 3 4 9 8 1 7 10 6  
i j

next iteration:

2 5 3 4 1 8 9 7 10 6  
i j

next iteration:

2 5 3 4 1 8 9 7 10 6  
i j

next iteration:

2 5 3 4 1 8 9 7 10 6  
i j

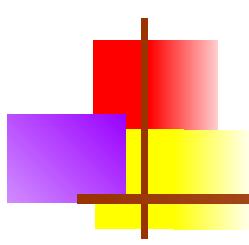
after final swap:

2 5 3 4 1 6 9 7 10 8  
i j

Partition(A, p, r)

```
x, i := A[r], p - 1;  
for j := p to r - 1 do  
  if A[j] ≤ x then  
    i := i + 1;  
    A[i] ↔ A[j]  
  fi  
od;  
A[i + 1] ↔ A[r];  
return i + 1
```

Activ  
Go to



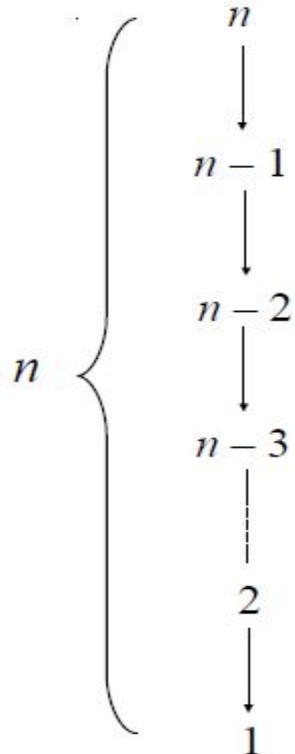
# Algorithm Performance

---

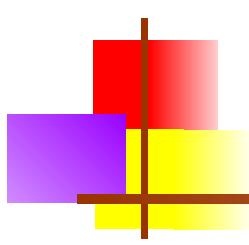
- ❑ Running time of quicksort depends on whether the partitioning is balanced or not.
- ❑ Worst-case partitioning(unbalanced partitions):
  - ❑ Occurs when every call to partition results in the most unbalanced partition.
  - ❑ Partition is most unbalanced when
    - ❑ Subproblem 1 is of size  $n-1$ , and subproblems 2 is of size 0 or vice versa.
    - ❑  $\text{pivot} \geq \text{every element in } A[p..r-1]$  or  $\text{pivot} < \text{every element in } A[p..r-1]$
  - ❑ Every call to partition is most unbalanced when
    - ❑ Array  $A[1..n]$  is sorted or reverse sorted.

# Worst-case Partition Analysis

Recursion tree for  
worst-case partition



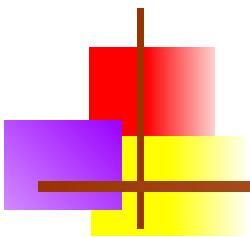
- Running time for worst-case partitions at each recursive level:
- $T(n) = T(n - 1) + T(0) + \text{PartitionTime}(n)$
- $= T(n - 1) + \Theta(n)$
- $= \sum_{k=1 \text{ to } n} \Theta(k)$
- $= \Theta(\sum_{k=1 \text{ to } n} k )$
- $= \Theta(n^2)$
-



# Best-case Partitioning

---

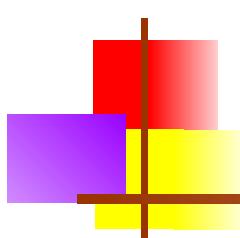
- Size of each subproblem  $\leq n/2$ .
  - one of the subproblems is of size  $n/2$
  - the other is of size  $n/2-1$
- Recurrence for running time
  - $$\begin{aligned} T(n) &\leq 2T(n/2) + \text{Partition Time}(n) \\ &= 2T(n/2) + \Theta(n) \end{aligned}$$
- $T(n)=\Theta(n \lg n)$



# Randomized QuickSort

---





# Randomized QuickSort

- ❑ An algorithm is **randomized** if its behavior is determined not only by the input but also by values produced by a random-number generator.
- ❑ Exchange  $A[r]$  with an element chosen at random from  $A[p \dots r]$  in Partition.
- ❑ This ensures that the pivot element **is equally likely to be any of input elements**
- ❑ We can sometimes add randomization to an algorithm in order to **obtain good average-case performance over all inputs.**

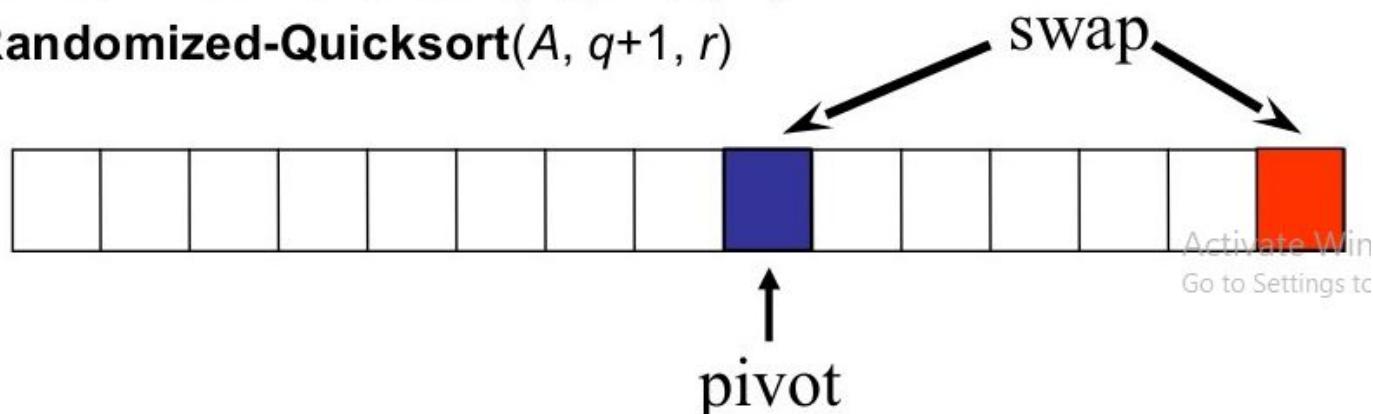
# Randomized QuickSort

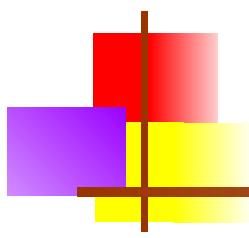
## Randomized-Partition( $A, p, r$ )

1.  $i \leftarrow \text{Random}(p, r)$
2. exchange  $A[r] \leftrightarrow A[i]$
3. **return Partition( $A, p, r$ )**

## Randomized-Quicksort( $A, p, r$ )

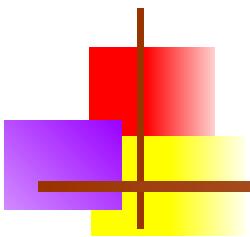
1. **if**  $p < r$
2. **then**  $q \leftarrow \text{Randomized-Partition}(A, p, r)$
3.     **Randomized-Quicksort( $A, p, q-1$ )**
4.     **Randomized-Quicksort( $A, q+1, r$ )**





# QuickSort

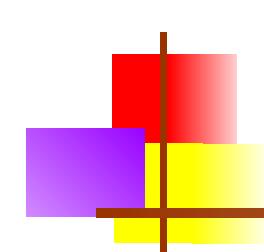
| Time Complexity  |               |
|------------------|---------------|
| Best             | $O(n \log n)$ |
| Worst            | $O(n^2)$      |
| Average          | $O(n \log n)$ |
| Space Complexity |               |
| Stability        |               |
|                  | No            |



# Asymptotic Analysis

---

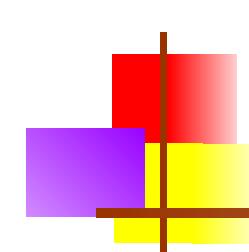




# Analysis of Algorithms

---

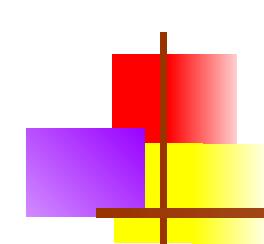
- An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.
- What is the goal of analysis of algorithms?
  - To compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)
- What do we mean by running time analysis?
  - Determine how running time increases as the **size** of the problem increases.



# Input Size

---

- Input size (number of elements in the input)
  - size of an array
  - polynomial degree
  - # of elements in a matrix
  - # of bits in the binary representation of the input
  - vertices and edges in a graph

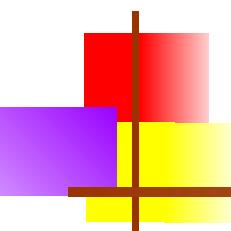


# Types of Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute **guarantee** that the algorithm would not run longer, no matter what the inputs are
- Best case
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest

$$\textit{Lower Bound} \leq \textit{Running Time} \leq \textit{Upper Bound}$$

- Average case
  - Provides a **prediction** about the running time
  - Assumes that the input is random



# How do we compare algorithms?

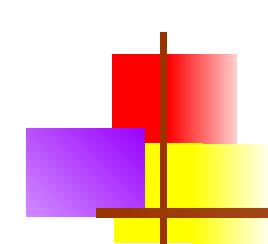
- We need to define a number of objective measures.

(1) Compare execution times?

**Not good:** times are specific to a particular computer !!

(2) Count the number of statements executed?

**Not good:** number of statements vary with the programming language as well as the style of the individual programmer.



# Ideal Solution

---

- Express running time as a function of the input size  $n$  (i.e.,  $f(n)$ ).
- Compare different functions corresponding to running times.
- Such an analysis is independent of machine time, programming style, etc.

# Example

- Associate a "cost" with each statement.
- Find the "total cost" by finding the total number of times each statement is executed.

*Algorithm 1*

|                      | <u>Cost</u>          |
|----------------------|----------------------|
| <u>arr[0] = 0;</u>   | <u>c<sub>1</sub></u> |
| <u>arr[1] = 0;</u>   | <u>c<sub>1</sub></u> |
| <u>arr[2] = 0;</u>   | <u>c<sub>1</sub></u> |
| ...                  | ...                  |
| <u>arr[N-1] = 0;</u> | <u>c<sub>1</sub></u> |

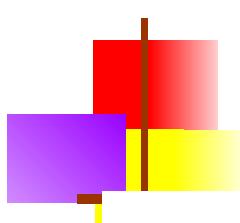
$$\text{-----}$$
$$c_1 + c_1 + \dots + c_1 = c_1 \times N$$

*Algorithm 2*

| for( <u>i=0; i&lt;N; i++</u> ) |
|--------------------------------|
| <u>arr[i] = 0;</u>             |

| <u>Cost</u>          |
|----------------------|
| <u>c<sub>2</sub></u> |
| <u>c<sub>1</sub></u> |

$$\text{-----}$$
$$(N+1) \times c_2 + N \times c_1 =$$
$$(c_2 + c_1) \times N + c_2$$



## Another Example

- **Algorithm 3**

sum = 0;

$c_1$

for(i=0; i<N; i++)

$c_2$

    for(j=0; j<N; j++)

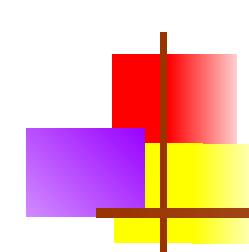
$c_2$

        sum += arr[i][j];

$c_3$

---

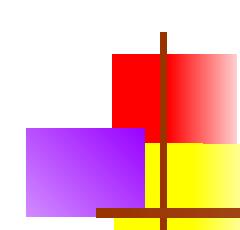
$$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2$$



# Asymptotic Analysis

---

- To compare two algorithms with running times  $f(n)$  and  $g(n)$ , we need a **rough measure** that characterizes **how fast each function grows**.
- Hint: use *rate of growth*
- Compare functions in the limit, that is, **asymptotically!**  
(i.e., for large values of  $n$ )



# Rate of Growth

- Consider the example of buying *elephants* and *goldfish*:

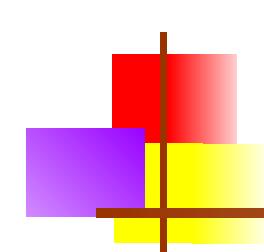
**Cost**: `cost_of_elephants + cost_of_goldfish`

**Cost**  $\sim$  `cost_of_elephants` (approximation)

- The low order terms in a function are relatively insignificant for **large  $n$**

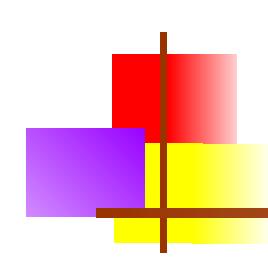
$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

i.e., we say that  $n^4 + 100n^2 + 10n + 50$  and  $n^4$  have the same **rate of growth**



# Asymptotic Notation

- • O notation: asymptotic “less than”:
  - $f(n)=O(g(n))$  implies:  $f(n)$  “ $\leq$ ”  $g(n)$
- $\Omega$  notation: asymptotic “greater than”:
  - $f(n)=\Omega(g(n))$  implies:  $f(n)$  “ $\geq$ ”  $g(n)$
- $\Theta$  notation: asymptotic “equality”:
  - $f(n)=\Theta(g(n))$  implies:  $f(n)$  “ $=$ ”  $g(n)$

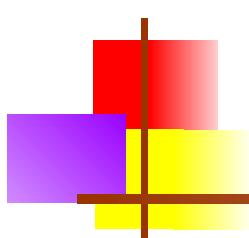


# Big-O Notation

---

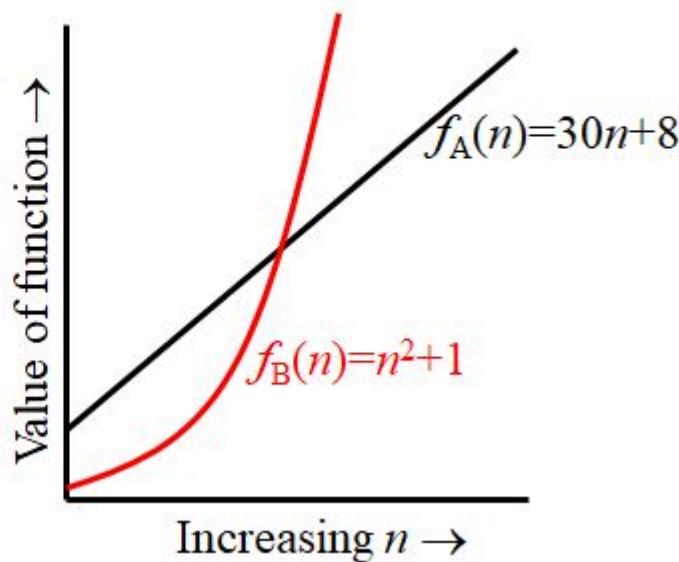
□

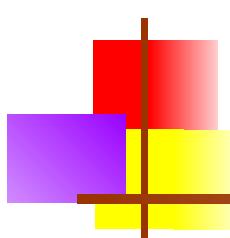
- We say  $f_A(n)=30n+8$  is *order n*, or  $O(n)$ .  
It is, at most, roughly *proportional* to  $n$ .
- $f_B(n)=n^2+1$  is *order  $n^2$* , or  $O(n^2)$ . It is, at most, roughly proportional to  $n^2$ .
- In general, any  $O(n^2)$  function is faster-growing than any  $O(n)$  function.



# Visualizing Orders of Growth

- On a graph, as you go to the right, a faster growing function eventually becomes larger...

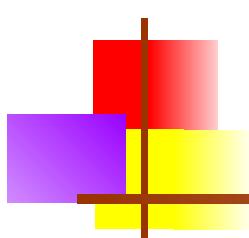




# More Examples ...

---

- $n^4 + 100n^2 + 10n + 50$  is  $O(n^4)$
- $10n^3 + 2n^2$  is  $O(n^3)$
- $n^3 - n^2$  is  $O(n^3)$
- constants
  - 10 is  $O(1)$
  - 1273 is  $O(1)$



# Back to Our Example

## Algorithm 1

|               | Cost           |
|---------------|----------------|
| arr[0] = 0;   | c <sub>1</sub> |
| arr[1] = 0;   | c <sub>1</sub> |
| arr[2] = 0;   | c <sub>1</sub> |
| ...           |                |
| arr[N-1] = 0; | c <sub>1</sub> |

$$\text{-----}$$
$$c_1 + c_1 + \dots + c_1 = c_1 \times N$$

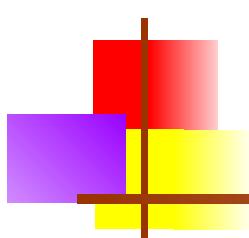
## Algorithm 2

```
for(i=0; i<N; i++)  
    arr[i] = 0;
```

| Cost           |
|----------------|
| c <sub>2</sub> |
| c <sub>1</sub> |

$$\text{-----}$$
$$(N+1) \times c_2 + N \times c_1 =$$
$$(c_2 + c_1) \times N + c_2$$

- Both algorithms are of the same order: O(N)



# Example (cont'd)

## *Algorithm 3*

```
sum = 0;  
for(i=0; i<N; i++)  
    for(j=0; j<N; j++)  
        sum += arr[i][j];
```

## *Cost*

$c_1$

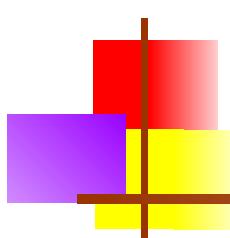
$c_2$

$c_2$

$c_3$

-----

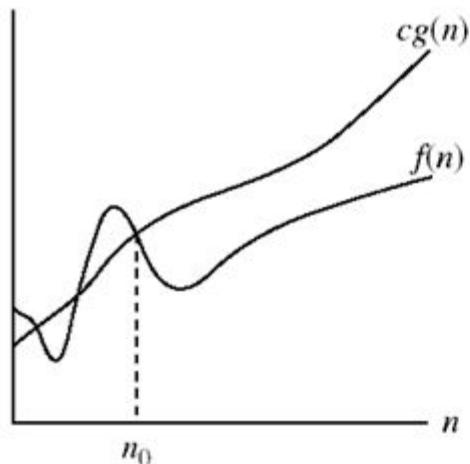
$$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2 = O(N^2)$$



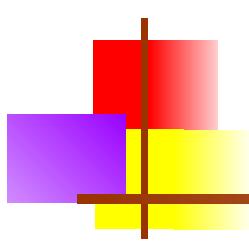
# Asymptotic notations

- *O-notation*

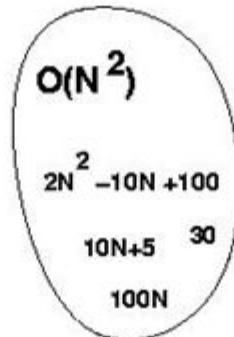
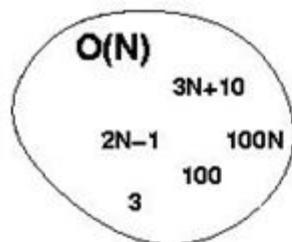
$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$   
 $0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$ .



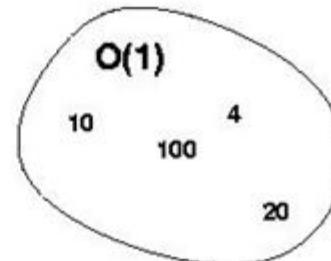
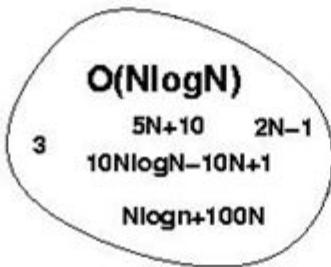
$g(n)$  is an *asymptotic upper bound* for  $f(n)$ .

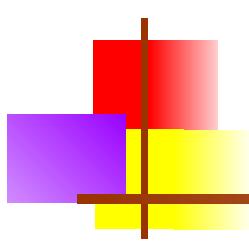


# Big-O Visualization



$O(g(n))$  is the set of functions with smaller or same order of growth as  $g(n)$



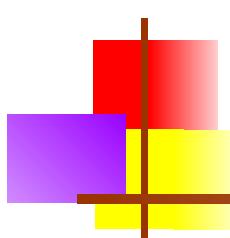


# Examples

- $2n^2 = O(n^3)$ :  $2n^2 \leq cn^3 \Rightarrow 2 \leq cn \Rightarrow c = 1$  and  $n_0 = 2$
- $n^2 = O(n^2)$ :  $n^2 \leq cn^2 \Rightarrow c \geq 1 \Rightarrow c = 1$  and  $n_0 = 1$
- $1000n^2 + 1000n = O(n^2)$ :

$$1000n^2 + 1000n \leq 1000n^2 + n^2 = 1001n^2 \Rightarrow c = 1001 \text{ and } n_0 = 1000$$

- $n = O(n^2)$ :  $n \leq cn^2 \Rightarrow cn \geq 1 \Rightarrow c = 1$  and  $n_0 = 1$



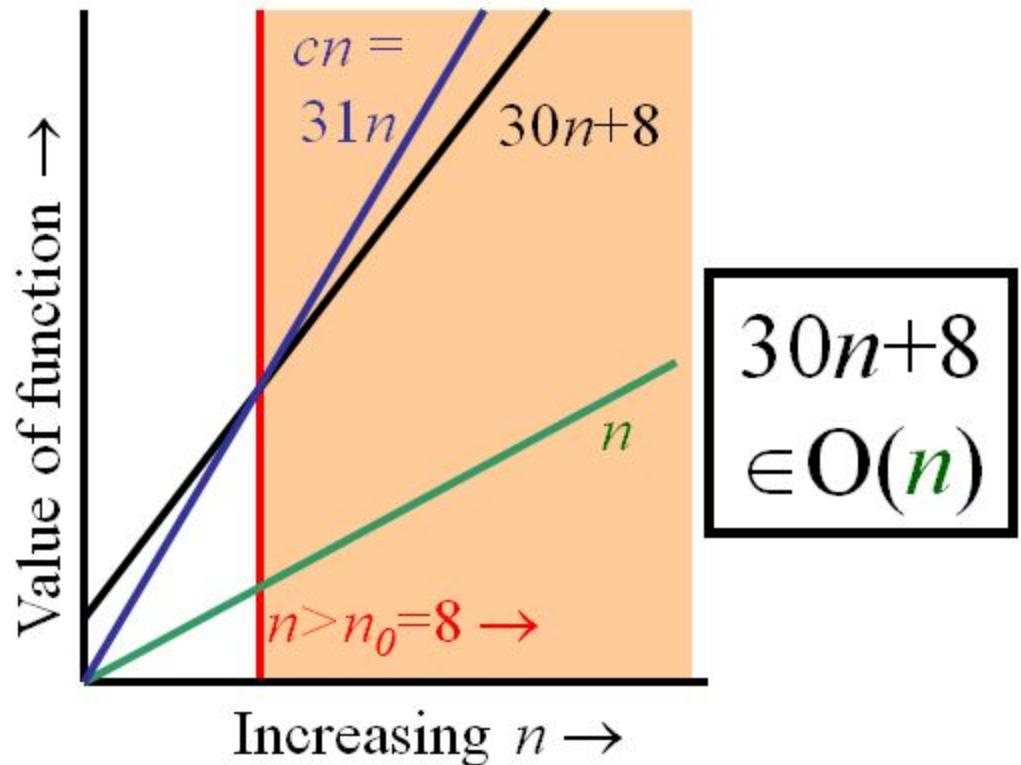
# More Examples

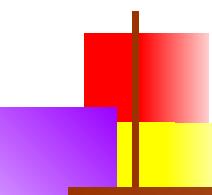
---

- Show that  $30n+8$  is  $O(n)$ .
  - Show  $\exists c, n_0: 30n+8 \leq cn, \forall n > n_0$ .
    - Let  $c=31, n_0=8$ . Assume  $n > n_0 = 8$ . Then  
 $cn = 31n = 30n + n > 30n+8$ , so  $30n+8 < cn$ .

# Big-O example, graphically

- Note  $30n+8$  isn't less than  $n$  anywhere ( $n>0$ ).
- It isn't even less than  $31n$  everywhere.
- But it *is* less than  $31n$  everywhere to the right of  $n=8$ .





# No Uniqueness

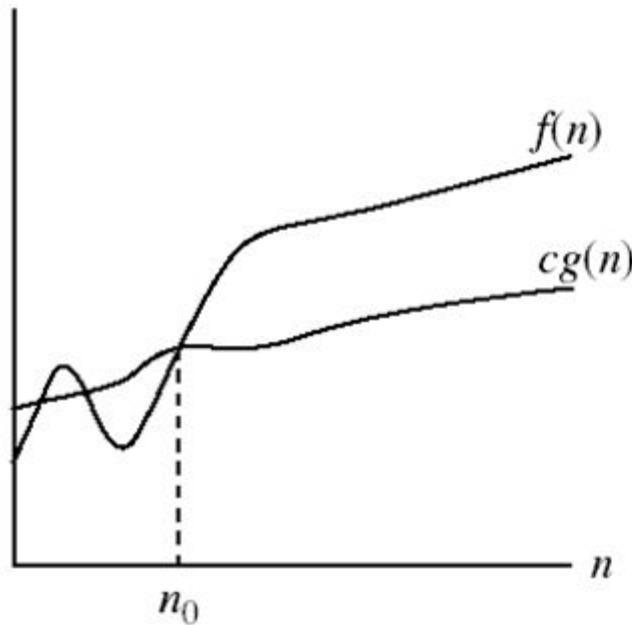
- There is no unique set of values for  $n_0$  and  $c$  in proving the asymptotic bounds
- Prove that  $100n + 5 = O(n^2)$ 
  - $100n + 5 \leq 100n + n = 101n \leq 101n^2$   
for all  $n \geq 5$
  - $n_0 = 5$  and  $c = 101$  is a solution
  - $100n + 5 \leq 100n + 5n = 105n \leq 105n^2$   
for all  $n \geq 1$
  - $n_0 = 1$  and  $c = 105$  is also a solution

Must find **SOME** constants  $c$  and  $n_0$  that satisfy the asymptotic notation relation

# Asymptotic notations (cont.)

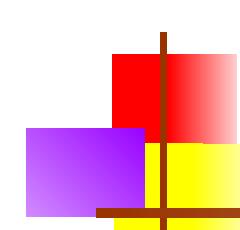
- $\Omega$  - notation

$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$   
 $0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$ .



$\Omega(g(n))$  is the set of functions  
with larger or same order of  
growth as  $g(n)$

$g(n)$  is an *asymptotic lower bound* for  $f(n)$ .



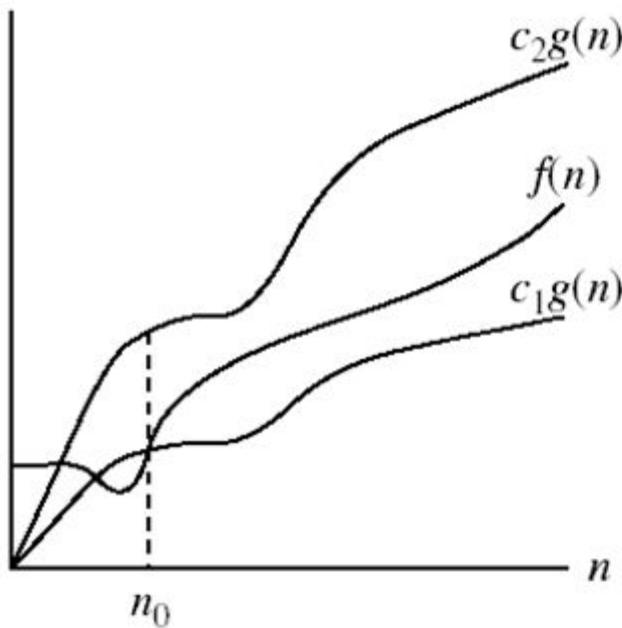
# Examples

- $5n^2 = \Omega(n)$   
 $\exists c, n_0$  such that:  $0 \leq cn \leq 5n^2 \Rightarrow cn \leq 5n^2 \Rightarrow c = 1$  and  $n_0 = 1$
- $100n + 5 \neq \Omega(n^2)$   
 $\exists c, n_0$  such that:  $0 \leq cn^2 \leq 100n + 5$   
 $100n + 5 \leq 100n + 5n \ (\forall n \geq 1) = 105n$   
 $cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$   
Since  $n$  is positive  $\Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$   
 $\Rightarrow$  contradiction:  $n$  cannot be smaller than a constant
- $n = \Omega(2n), n^3 = \Omega(n^2), n = \Omega(\log n)$

# Asymptotic notations (cont.)

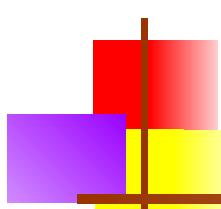
- $\Theta$ -notation

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}$   
 $0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$ .



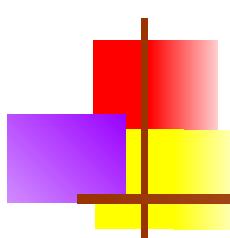
$\Theta(g(n))$  is the set of functions  
with the same order of growth  
as  $g(n)$

$g(n)$  is an *asymptotically tight bound* for  $f(n)$ .



# Examples

- $n^2/2 - n/2 = \Theta(n^2)$ 
  - $\frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad \forall n \geq 0 \Rightarrow c_2 = \frac{1}{2}$
  - $\frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n * \frac{1}{2}n \quad (\forall n \geq 2) = \frac{1}{4}n^2$  $\Rightarrow c_1 = \frac{1}{4}$
- $n \neq \Theta(n^2)$ :  $c_1 n^2 \leq n \leq c_2 n^2$   
 $\Rightarrow$  only holds for:  $n \leq 1/c_1$



# Examples

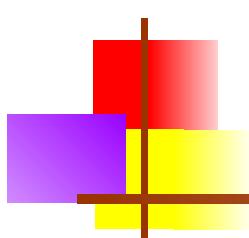
---

-  $6n^3 \neq \Theta(n^2)$ :  $c_1 n^2 \leq 6n^3 \leq c_2 n^2$

$\Rightarrow$  only holds for:  $n \leq c_2 / 6$

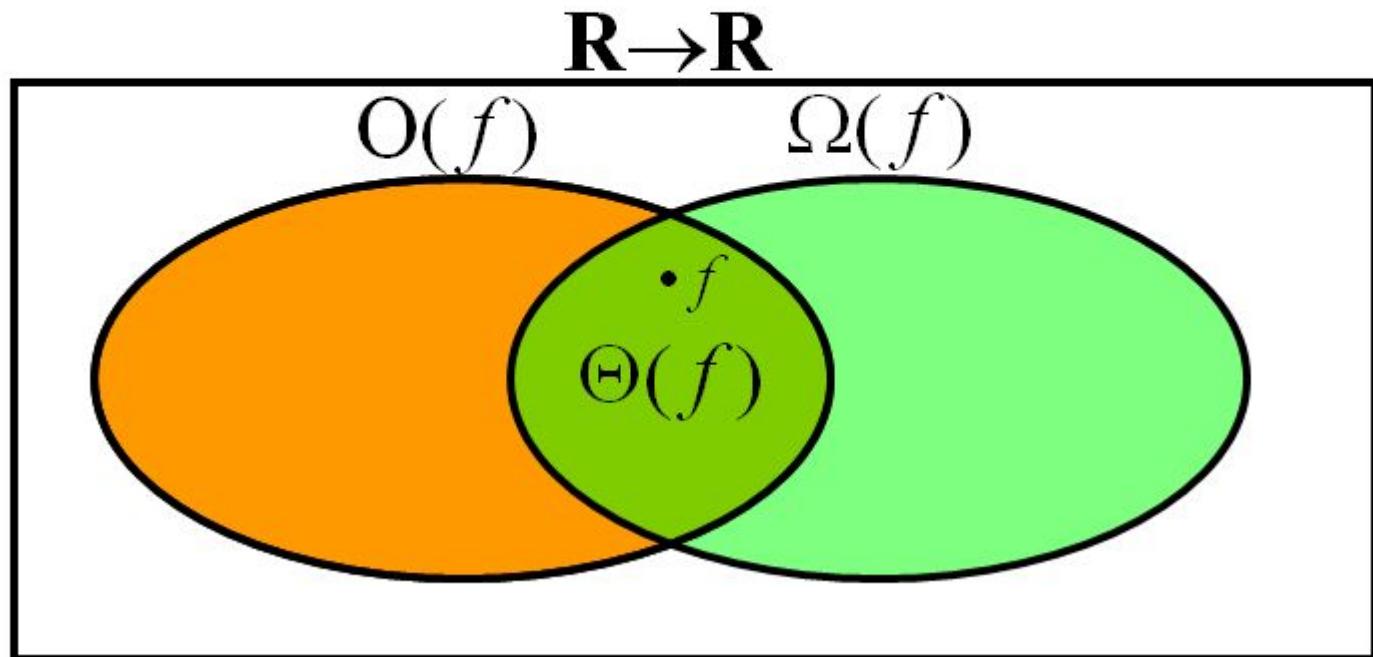
-  $n \neq \Theta(\log n)$ :  $c_1 \log n \leq n \leq c_2 \log n$

$\Rightarrow c_2 \geq n/\log n, \forall n \geq n_0$  - impossible

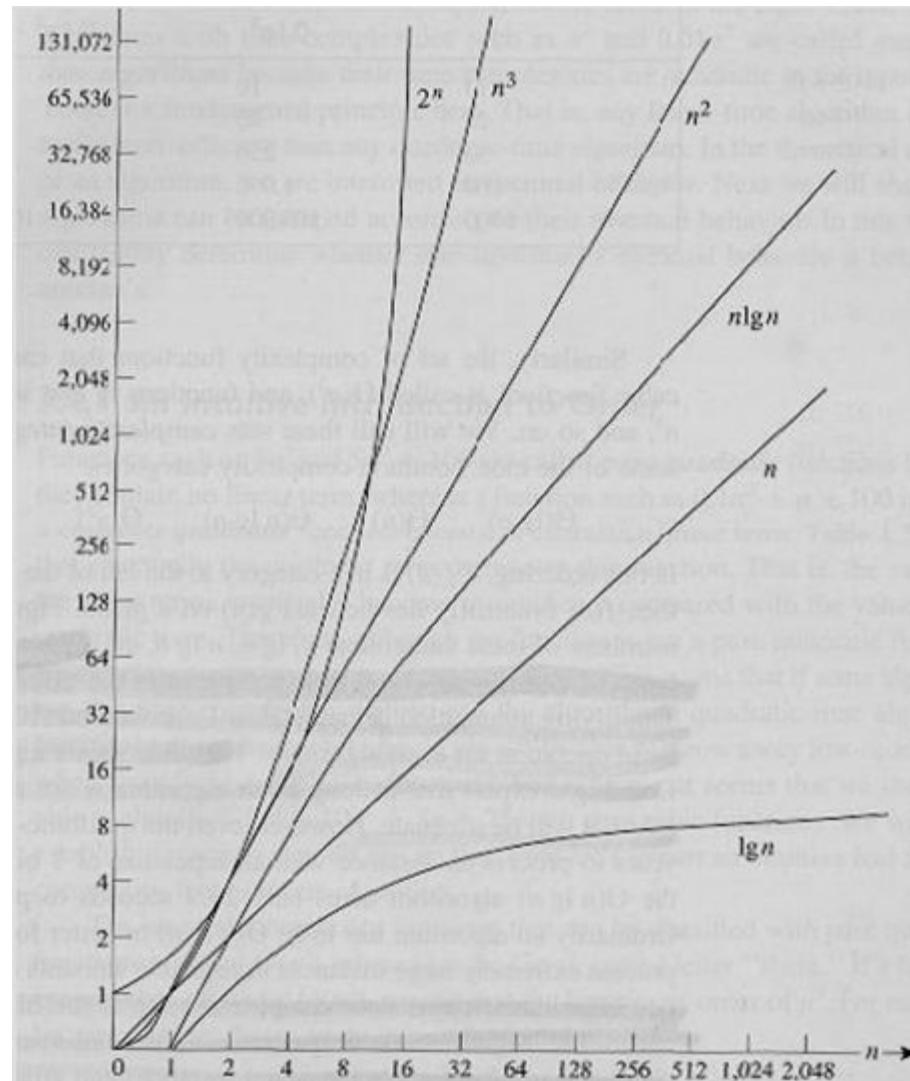


# Relations Between Different Sets

- Subset relations between order-of-growth sets.



# Common orders of magnitude



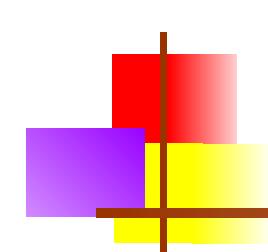
# Common orders of magnitude

**Table 1.4** Execution times for algorithms with the given time complexities

| $n$    | $f(n) = \lg n$ | $f(n) = n$   | $f(n) = n \lg n$ | $f(n) = n^2$ | $f(n) = n^3$             | $f(n) = 2^n$             |
|--------|----------------|--------------|------------------|--------------|--------------------------|--------------------------|
| 10     | 0.003 $\mu$ s* | 0.01 $\mu$ s | 0.033 $\mu$ s    | 0.1 $\mu$ s  | 1 $\mu$ s                | 1 $\mu$ s                |
| 20     | 0.004 $\mu$ s  | 0.02 $\mu$ s | 0.086 $\mu$ s    | 0.4 $\mu$ s  | 8 $\mu$ s                | 1 ms†                    |
| 30     | 0.005 $\mu$ s  | 0.03 $\mu$ s | 0.147 $\mu$ s    | 0.9 $\mu$ s  | 27 $\mu$ s               | 1 s                      |
| 40     | 0.005 $\mu$ s  | 0.04 $\mu$ s | 0.213 $\mu$ s    | 1.6 $\mu$ s  | 64 $\mu$ s               | 18.3 mir                 |
| 50     | 0.005 $\mu$ s  | 0.05 $\mu$ s | 0.282 $\mu$ s    | 2.5 $\mu$ s  | 25 $\mu$ s               | 13 days                  |
| $10^2$ | 0.007 $\mu$ s  | 0.10 $\mu$ s | 0.664 $\mu$ s    | 10 $\mu$ s   | 1 ms                     | $4 \times 10^{15}$ years |
| $10^3$ | 0.010 $\mu$ s  | 1.00 $\mu$ s | 9.966 $\mu$ s    | 1 ms         | 1 s                      |                          |
| $10^4$ | 0.013 $\mu$ s  | 0 $\mu$ s    | 130 $\mu$ s      | 100 ms       | 16.7 min                 |                          |
| $10^5$ | 0.017 $\mu$ s  | 0.10 ms      | 1.67 ms          | 10 s         | 11.6 days                |                          |
| $10^6$ | 0.020 $\mu$ s  | 1 ms         | 19.93 ms         | 16.7 min     | 31.7 years               |                          |
| $10^7$ | 0.023 $\mu$ s  | 0.01 s       | 0.23 s           | 1.16 days    | 31,709 years             |                          |
| $10^8$ | 0.027 $\mu$ s  | 0.10 s       | 2.66 s           | 115.7 days   | $3.17 \times 10^7$ years |                          |
| $10^9$ | 0.030 $\mu$ s  | 1 s          | 29.90 s          | 31.7 years   |                          |                          |

\*1  $\mu$ s =  $10^{-6}$  second.

†1 ms =  $10^{-3}$  second.



# Logarithms and properties

- In algorithm analysis we often use the notation “ $\log n$ ” without specifying the base

Binary logarithm  $\lg n = \log_2 n$

Natural logarithm  $\ln n = \log_e n$

$$\lg^k n = (\lg n)^k$$

$$\lg \lg n = \lg(\lg n)$$

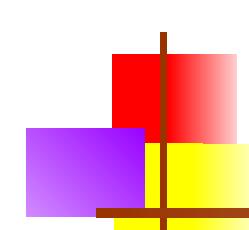
$$\log x^y = y \log x$$

$$\log xy = \log x + \log y$$

$$\log \frac{x}{y} = \log x - \log y$$

$$a^{\log_b x} = x^{\log_b a}$$

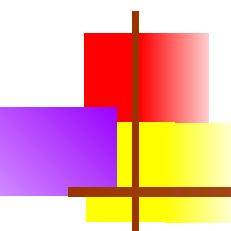
$$\log_b x = \frac{\log_a x}{\log_a b}$$



# More Examples

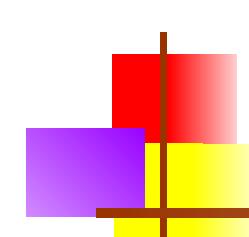
- For each of the following pairs of functions, either  $f(n)$  is  $O(g(n))$ ,  $f(n)$  is  $\Omega(g(n))$ , or  $f(n) = \Theta(g(n))$ . Determine which relationship is correct.

|                                           |                       |
|-------------------------------------------|-----------------------|
| - $f(n) = \log n^2$ ; $g(n) = \log n + 5$ | $f(n) = \Theta(g(n))$ |
| - $f(n) = n$ ; $g(n) = \log n^2$          | $f(n) = \Omega(g(n))$ |
| - $f(n) = \log \log n$ ; $g(n) = \log n$  | $f(n) = O(g(n))$      |
| - $f(n) = n$ ; $g(n) = \log^2 n$          | $f(n) = \Omega(g(n))$ |
| - $f(n) = n \log n + n$ ; $g(n) = \log n$ | $f(n) = \Omega(g(n))$ |
| - $f(n) = 10$ ; $g(n) = \log 10$          | $f(n) = \Theta(g(n))$ |
| - $f(n) = 2^n$ ; $g(n) = 10n^2$           | $f(n) = \Omega(g(n))$ |
| - $f(n) = 2^n$ ; $g(n) = 3^n$             | $f(n) = O(g(n))$      |



# Properties

- *Theorem:*
$$f(n) = \Theta(g(n)) \Leftrightarrow f = O(g(n)) \text{ and } f = \Omega(g(n))$$
- **Transitivity:**
  - $f(n) = \Theta(g(n))$  and  $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - Same for  $O$  and  $\Omega$
- **Reflexivity:**
  - $f(n) = \Theta(f(n))$
  - Same for  $O$  and  $\Omega$
- **Symmetry:**
  - $f(n) = \Theta(g(n))$  if and only if  $g(n) = \Theta(f(n))$
- **Transpose symmetry:**
  - $f(n) = O(g(n))$  if and only if  $g(n) = \Omega(f(n))$



# Asymptotic Notations in Equations

- On the right-hand side

- $\Theta(n^2)$  stands for some anonymous function in  $\Theta(n^2)$

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$  means:

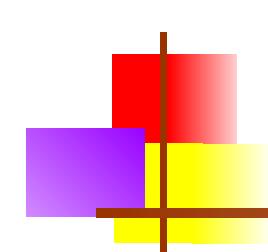
There exists a function  $f(n) \in \Theta(n)$  such that

$$2n^2 + 3n + 1 = 2n^2 + f(n)$$

- On the left-hand side

$$2n^2 + \Theta(n) = \Theta(n^2)$$

No matter how the anonymous function is chosen on the left-hand side, there is a way to choose the anonymous function on the right-hand side to make the equation valid.



# Common Summations

- Arithmetic series:

$$\sum_{k=1}^n k = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

- Geometric series:

- Special case:  $|x| < 1$ :

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \dots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

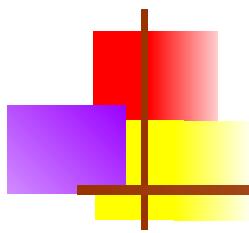
- Harmonic series:

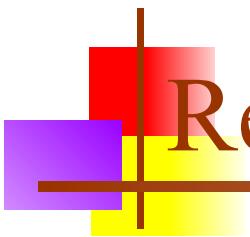
$$\sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \dots + \frac{1}{n} \approx \ln n$$

- Other important formulas:

$$\sum_{k=1}^n \lg k \approx n \lg n$$

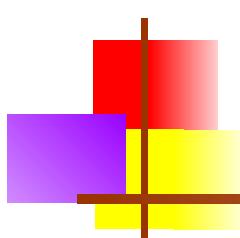
$$\sum_{k=1}^n k^p = 1^p + 2^p + \dots + n^p \approx \frac{1}{p+1} n^{p+1}$$





# Recurrence Relation

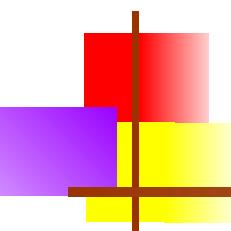




# Recurrence Relation

---

- ❑ A recurrence relation for the sequence,  $a_0, a_1, \dots, a_n$ , is an equation that relates  $a_n$  to certain of its predecessors  $a_0, a_1, \dots, a_{n-1}$ .
  
- ❑ Initial conditions for the sequence  $a_0, a_1, \dots$  are explicitly given values for a finite number of the terms of the sequence.



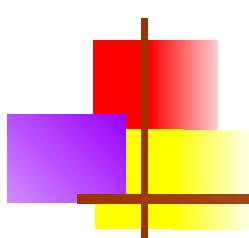
# Definition

- A recurrence relation,  $T(n)$ , is a recursive function of integer variable  $n$
- Like all recursive functions, it has both recursive case and base case.

- Example:

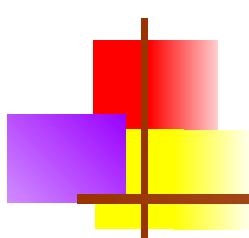
$$T(n) = \begin{cases} a & \text{if } n = 1 \\ 2T(n / 2) + bn + c & \text{if } n > 1 \end{cases}$$

- The portion of the definition that does not contain  $T$  is called the **base case** of the recurrence relation; the portion that contains  $T$  is called the **recurrent or recursive case**.
- Recurrence relations are useful for expressing the running times (i.e., the number of basic operations executed) of recursive algorithms



# Example: Fibonacci Sequence

- The Fibonacci sequence is defined by recurrence relation
- $f_n = f_{n-1} + f_{n-2}$ ,  $n \geq 3$  and initial conditions
- $f_1 = 1, f_2 = 1.$
- Fibonacci Sequence: 1, 1, 2, 3, 5, 8, 13, 21,.....



# Forming Recurrence Relation

- ❑ For a given recursive method, the base case and the recursive case of its recurrence relation correspond directly to the base case and the recursive case of the method.
- ❑ Example 1: Write the recurrence relation for the following method.

```
void f(int n)
{  if (n >
0)  {
    cout<<n;
    f(n-1);
```

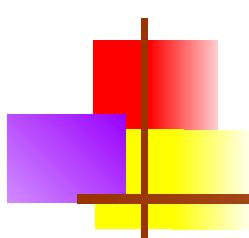
- ❑ The base case is reached when  $n == 0$ . The method performs one comparison. Thus, the number of operations when  $n == 0$ ,  $T(0)$ , is some constant  $a$ .
- ❑ When  $n > 0$ , the method performs two basic operations and then calls itself, using ONE recursive call, with a parameter  $n - 1$ .
- ❑ Therefore the recurrence relation is:

$$T(0) = a$$

$$T(n) = b + T(n-1)$$

where  $a$  is constant

where  $b$  is constant,  $n > 0$



# Forming Recurrence Relation

- Example 2: Write the recurrence relation for the following method.

```
int g(int n) {  
    if (n == 1)  
        return 2;  
    else  
        return 3 * g(n / 2) + g( n / 2) + 5;  
}
```

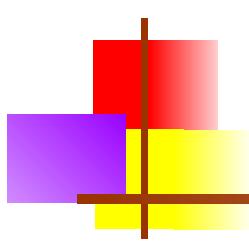
- The base case is reached when  $n == 1$ . The method performs one comparison and one return statement. Therefore,  $T(1)$ , is constant  $c$ .
- When  $n > 1$ , the method performs TWO recursive calls, each with the parameter  $n/2$ , and some constant # of basic operations.
- Hence, the recurrence relation is:

$$T(1) = c$$

for some constant  $c$

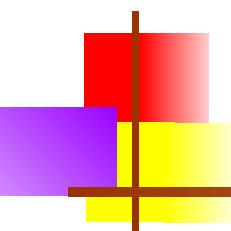
$$T(n) = b + 2T(n / 2)$$

for a constant  $b$



# Solving Recurrence Relation

- ❑ To solve a recurrence relation  $T(n)$  we need to derive a form of  $T(n)$  that is not a recurrence relation. Such a form is called a “closed form” of the recurrence relation.
- ❑ There are four methods to solve recurrence relations that represent the running time of recursive methods:
  - ❑ Iteration method (unrolling and summing)
  - ❑ Recursion Tree method
  - ❑ Substitution method
  - ❑ Master method

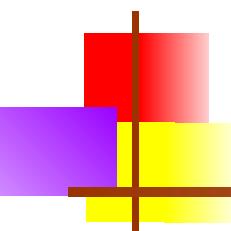


# Solving Recurrence Relations - Iteration method

---

Steps:

- Expand the recurrence
- Express the expansion as a summation by plugging the recurrence back into itself until you see a pattern.
- Evaluate the summation



# Solving Recurrence Relations - Iteration method

- In evaluating the summation one or more of the following summation formulae may be used:
- Arithmetic series:

$$\sum_{k=1}^n k = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

- Geometric Series:

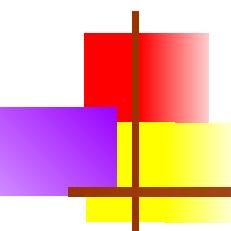
Special Cases of Geometric Series:

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \dots + x^n = \frac{x^{n+1} - 1}{x - 1} \quad (x \neq 1)$$

$$\sum_{k=0}^{n-1} 2^k = 2^n - 1$$

$$\sum_{k=0}^{n-1} x^k = \frac{x^n - 1}{x - 1} \quad (x \neq 1)$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad \text{if } x < 1$$



# Solving Recurrence Relations - Iteration method

- Harmonic Series

$$\sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \dots + \frac{1}{n} \approx \ln n$$

- Others:

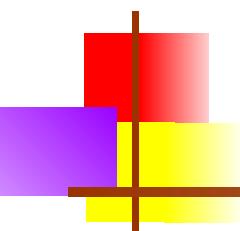
$$\sum_{k=1}^n \lg k \approx n \lg n$$

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=0}^{n-1} c = cn.$$

$$\sum_{k=0}^{n-1} \frac{1}{2^k} = 2 - \frac{1}{2^{n-1}}$$

$$\sum_{k=0}^n k(k+1) = \frac{n(n+1)(n+2)}{3}$$



# Analysis Of Recursive Factorial Method

- Example 1: Form and solve the recurrence relation for the running time of factorial method and hence determine its big-O complexity:

- $T(0) = c$

$$T(n) = b + T(n - 1)$$

$$= b + b + T(n - 2)$$

$$= b + b + b + T(n - 3)$$

...

$$= kb + T(n - k)$$

When  $k = n$ , we have:

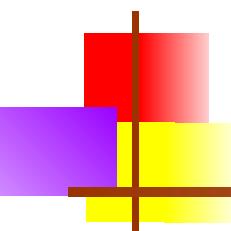
$$T(n) = nb + T(n - n)$$

$$= bn + T(0)$$

$$= bn + c.$$

Therefore method factorial is  $O(n)$ .

```
long factorial (int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial (n - 1);  
}
```



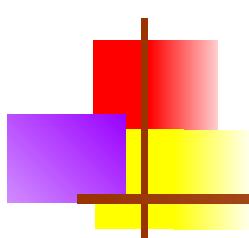
# Analysis Of Recursive Binary Search

```
public int binarySearch (int target, int[] array,
                      int low, int high) {
    if (low >
        high)
        return -1;
    else {
        int middle = (low + high)/2;
        if (array[middle] == target)
            return middle;
        else if(array[middle] < target)
            return binarySearch(target, array, middle + 1,
high);  else
            return binarySearch(target, array, low, middle - 1);
    }
}
```

The recurrence relation for the running time of the method is:

$$T(1) = a \quad \text{if } n = 1 \text{ (one element array)}$$

$$T(n) = T(n / 2) + b \quad \text{if } n > 1$$



# Analysis Of Recursive Binary Search

Expanding:

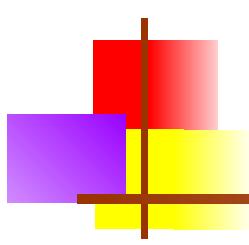
$$\begin{aligned}T(n) &= T(n / 2) + b \\&= [T(n / 4) + b] + b = T(n / 2^2) + 2b \\&= [T(n / 8) + b] + 2b = T(n / 2^3) + 3b \\&= \dots\dots\dots \\&= T(n / 2^k) + kb\end{aligned}$$

When  $n / 2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$ , we have:

$$T(n) = T(1) + b \log_2 n$$

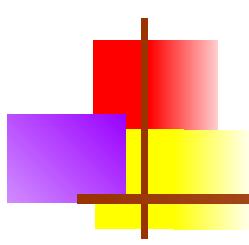
$$= a + b \log_2 n$$

Therefore, Recursive Binary Search is **O(log n)**



# Recursion-tree method

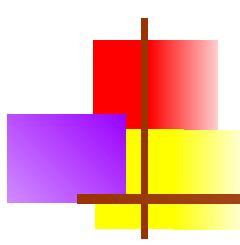
- ❑ A recursion tree models the costs (time) of a recursive execution of an algorithm.
- ❑ The recursion tree method is good for generating guesses for the substitution method.
- ❑ The recursion-tree method can be unreliable.
- ❑ The recursion-tree method promotes intuition, however.
- ❑ In this case, only the largest term in the geometric series matters; all of the other terms are swallowed up by the  $\Theta(\cdot)$  notation.



# Example of recursion tree

Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :

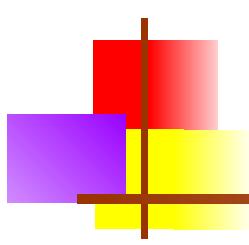




# Example of recursion tree

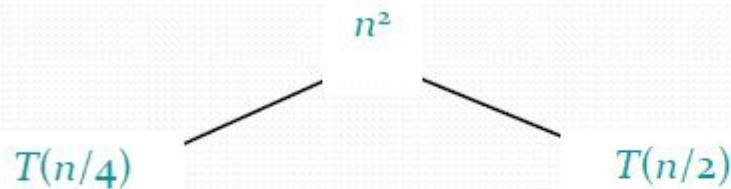
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :

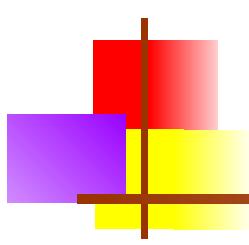
$$T(n)$$



# Example of recursion tree

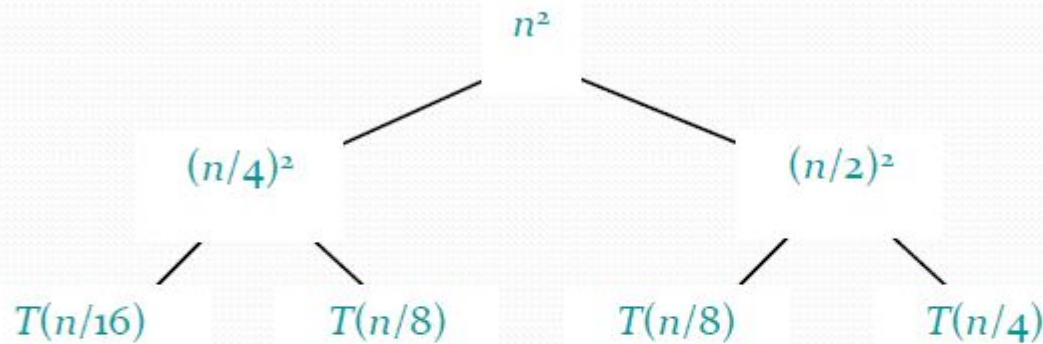
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :





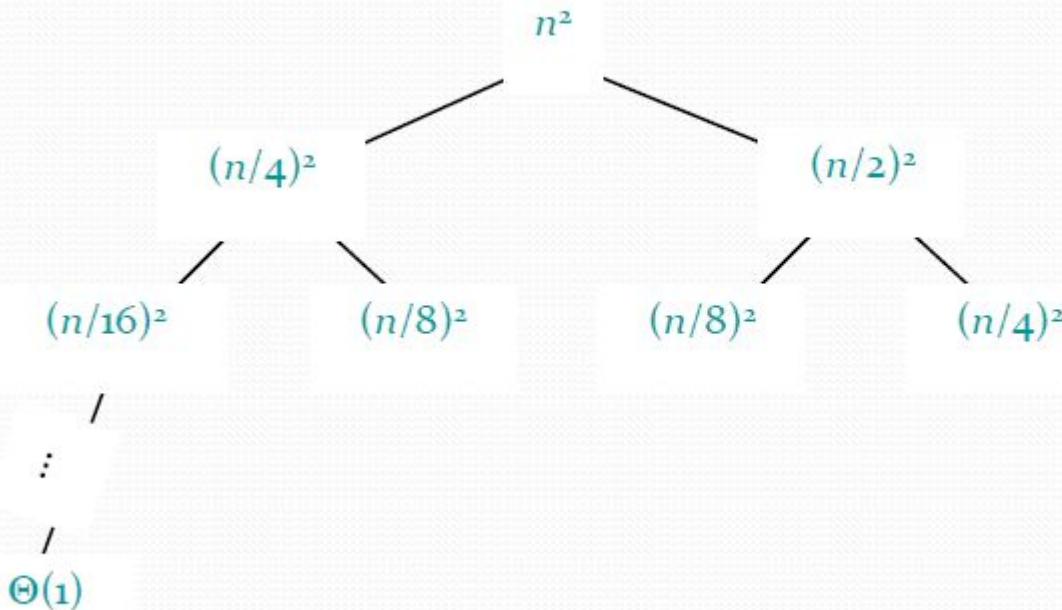
# Example of recursion tree

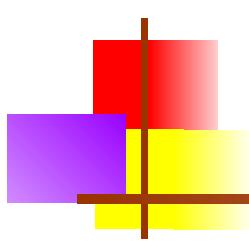
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :



# Example of recursion tree

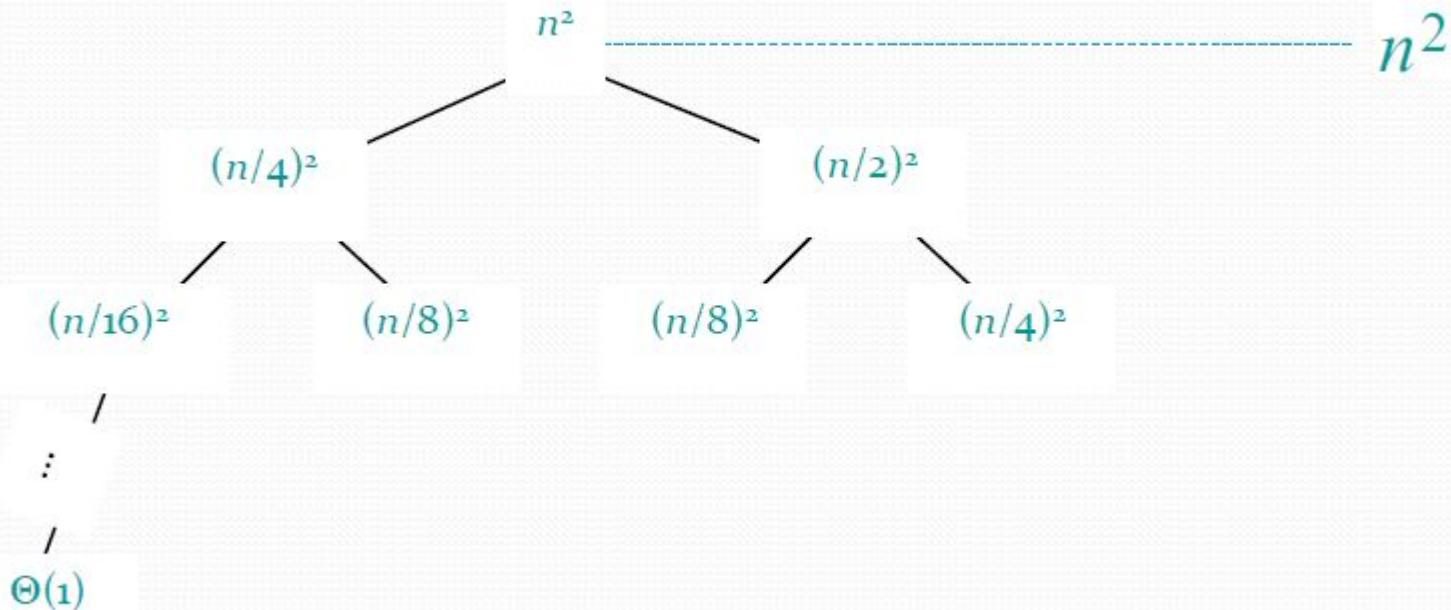
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :





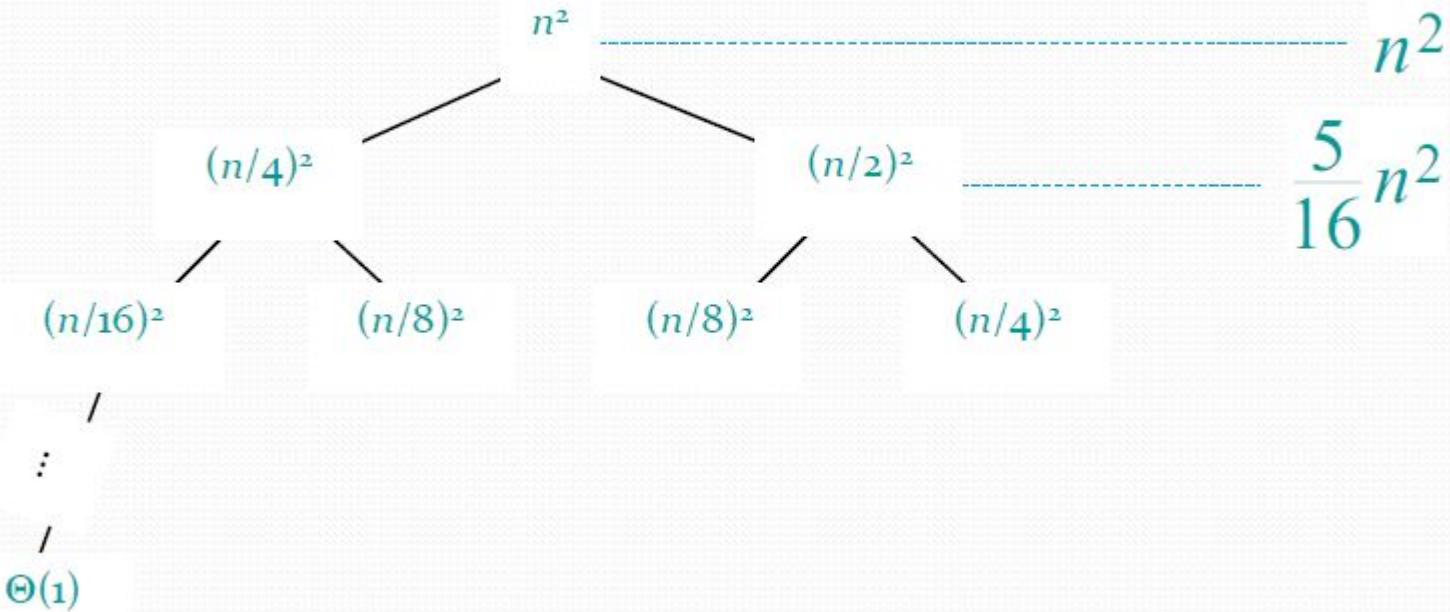
# Example of recursion tree

Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :



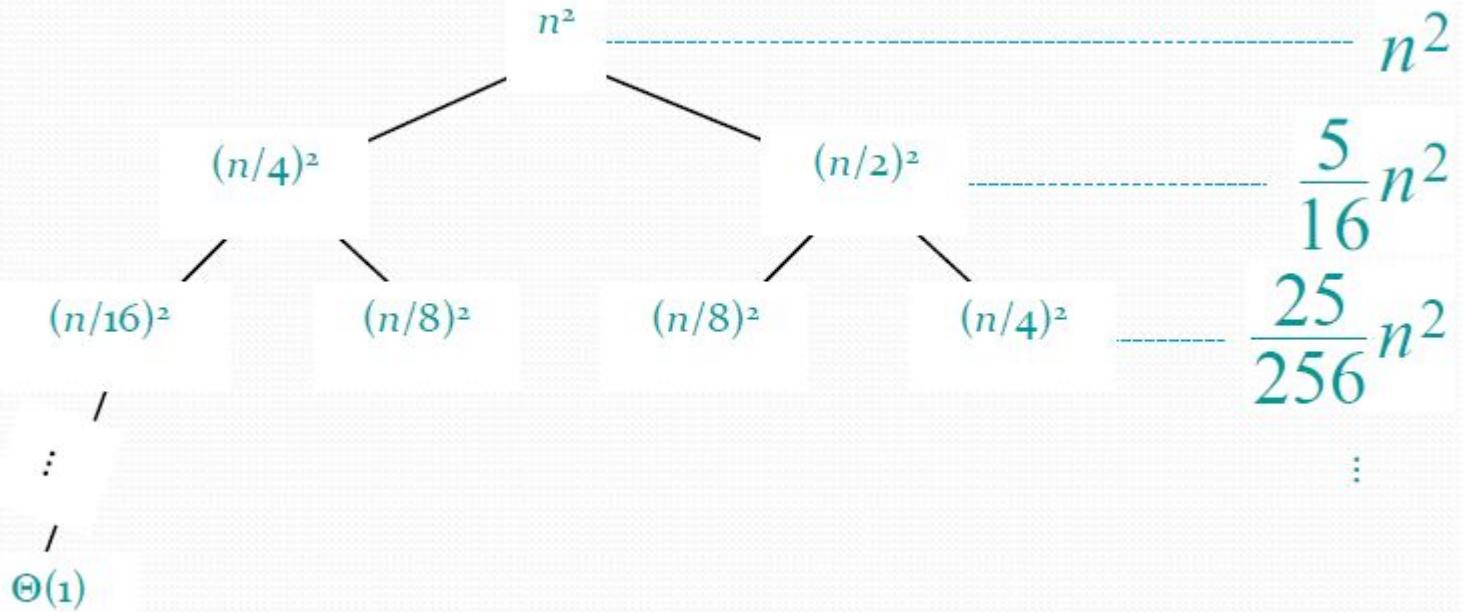
# Example of recursion tree

Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :



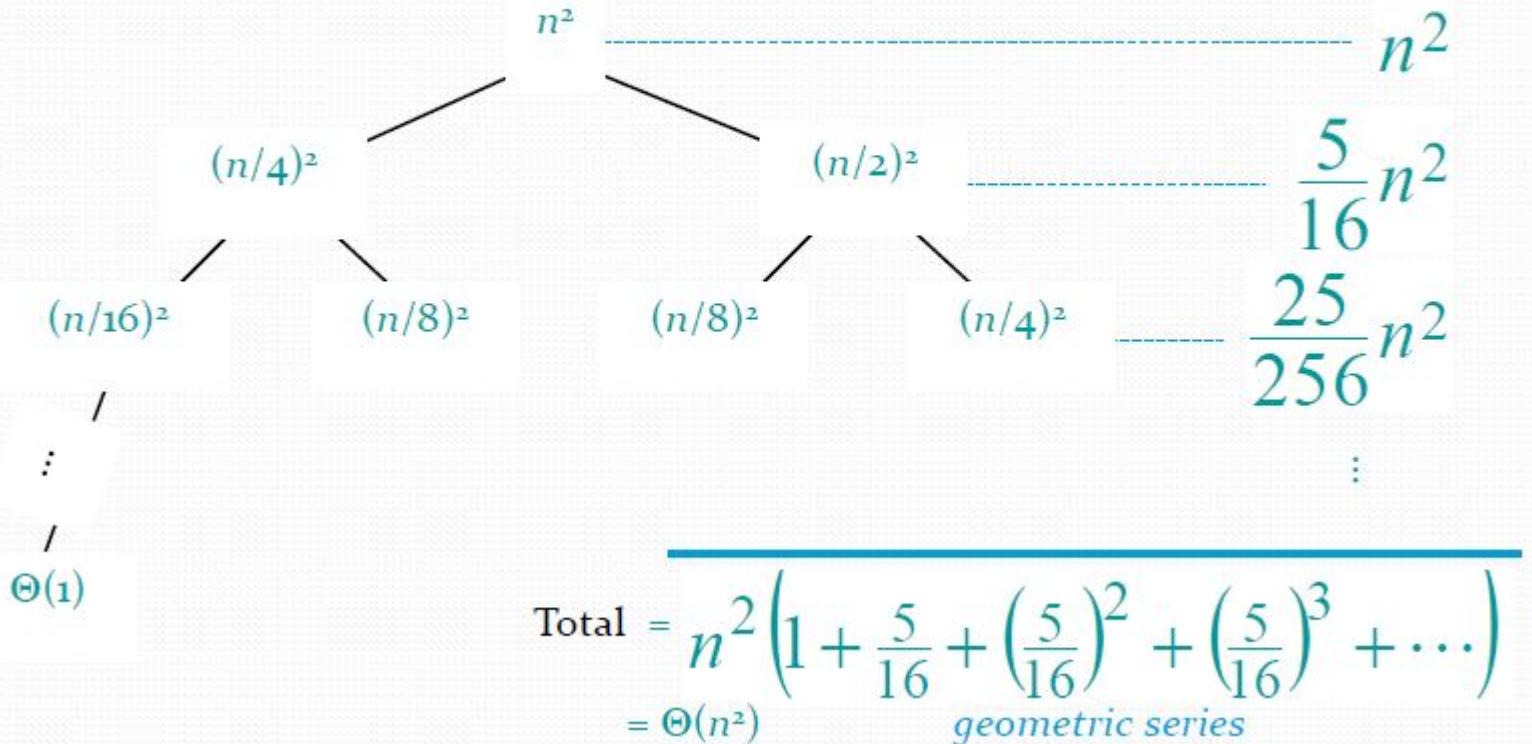
# Example of recursion tree

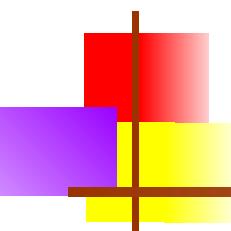
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :



# Example of recursion tree

Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :

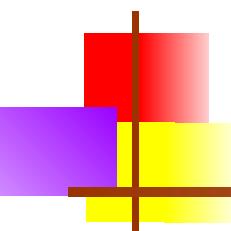




# Solving Recurrence Relations-Substitution Method

---

- ❑ The substitution method
  - ❑ A.k.a. the “making a good guess method”
  - ❑ Guess the form of the answer, then use induction to find the constants and show that solution works
  - ❑ Run an example: merge sort
    - ❑  $T(n) = 2T(n/2) + cn$
    - ❑ We guess that the answer is  $O(n \lg n)$
    - ❑ Prove it by induction
  - ❑ Can similarly show  $T(n) = \Omega(n \lg n)$ , thus  $\Theta(n \lg n)$  24



# Solving Recurrence Relations-Substitution Method

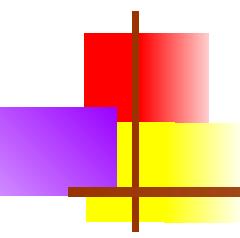
---

Example:  $T(n) = 2T(n/2) + n$

- ❑ Guess  $T(n) \leq cn \log n$  for some constant  $c$  (that is,  $T(n) = O(n \log n)$ )
- ❑  $T(n) = 2T(n/2) + n \leq 2(c n/2 \log n/2) + n$ 
  - ❑  $= cn \log n/2 + n$
  - ❑  $= cn \log n - cn \log 2 + n$
  - ❑  $= cn \log n - cn + n$

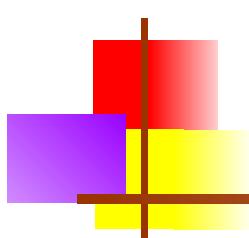
if  $c \geq 1$  , condition is satisfied.

Thus,  $T(n) = O(n \log n)$



# Solving Recurrence Relations- The Master Theorem

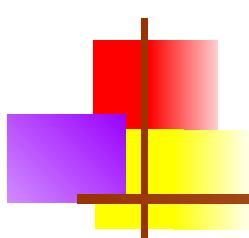
- ❑ Given: a divide and conquer algorithm
  - ❑ An algorithm that divides the problem of size  $n$  into  $a$  subproblems, each of size  $n/b$
  - ❑ Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function  $f(n)$ .
- ❑ Then, the Master Theorem gives us a method for the algorithm's running time:



# The Master Theorem

- if  $T(n) = aT(n/b) + f(n)$  then (where  $a \geq 1$ ,  $b > 1$ )

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \varepsilon}) \\ \Theta(n^{\log_b a} \log n) & f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \varepsilon}) \text{ AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{cases} \begin{cases} \varepsilon > 0 \\ c < 1 \end{cases}$$



# The Master Theorem

$$T(n) = 9T(n/3) + n$$

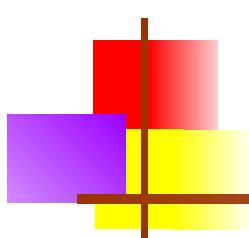
- $a=9, b=3, f(n) = n$
- $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$
- Since  $f(n) = n, f(n) < n^{\log_b a}$
- **Case 1 applies:**

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \varepsilon}) \\ \Theta(n^{\log_b a} \log n) & f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \varepsilon}) \text{ AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{cases}$$

$\varepsilon > 0$   
 $c < 1$

$$T(n) = \Theta(n^{\log_b a}) \text{ when } f(n) = O(n^{\log_b a - \varepsilon})$$

- Thus the solution is  $T(n) = \Theta(n^2)$



# The Master Theorem

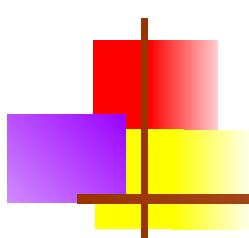
- $T(n) = 4T(n/2) + n^2$
- $a = 4, b = 2, f(n) = n^2$
- $n^{\log_b a} = n^2$
- Since,  $f(n) = n^2$
- Thus,  $f(n) = n^{\log_b a}$

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \varepsilon}) \\ \Theta(n^{\log_b a} \log n) & f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \varepsilon}) \text{ AND } af(n/b) < cf(n) \text{ for large } n \end{cases} \begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}$$

- **Case 2 applies:**

$$f(n) = \Theta(n^2 \log n)$$

- Thus the solution is  $T(n) = \Theta(n^2 \log n)$ .



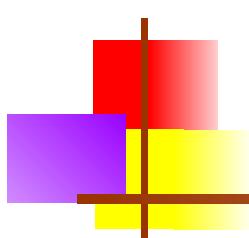
# The Master Theorem

*Ex.*  $T(n) = 4T(n/2) + n^3$

- $a = 4, b = 2, f(n) = n^3$
- $n^{\log_b a} = n^2; f(n) = n^3.$
- Since,  $f(n) = n^3$
- Thus,  $f(n) > n^{\log_b a}$
  
- **Case 3 applies:**  

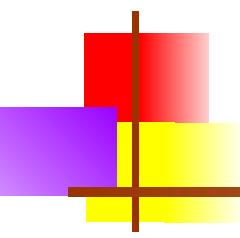
$$f(n) = \Omega(n^3)$$
- **and**  $4(n/2)^3 \leq cn^3$  (regulatory condition) for  $c = 1/2.$

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \varepsilon}) \\ \Theta(n^{\log_b a} \log n) & f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \varepsilon}) \text{ AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{cases} \begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}$$



# Some Common Recurrence Relation

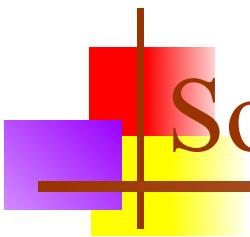
| Recurrence Relation        | Complexity    | Problem                        |
|----------------------------|---------------|--------------------------------|
| $T(n) = T(n/2) + c$        | $O(\log n)$   | Binary Search                  |
| $T(n) = 2T(n-1) + c$       | $O(2^n)$      | Tower of Hanoi                 |
| $T(n) = T(n-1) + c$        | $O(n)$        | Linear Search                  |
| $T(n) = 2T(n/2) + n$       | $O(n \log n)$ | Merge Sort                     |
| $T(n) = T(n-1) + n$        | $O(n^2)$      | Selection Sort, Insertion Sort |
| $T(n) = T(n-1)+T(n-2) + c$ | $O(2^n)$      | Fibonacci Series               |



# Summary

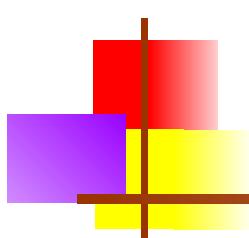
---

- ❑ Recurrence Relation
  - ❑ Iteration method (unrolling and summing)
  - ❑ Recursion Tree method
  - ❑ Substitution method
  - ❑ Master method



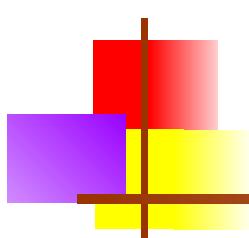
# Sorting in Linear Time





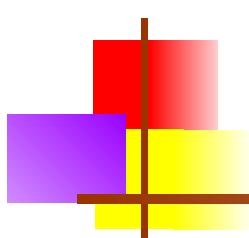
# Comparison Sorting Review

- ❑ Insertion sort:
  - ❑ Pro's:
    - ❑ Easy to code
    - ❑ Fast on small inputs (less than ~50 elements)
    - ❑ Fast on nearly-sorted inputs
  - ❑ Con's:
    - ❑  $O(n^2)$  worst case
    - ❑  $O(n^2)$  average case
    - ❑  $O(n^2)$  reverse-sorted case



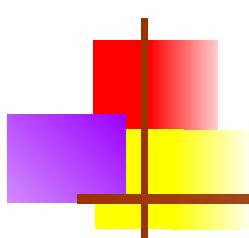
# Comparison Sorting Review

- ❑ Merge sort:
  - ❑ Divide-and-conquer:
    - ❑ Split array in half
    - ❑ Recursively sort sub-arrays
    - ❑ Linear-time merge step
  - ❑ Pro's:
    - ❑  $O(n \lg n)$  worst case - asymptotically optimal for comparison sorts
  - ❑ Con's:
    - ❑ Doesn't sort in place



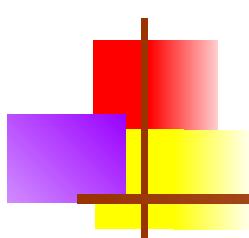
# Comparison Sorting Review

- ❑ Heap sort:
  - ❑ Uses the very useful heap data structure
    - ❑ Complete binary tree
    - ❑ Heap property: parent key > children's keys
  - ❑ Pro's:
    - ❑  $O(n \lg n)$  worst case - asymptotically optimal for comparison sorts
    - ❑ Sorts in place
  - ❑ Con's:
    - ❑ Fair amount of shuffling memory around



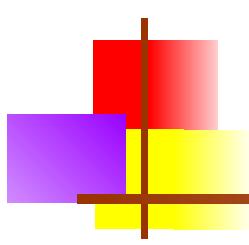
# Comparison Sorting Review

- ❑ Quick sort:
  - ❑ Divide-and-conquer:
    - ❑ Partition array into two sub-arrays, recursively sort
    - ❑ All of first sub-array < all of second sub-array
  - ❑ Pro's:
    - ❑  $O(n \lg n)$  average case
    - ❑ Sorts in place
    - ❑ Fast in practice (why?)
  - ❑ Con's:
    - ❑  $O(n^2)$  worst case
      - ❑ Naïve implementation: worst case on sorted input
      - ❑ Good partitioning makes this very unlikely.



# Non-Comparison Based Sorting

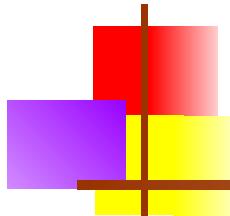
- ❑ Many times we have restrictions on our keys
  - ❑ Social Security Numbers
  - ❑ Employee ID's
- ❑ We will examine three algorithms which under certain conditions can run in  $O(n)$  time.
  - ❑ Counting sort
  - ❑ Radix sort
  - ❑ Bucket sort



# Counting Sort

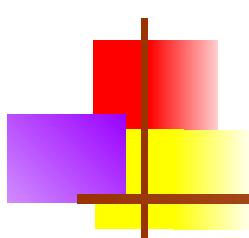
---

- ❑ Why it's not a comparison sort:
  - ❑ Assumption: input - integers in the range 0..k
  - ❑ No comparisons made!
- ❑ Basic idea:
  - ❑ determine for each input element  $x$  its rank: the number of elements less than  $x$ .
  - ❑ once we know the rank  $r$  of  $x$ , we can place it in position  $r+1$
- ❑ Depends on assumption about the numbers being sorted
  - ❑ Assume numbers are in the range 1.. k
- ❑ The algorithm:
  - ❑ Input:  $A[1..n]$ , where  $A[j] \in \{1, 2, 3, \dots, k\}$
  - ❑ Output:  $B[1..n]$ , sorted (not sorted in place)
  - ❑ Also: Array  $C[1..k]$  for auxiliary storage



# Counting Sort Example

|       |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|
|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| $A =$ | 2 | 5 | 3 | 0 | 2 | 3 | 0 | 3 |
| $C =$ | 0 | 1 | 2 | 3 | 4 | 5 |   |   |
|       | 2 | 0 | 2 | 3 | 0 | 1 |   |   |
| $C =$ | 0 | 1 | 2 | 3 | 4 | 5 |   |   |
|       | 2 | 2 | 4 | 7 | 7 | 8 |   |   |
| $B =$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|       |   |   |   |   |   |   | 3 |   |
| $C =$ | 0 | 1 | 2 | 3 | 4 | 5 |   |   |
|       | 2 | 2 | 4 | 6 | 7 | 8 |   |   |



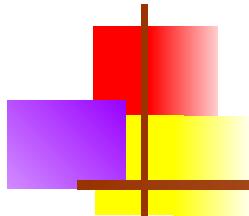
# Counting Sort Example

|       |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|
|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| $A =$ | 2 | 5 | 3 | 0 | 2 | 3 | 0 | 3 |

|       |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|
|       | 0 | 1 | 2 | 3 | 4 | 5 |
| $C =$ | 2 | 2 | 4 | 6 | 7 | 8 |

|       |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|
|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| $B =$ |   | 0 |   |   |   |   |   | 3 |

|       |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|
|       | 0 | 1 | 2 | 3 | 4 | 5 |
| $C =$ | 1 | 2 | 4 | 6 | 7 | 8 |



# Counting Sort Example

$$A = \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boxed{2} & \boxed{5} & \boxed{3} & \boxed{0} & \boxed{2} & \boxed{3} & \boxed{0} & \boxed{3} \end{array}$$
$$C = \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ \boxed{2} & \boxed{2} & \boxed{4} & \boxed{6} & \boxed{7} & \boxed{8} \end{array}$$
$$B = \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ & & & & & \boxed{3} & \boxed{3} & \\ 0 & 1 & 2 & 3 & 4 & 5 & & \end{array}$$
$$C = \begin{array}{cccccc} 1 & 2 & 4 & 5 & 7 & 8 \\ \boxed{1} & \boxed{2} & \boxed{4} & \boxed{5} & \boxed{7} & \boxed{8} \end{array}$$

# Counting Sort

```
1   CountingSort(A, B, k)
```

```
2       for i=1 to k
```

```
3           C[i] = 0;
```

```
4       for j=1 to n
```

```
5           C[A[j]] += 1;
```

```
6       for i=2 to k
```

```
7           C[i] = C[i] + C[i-1];
```

```
8       for j=n downto 1
```

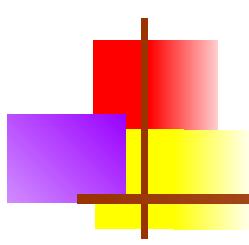
```
9           B[C[A[j]]] = A[j];
```

```
10          C[A[j]] -= 1;
```

*Takes time  $O(k)$*

*Takes time  $O(n)$*

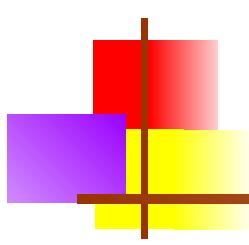
*What will be the running time?*



# Counting Sort

---

- ❑ Total time:  $O(n + k)$ 
  - ❑ Usually,  $k = O(n)$
  - ❑ Thus counting sort runs in  $O(n)$  time
- ❑ But sorting is  $(n \lg n)!$ 
  - ❑ No contradiction--this is not a comparison sort (in fact, there are no comparisons at all!)
  - ❑ Notice that this algorithm is stable
  - ❑ If numbers have the same value, they keep their original order



# Stable Sorting Algorithms

- A sorting algorithm is stable if for any two indices  $i$  and  $j$  with  $i < j$  and  $a_i = a_j$ , element  $a_i$  precedes element  $a_j$  in the output sequence.

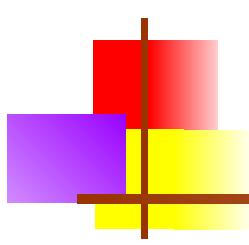
Input

|                |                |                |                |                |                |                |                |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 2 <sub>1</sub> | 7 <sub>1</sub> | 4 <sub>1</sub> | 4 <sub>2</sub> | 2 <sub>2</sub> | 5 <sub>1</sub> | 2 <sub>3</sub> | 6 <sub>1</sub> |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|

Output

|                |                |                |                |                |                |                |                |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 2 <sub>1</sub> | 2 <sub>2</sub> | 2 <sub>3</sub> | 4 <sub>1</sub> | 4 <sub>2</sub> | 5 <sub>1</sub> | 6 <sub>1</sub> | 7 <sub>1</sub> |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|

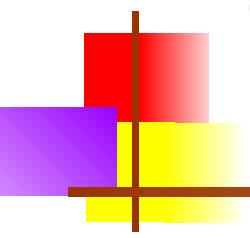
**Observation:** *Counting Sort is stable.*



# Counting Sort

---

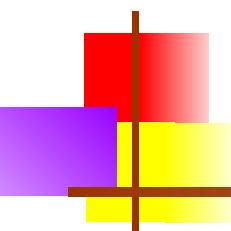
- ❑ Linear Sort! Cool! Why don't we always use counting sort?
- ❑ Because it depends on range  $k$  of elements
- ❑ Could we use counting sort to sort 32 bit integers?  
Why or why not?
- ❑ Answer: no,  $k$  too large ( $2^{32} = 4,294,967,296$ )



# Counting Sort

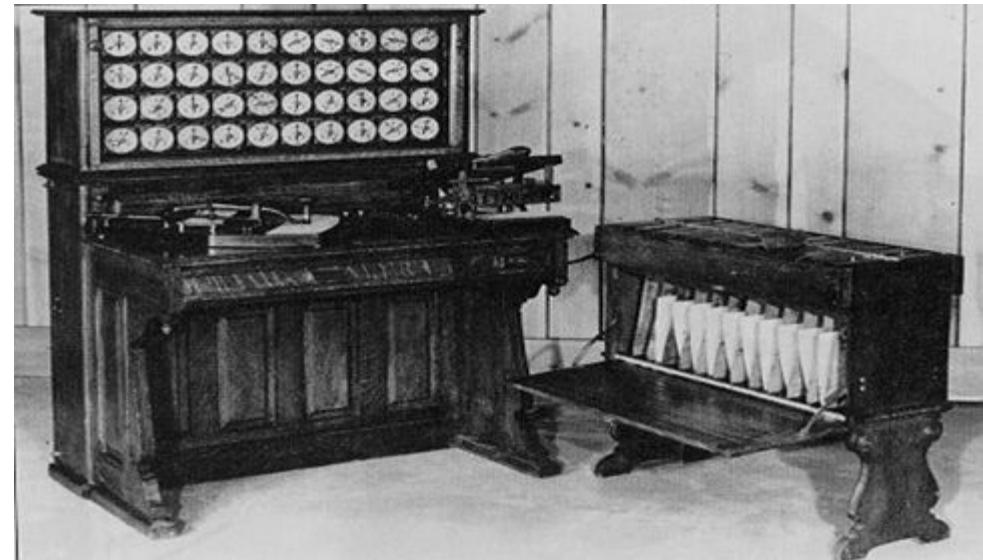
---

- ❑ Assumption: input taken from small set of numbers of size  $k$
- ❑ Basic idea:
  - ❑ Count number of elements less than you for each element.
  - ❑ This gives the position of that number – similar to selection sort.
- ❑ Pro's:
  - ❑ Fast
  - ❑ Asymptotically fast -  $O(n+k)$
  - ❑ Simple to code
- ❑ Con's:
  - ❑ Doesn't sort in place.
  - ❑ Elements must be integers. ***countable***
  - ❑ Requires  $O(n+k)$  extra storage.

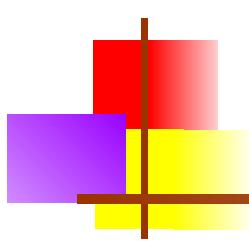


# Radix Sort

- ❑ Origin : Herman Hollerith's card-sorting machine for the 1890 U.S Census



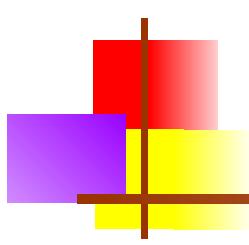
- ❑ Digit-by-digit sort
- ❑ Hollerith's original (bad) idea : sort on most-significant digit first.
- ❑ Good idea : Sort on least-significant digit first with auxiliary stable sort



# Radix Sort

IBM 083  
**punch card**  
sorter





# Radix Sort

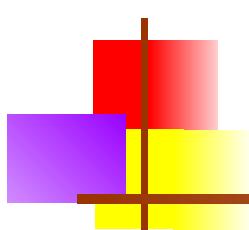
---

- ❑ Intuitively, you might sort on the most significant digit, then the second msd, etc.
- ❑ Problem: lots of intermediate piles of cards (read: scratch arrays) to keep track of
- ❑ Key idea: sort the least significant digit first

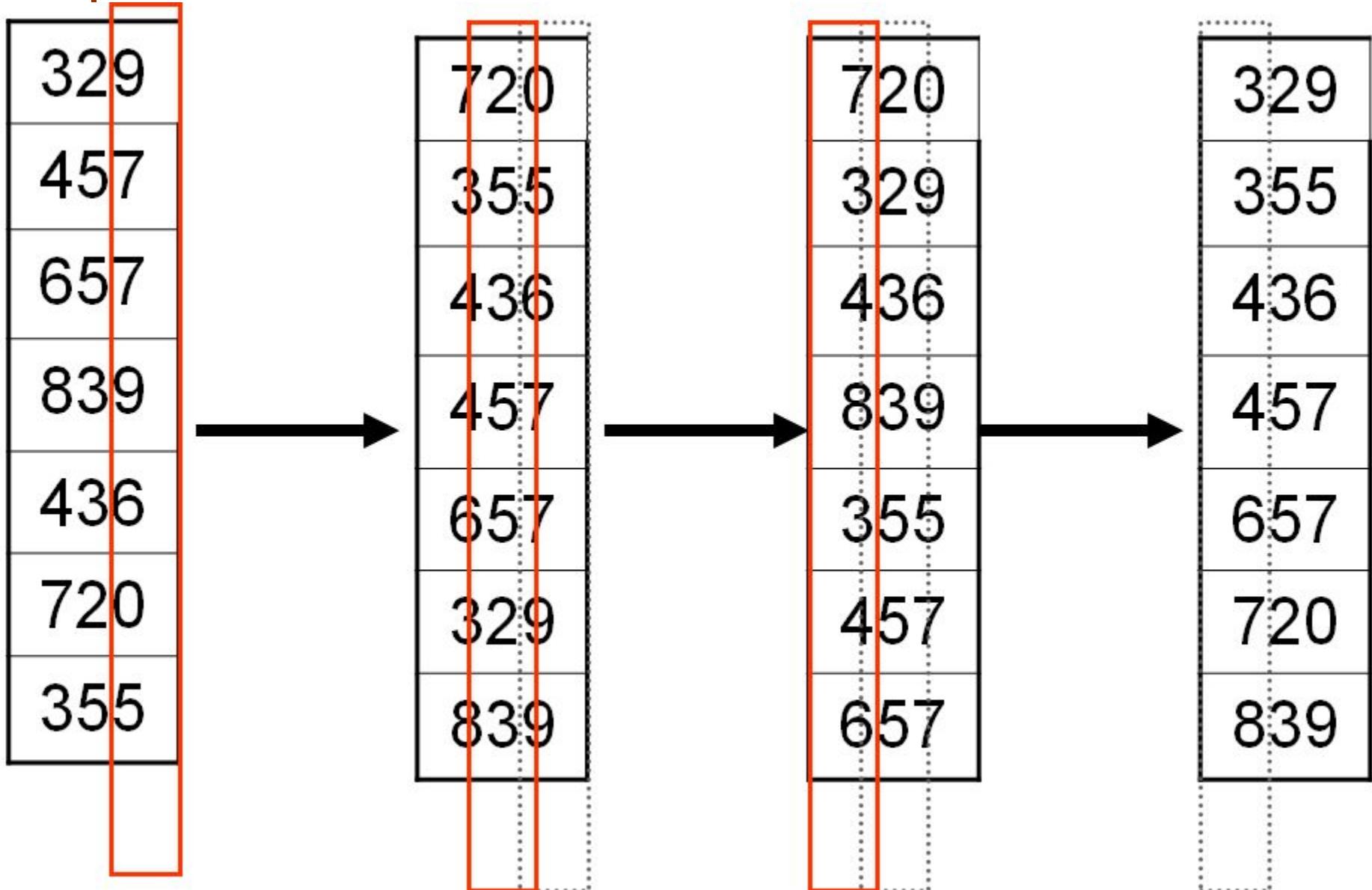
RadixSort( $A, d$ )

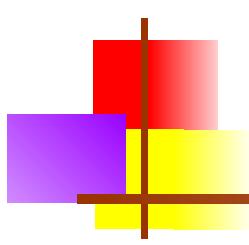
for  $i=1$  to  $d$

StableSort( $A$ ) on digit  $i$



# Radix Sort Example

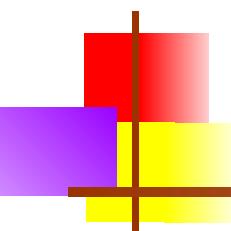




# Radix Sort

---

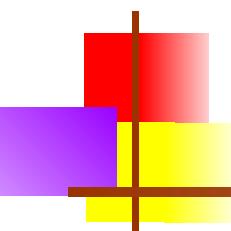
- ❑ What is the running time of radix sort?
  - ❑ Each pass over the  $d$  digits takes time  $O(n+k)$ , so total time  $O(dn+dk)$ 
    - ❑ When  $d$  is constant and  $k=O(n)$ , takes  $O(n)$  time
- ❑ Stable, Fast
- ❑ Doesn't sort in place (because counting sort is used)



# Radix Sort

---

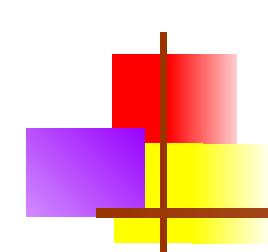
- ❑ Problem: sort 1 million 64-bit numbers
  - ❑ Treat as four-digit radix 216 numbers
  - ❑ Can sort in just four passes with radix sort!
- ❑ Performs well compared to typical  $O(n \lg n)$  comparison sort
  - ❑ Approx  $\lg(1,000,000)$  20 comparisons per number being sorted



# Radix Sort

---

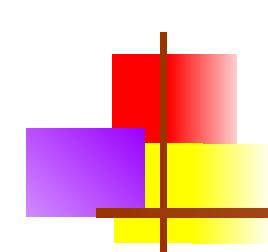
- ❑ Assumption: input has  $d$  digits ranging from 0 to  $k$
- ❑ Basic idea:
  - ❑ Sort elements by digit starting with least significant
  - ❑ Use a stable sort (like counting sort) for each stage
- ❑ Pro's:
  - ❑ Fast
  - ❑ Asymptotically fast (i.e.,  $O(n)$  when  $d$  is constant and  $k=O(n)$ )
  - ❑ Simple to code
  - ❑ A good choice
- ❑ Con's:
  - ❑ Doesn't sort in place
  - ❑ Not a good choice for floating point numbers or arbitrary strings.



# Bucket Sort

---

- ❑ Assumption: input - n real numbers from  $[0, 1)$
- ❑ Basic idea:
  - ❑ Create n linked lists (buckets) to divide interval  $[0, 1)$  into subintervals of size  $1/n$
  - ❑ Add each input element to appropriate bucket and sort buckets with insertion sort
- ❑ Uniform input distribution  $O(1)$  bucket size
  - ❑ Therefore the expected total time is  $O(n)$



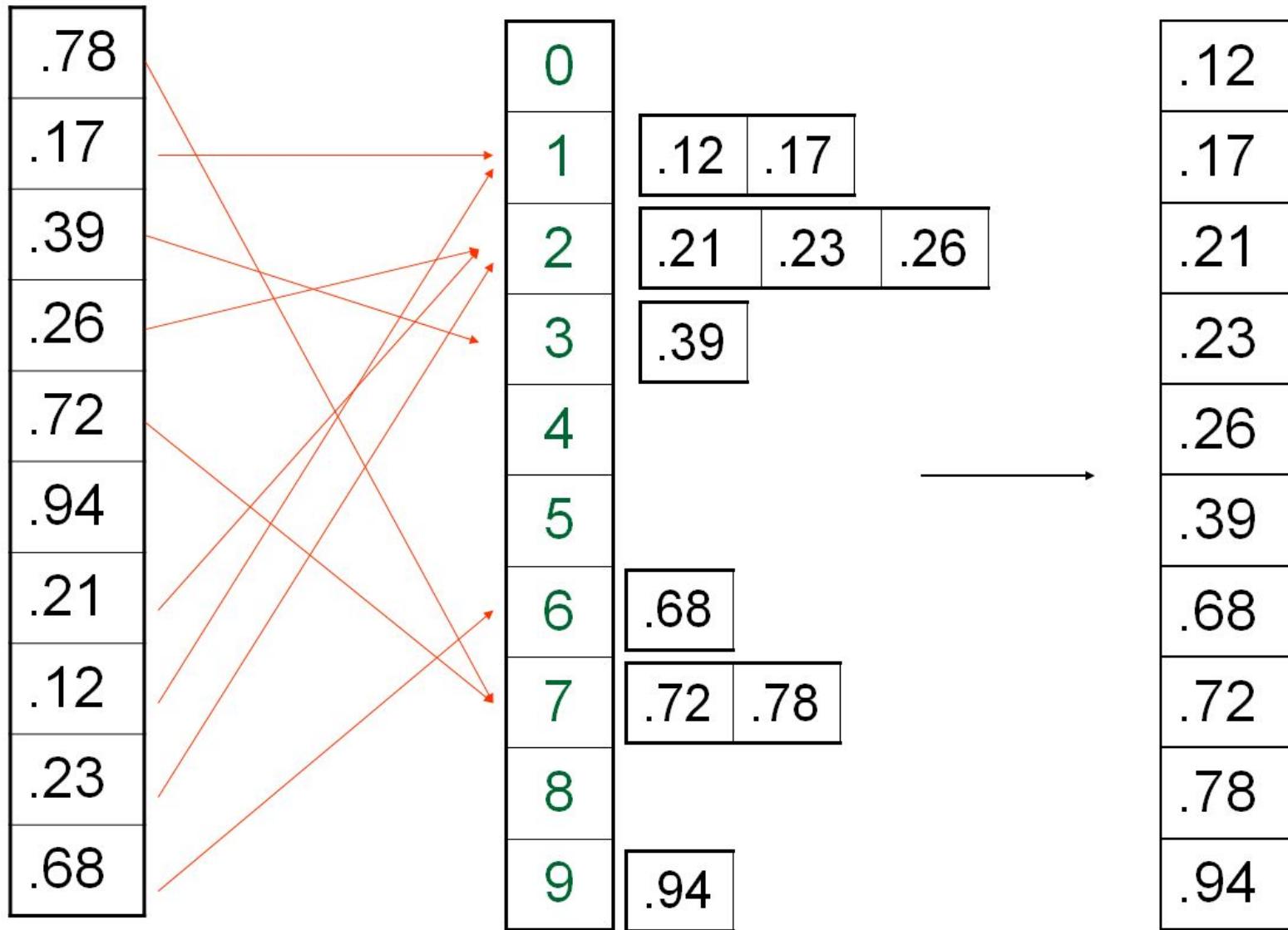
# Bucket Sort

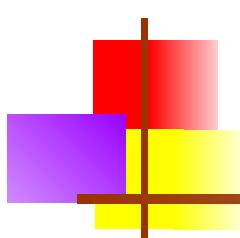
---

**Bucket-Sort(A)**

1.  $n \leftarrow \text{length}(A)$
2. **for**  $i \leftarrow 0$  to  $n$   $\leftarrow$  Distribute elements over buckets
3.   **do** insert  $A[i]$  into list  $B[\text{floor}(n^*A[i])]$
4. **for**  $i \leftarrow 0$  to  $n - 1$   $\leftarrow$  Sort each bucket
5.   **do** Insertion-Sort( $B[i]$ )
6. Concatenate lists  $B[0], B[1], \dots, B[n - 1]$  in order

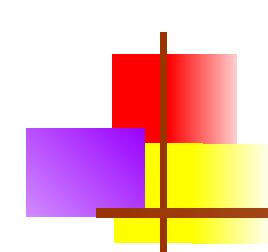
# Bucket Sort Example





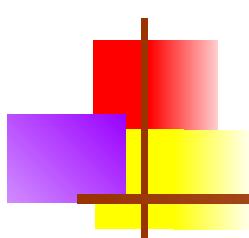
# Bucket Sort-Running Time

- ❑ All lines except line 5 (Insertion-Sort) take  $O(n)$  in the worst case.
- ❑ In the worst case,  $O(n)$  numbers will end up in the same bucket, so in the worst case, it will take  $O(n^2)$  time.
- ❑ Lemma: Given that the input sequence is drawn uniformly at random from  $[0,1]$ , the expected size of a bucket is  $O(1)$ .
- ❑ So, in the average case, only a constant number of elements will fall in each bucket, so it will take  $O(n)$  (see proof in book).
- ❑ Use a different indexing scheme (hashing) to distribute the numbers uniformly.



# Bucket Sort Review

- ❑ Assumption: input is uniformly distributed across a range
- ❑ Basic idea:
  - ❑ Partition the range into a fixed number of buckets.
  - ❑ Toss each element into its appropriate bucket.
  - ❑ Sort each bucket.
- ❑ Pro's:
  - ❑ Fast
  - ❑ Asymptotically fast (i.e.,  $O(n)$  when distribution is uniform)
  - ❑ Simple to code
  - ❑ Good for a rough sort.
- ❑ Con's:
  - ❑ Doesn't sort in place



# Summary of Linear Sorting

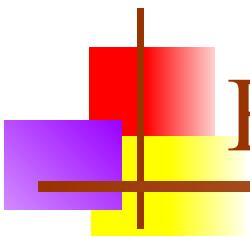
## Non-Comparison Based Sorts

|               | Running Time   |                |                |          |
|---------------|----------------|----------------|----------------|----------|
|               | worst-case     | average-case   | best-case      | in place |
| Counting Sort | $O(n + k)$     | $O(n + k)$     | $O(n + k)$     | no       |
| Radix Sort    | $O(d(n + k'))$ | $O(d(n + k'))$ | $O(d(n + k'))$ | no       |
| Bucket Sort   |                | $O(n)$         |                | no       |

Counting sort assumes input elements are in range  $[0, 1, 2, \dots, k]$  and uses array indexing to count the number of occurrences of each value.

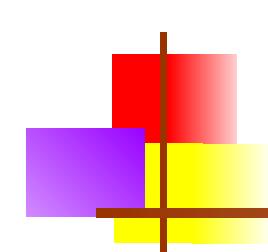
Radix sort assumes each integer consists of  $d$  digits, and each digit is in range  $[1, 2, \dots, k']$ .

Bucket sort requires advance knowledge of input distribution (sorts  $n$  numbers uniformly distributed in range in  $O(n)$  time).



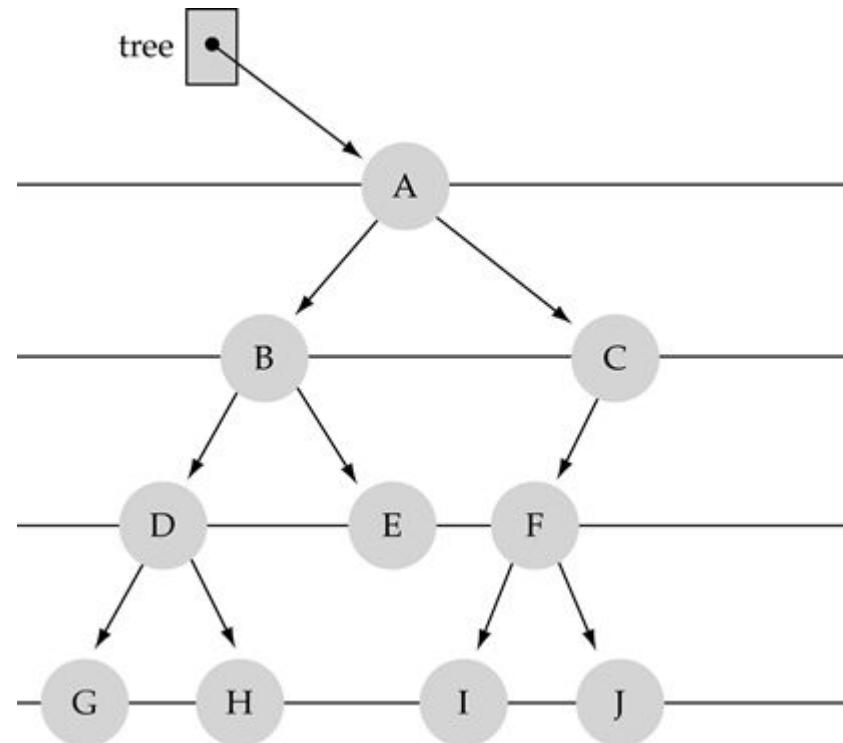
# Binary Search Tree and AVL Tree

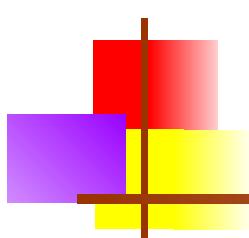




# What is a binary tree?

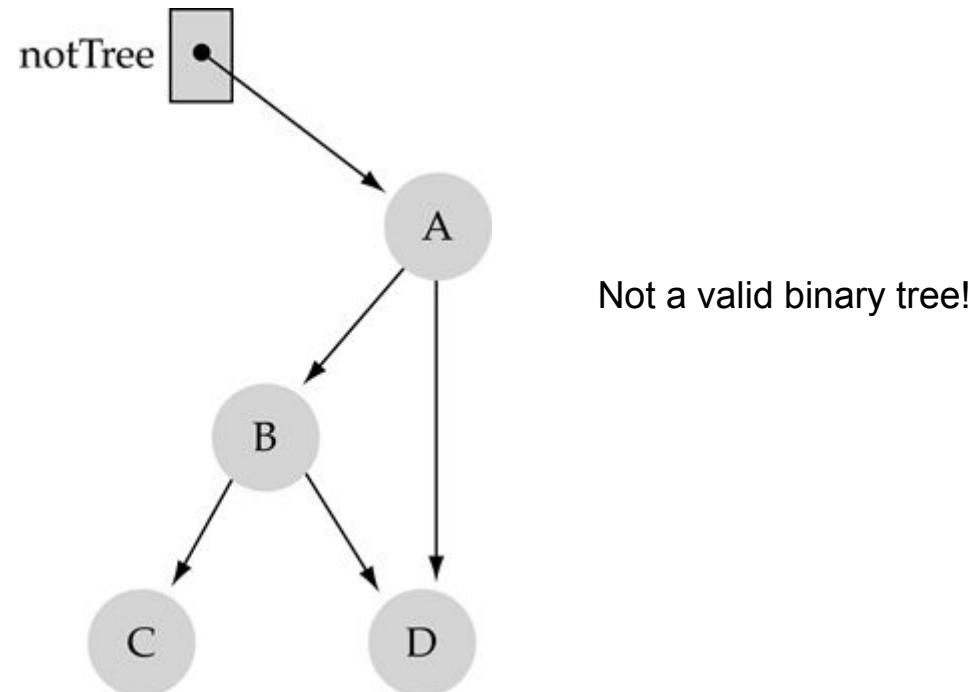
Property 1: each node can have up to two successor nodes.





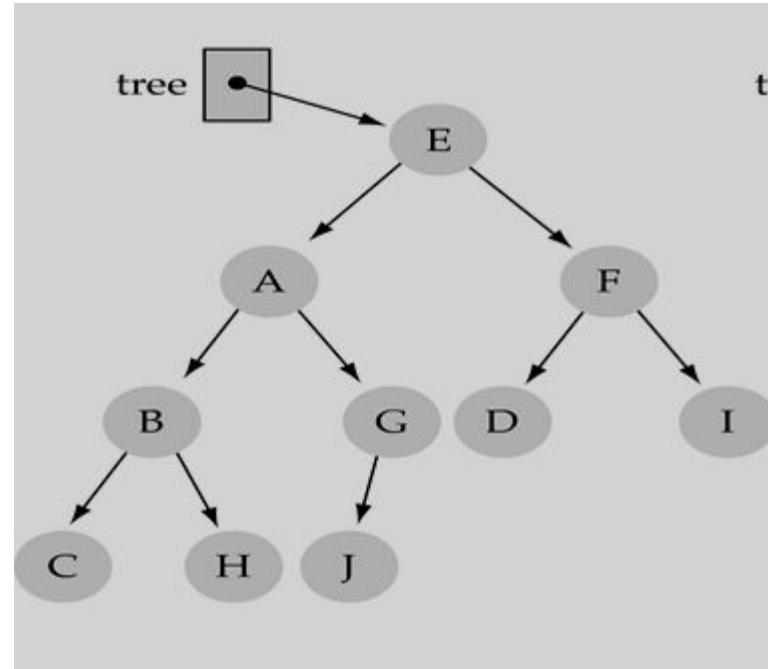
# What is a binary tree? (cont.)

Property 2: a unique path exists from the root to every other node



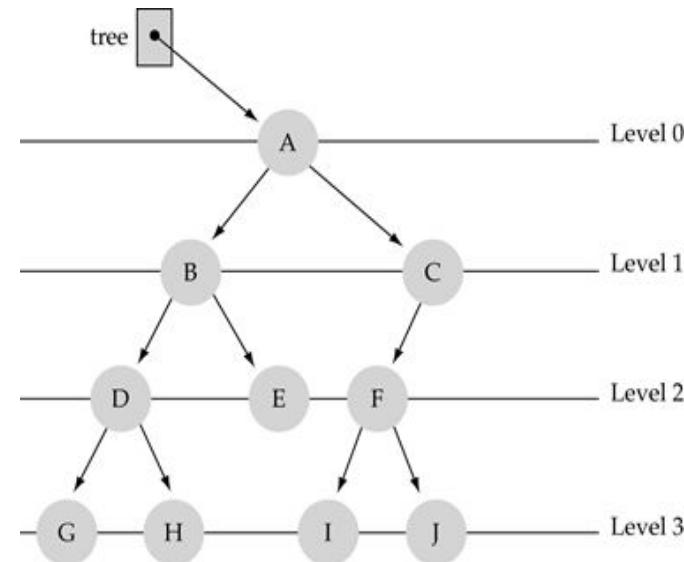
# Some terminology

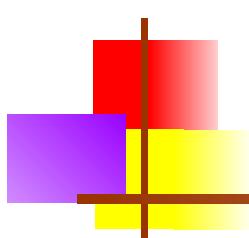
- ❑ The successor nodes of a node are called its **children**
- ❑ The predecessor node of a node is called its **parent**
- ❑ The "beginning" node is called the **root** (has no parent)
- ❑ A node without children is called a **leaf**



# Some terminology (cont'd)

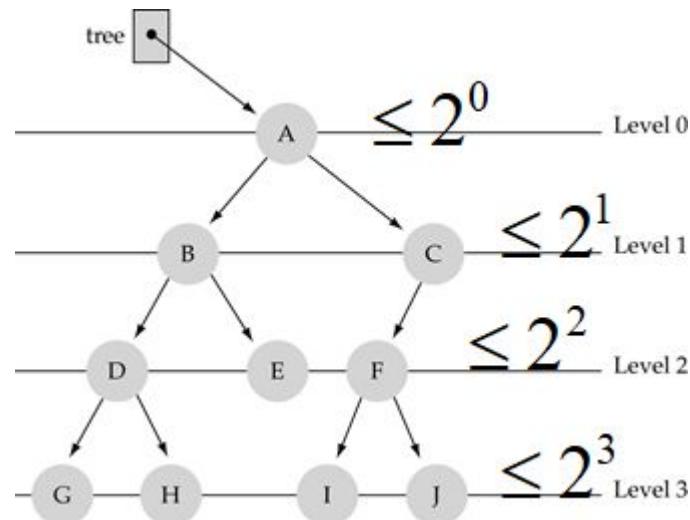
- ❑ Nodes are organized in levels (indexed from 0).
- ❑ Level (or depth) of a node: number of edges in the path from the root to that node.
- ❑ Height of a tree  $h$ : #levels =  $L$   
(Warning: some books define  $h$  as #levels-1).
- ❑ Full tree: every node has exactly
  - ❑ two children and all the
  - ❑ leaves are on the same level.

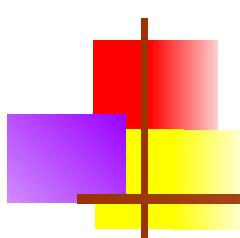




# What is the max #nodes at some level l?

The max #nodes at level l is  $2^l$  where l=0,1,2, L-1





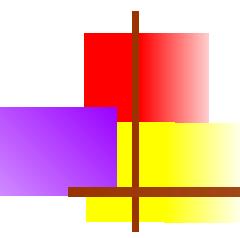
# What is the total #nodes N of a full tree with height h?

$$N = 2^0 + 2^1 + \dots + 2^{h-1} = 2^h - 1$$

| = 0      | = 1      | = h-1

using the geometric series:

$$x^0 + x^1 + \dots + x^{n-1} = \sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

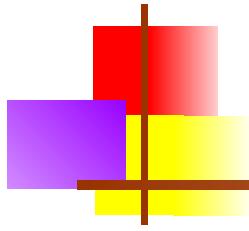


# What is the height $h$ of a full tree with $N$ nodes?

$$2^h - 1 = N$$

$$\Rightarrow 2^h = N + 1$$

$$\Rightarrow h = \log(N + 1) \rightarrow O(\log N)$$

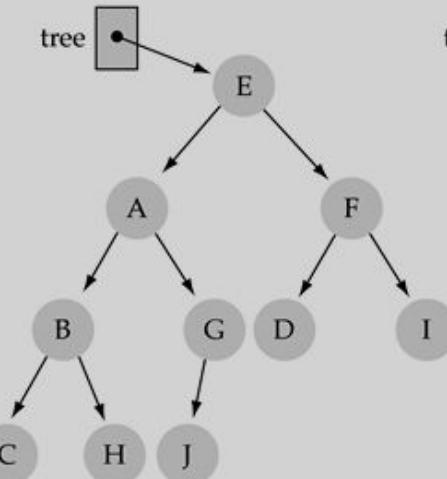


# Why is h important?

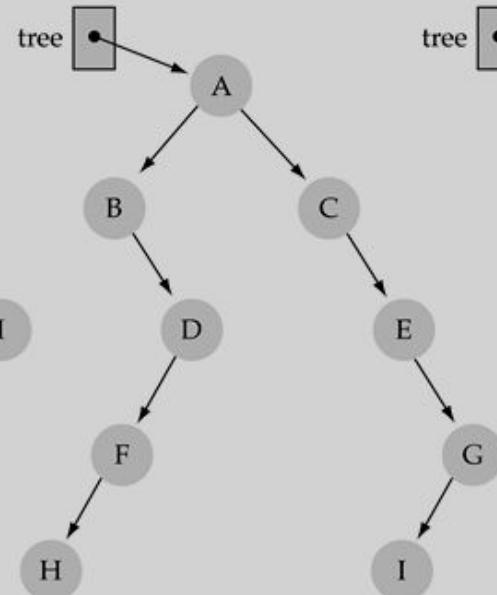
---

- Tree operations (e.g., insert, delete, retrieve etc.) are typically expressed in terms of  $h$ .
  
- So,  $h$  determines running time!

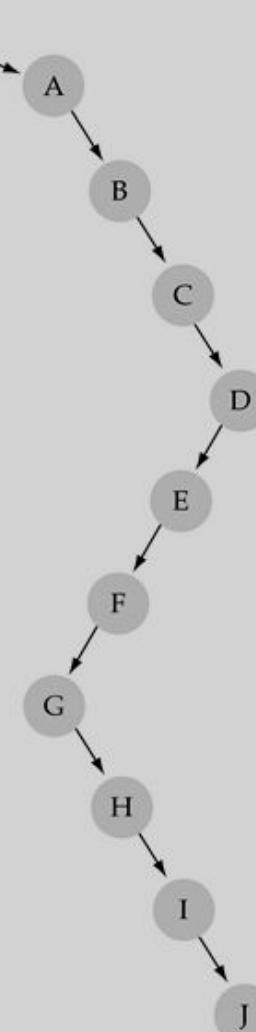
(a) A 4-level tree

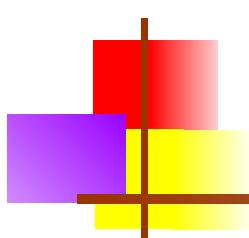


(b) A 5-level tree



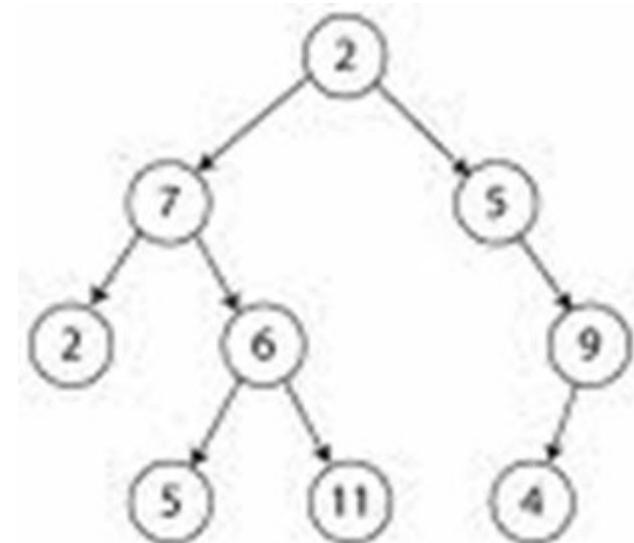
(c) A 10-level tree





# How to search a binary tree?

- (1) Start at the root
- (2) Search the tree level by level, until you find the element you are searching for or you reach a leaf.



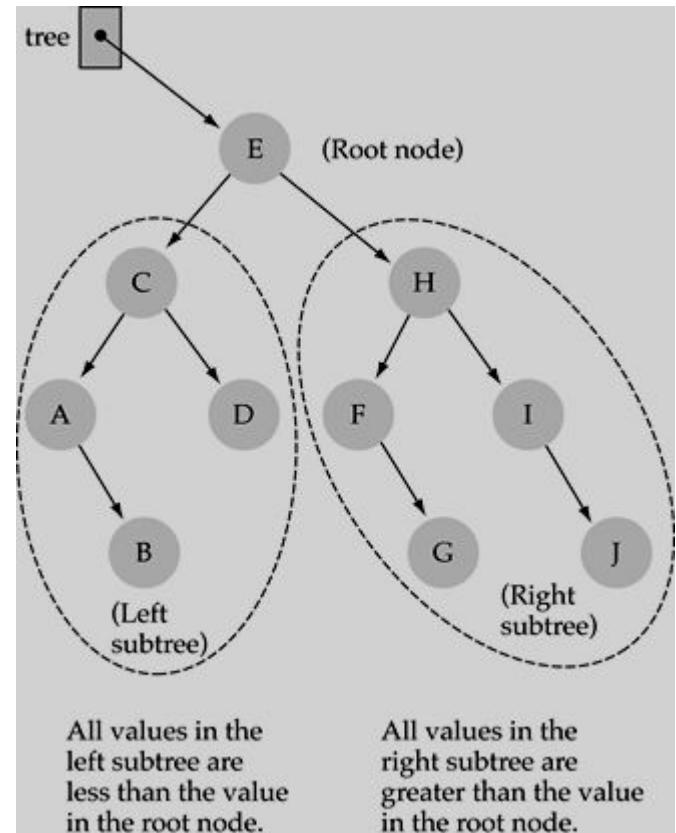
Is this better than searching a linked list?

No → O(N)

# Binary Search Trees (BSTs)

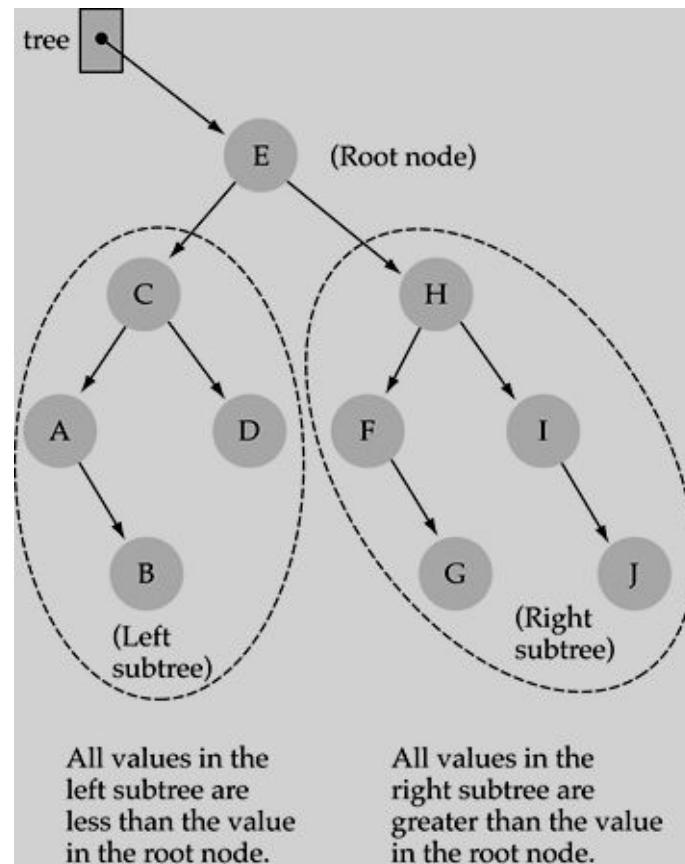
- Binary Search Tree Property:

The value stored at a node is greater than the value stored at its left child and less than the value stored at its right child



# Binary Search Trees (BSTs)

- In a BST, the value stored at the root of a subtree is **greater** than any value in its **left subtree** and less than any value in its **right subtree**!



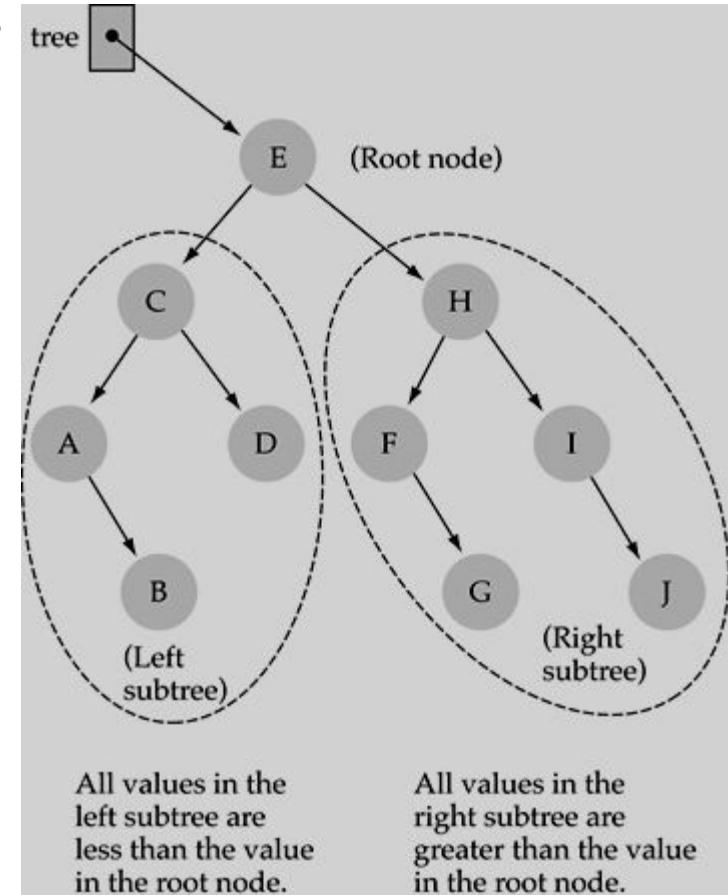
# Binary Search Trees (BSTs)

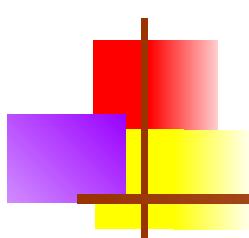
- Where is the **smallest** element?

Ans: **leftmost element**

- Where is the **largest** element?

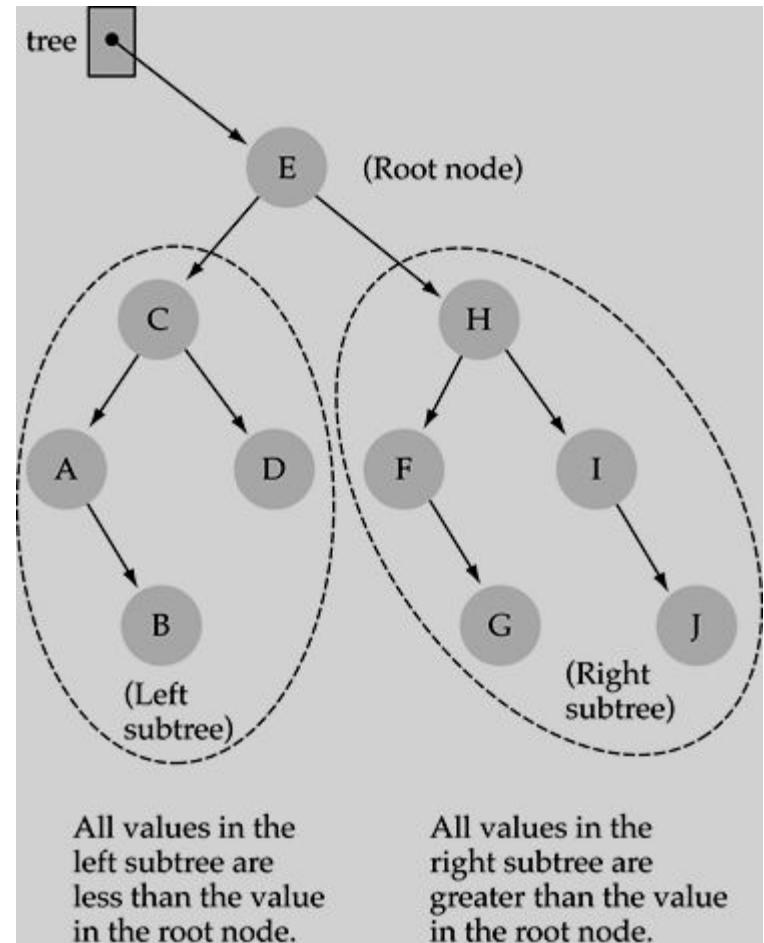
Ans: **rightmost element**

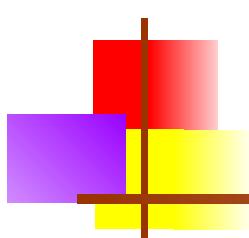




# How to search a binary search tree?

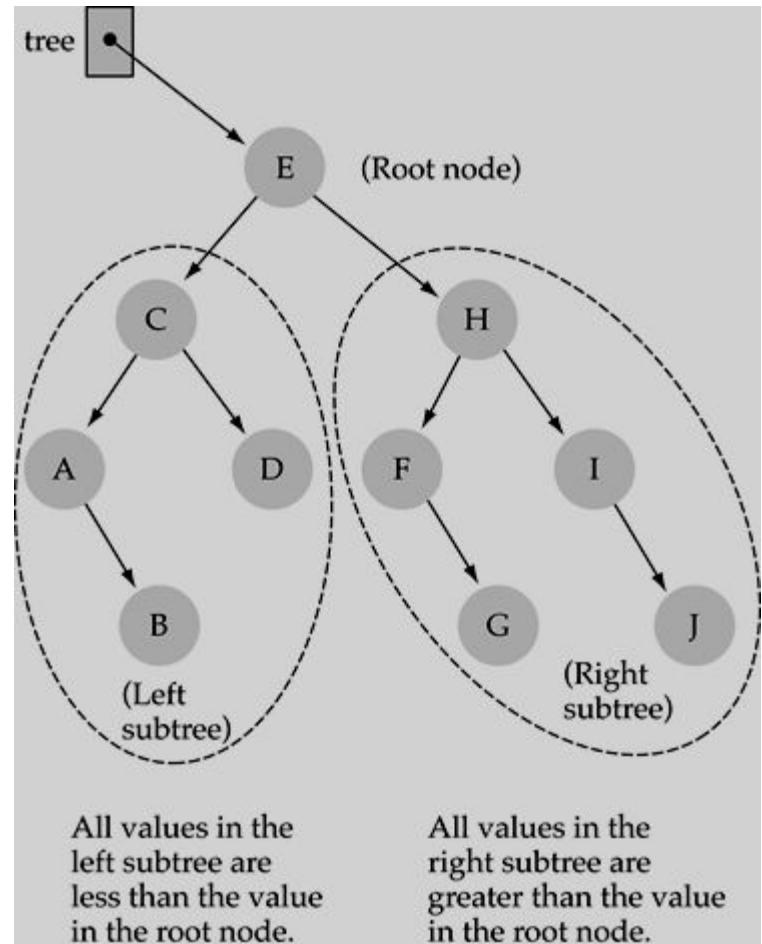
- (1) Start at the root
- (2) Compare the value of the item you are searching for with the value stored at the root
- (3) If the values are equal, then item found; otherwise, if it is a leaf node, then not found

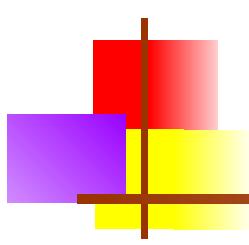




# How to search a binary search tree?

- (4) If it is less than the value stored at the root, then search the left subtree
- (5) If it is greater than the value stored at the root, then search the right subtree
- (6) Repeat steps 2-6 for the root of the subtree chosen in the previous step 4 or 5

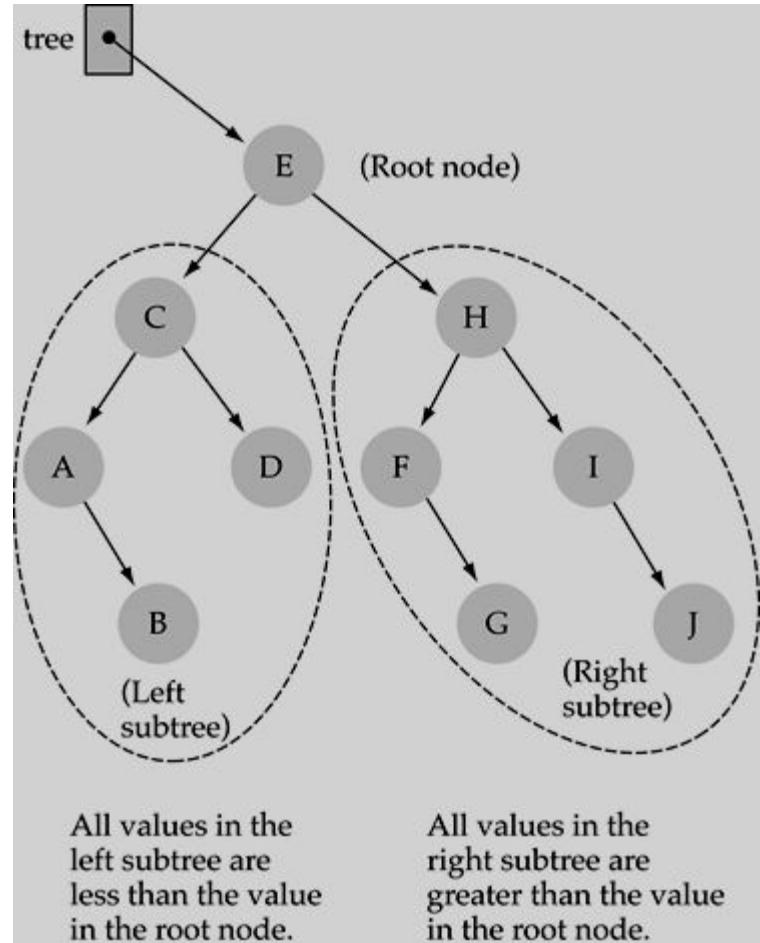




# How to search a binary search tree?

How to search a binary search tree?

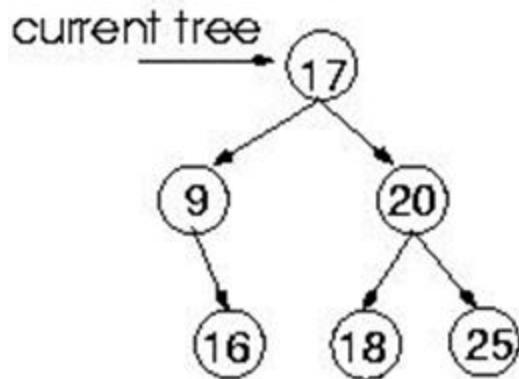
Yes !! --->  $O(\log N)$



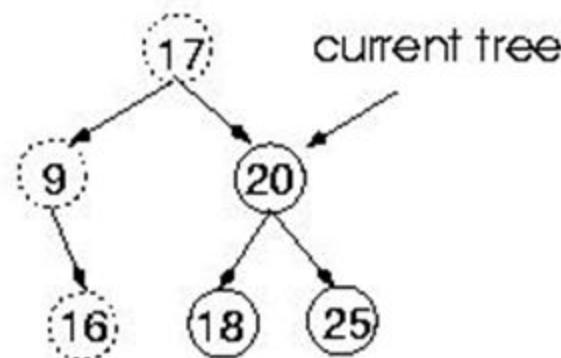
# Function Retrieve Item

**Retrieve: 18**

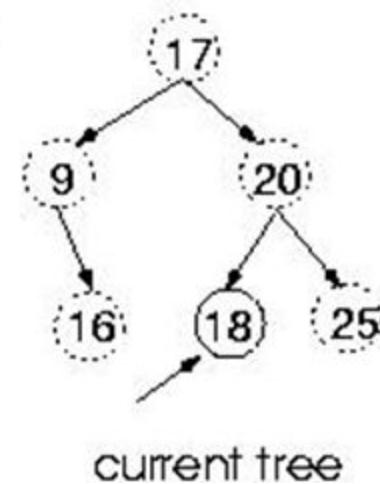
**Compare 18 with 17:**  
**Choose right subtree**



**Compare 18 with 20:**  
**Choose left subtree**



**Compare 18 with 18:**  
**Found !!**

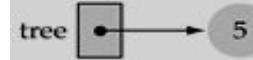


# Function Insert Item

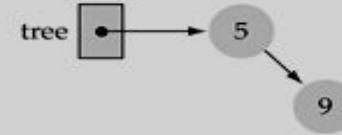
Use the  
binary  
search tree  
property to  
insert the  
new item at  
the correct  
place

(a) tree 

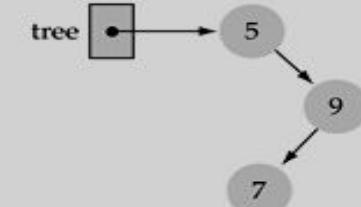
(b) Insert 5



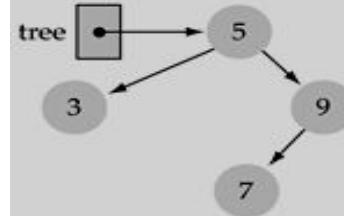
(c) Insert 9



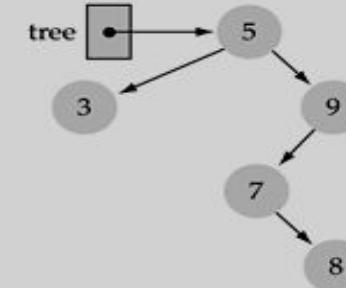
(d) Insert 7



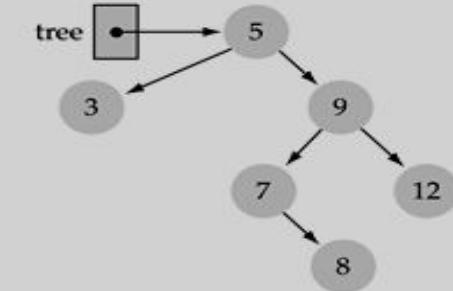
(e) Insert 3



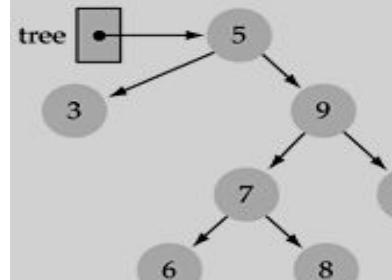
(f) Insert 8



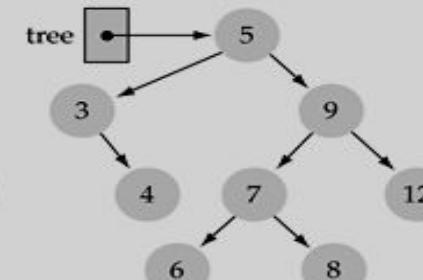
(g) Insert 12



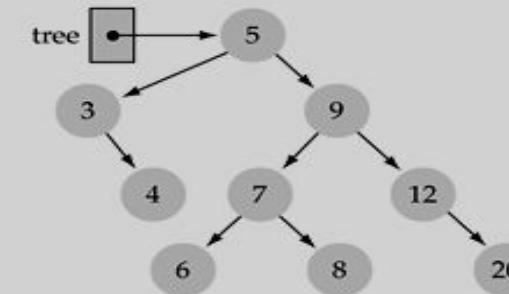
(h) Insert 6



(i) Insert 4

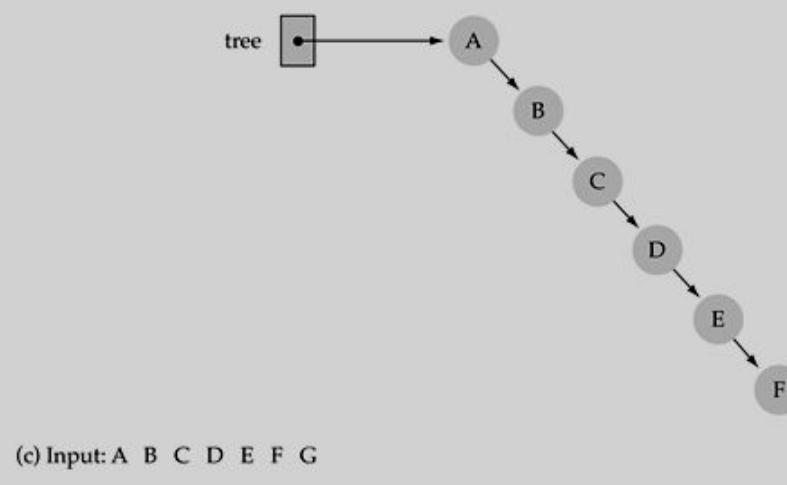
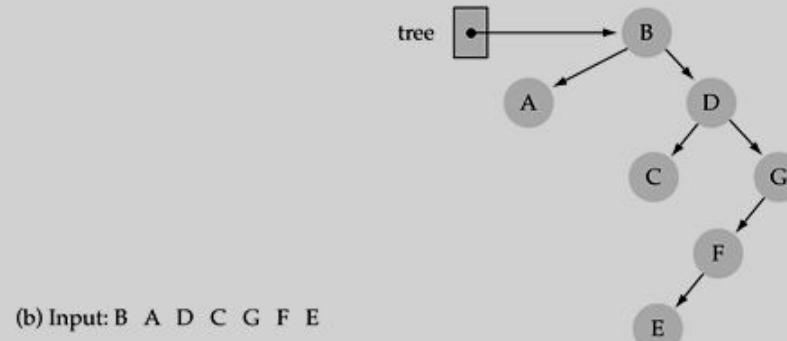
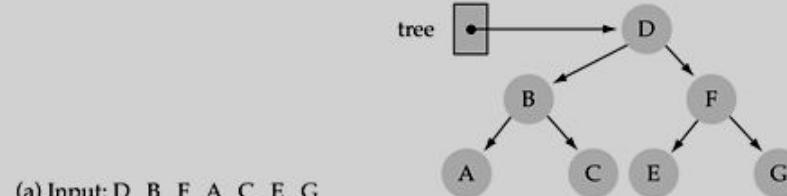


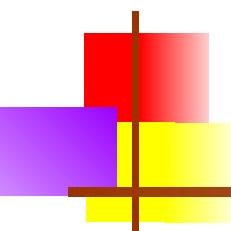
(j) Insert 20



# Does the order of inserting elements into a tree matter?

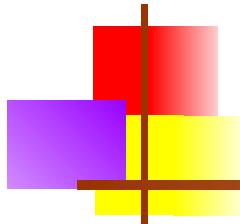
Yes,  
certain  
orders  
might  
produce  
very  
unbalance  
ed trees!





# Does the order of inserting elements into a tree matter? (cont'd)

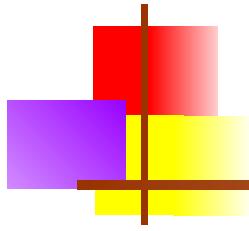
- ❑ Unbalanced trees are not desirable because search time increases!
- ❑ Advanced tree structures, such as red-black trees, guarantee balanced trees.



# Function Delete Item

---

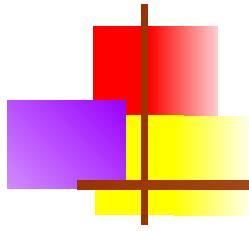
- ❑ First, find the item; then, delete it
- ❑ Binary search tree property must be preserved!!
- ❑ We need to consider three different cases:
  - (1) Deleting a leaf
  - (2) Deleting a node with only one child
  - (3) Deleting a node with two children



# Deleting leaf node

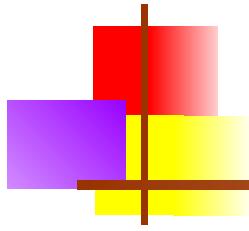
---

- ❑ Find the node in the given BST
- ❑ Simply delete that Node.



# Deleting a node with only one child

- ❑ Find the node in the BST
- ❑ Delete this node by connecting grandfather and grandchildren



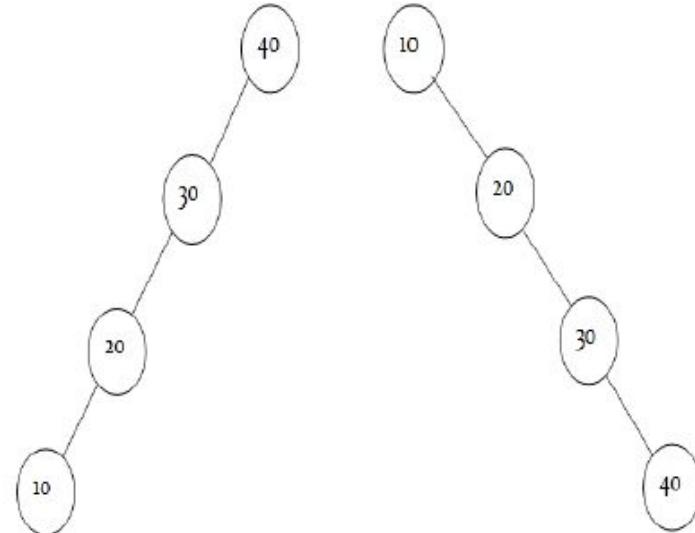
# If the node contains 2-children

---

- ❑ Find the node in the given BST.
- ❑ Delete that by replacing inorder successor or predecessor.

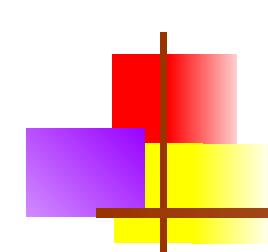
# Skewed BST

- If a tree which is dominated by left child node or right child node, is said to be a Skewed Binary Tree.
- In a left skewed tree, most of the nodes have the left child without corresponding right child.
- In a right skewed tree, most of the nodes have the right child without corresponding left child.



Left Skewed Tree

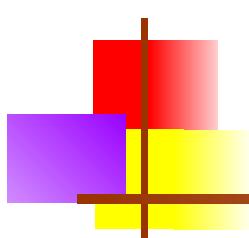
Right Skewed Tree



# Limitation of BST

---

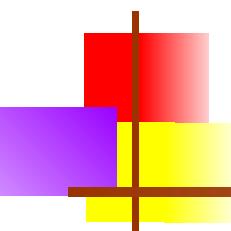
- ❑ The average search time for a binary search tree is directly proportional to its height:  $O(h)$ . Most of the operation average case time is  $O(\log 2n)$ .
- ❑ BST's are not guaranteed to be balanced. It may be skewed tree also.
- ❑ For skewed BST, the average search time becomes  $O(n)$ . So, it is working like an linear array.
- ❑ To improve average search time and make BST balanced, AVL trees are used.



# AVL Tree

---

- ❑ AVL tree is a height balanced tree.
- ❑ It is a self-balancing binary search tree.
- ❑ It was invented by Adelson-Velskii and Landis.
- ❑ AVL trees have a faster retrieval.
- ❑ It takes  $O(\log n)$  time for insertion and deletion operation.
- ❑ In AVL tree, difference between heights of left and right subtree cannot be more than one for all nodes.



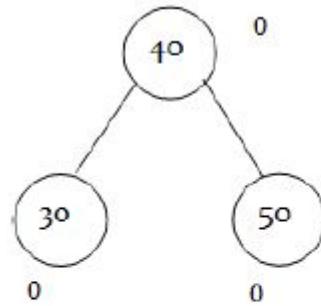
# AVL Tree

---

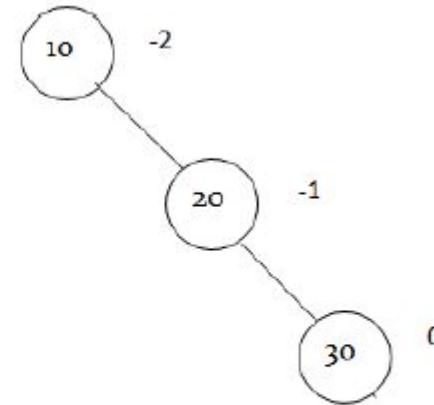
- ❑ Balance Factor of node is:
  - ❑ Height of left subtree – Height of Right subtree
- ❑ Balance Factor is calculated for every node of AVL tree.
- ❑ At every node, height of left and right subtree can differ by no more than 1.
- ❑ For AVL tree, the possible values of balance factor are -1, 0, 1
- ❑ Balance Factor of leaf nodes is 0 (zero).

# Example

Every AVL Tree is a binary search tree but all the Binary Search Tree need not to be AVL trees.

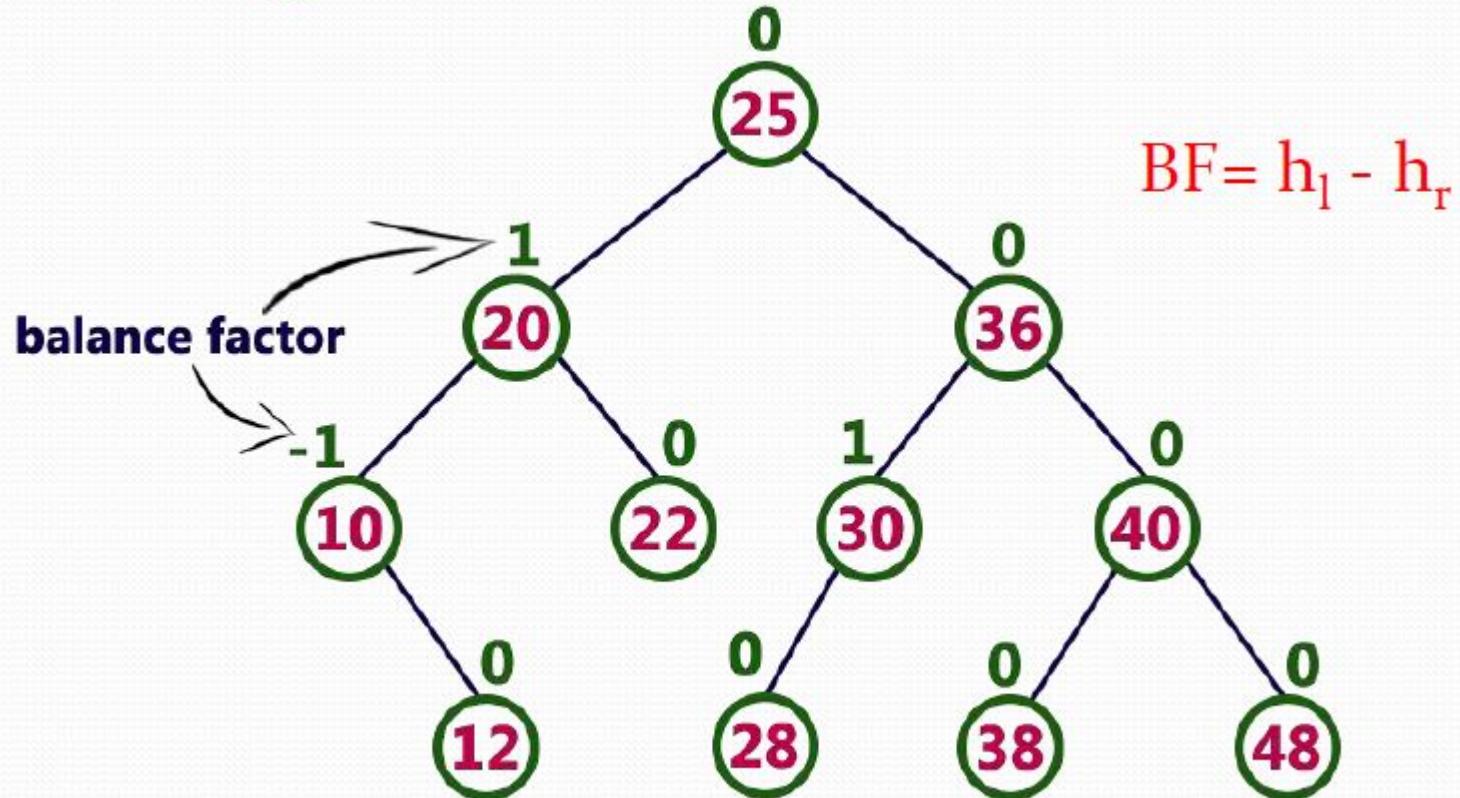


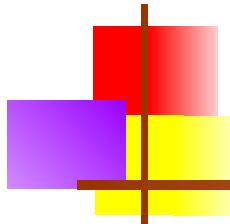
BST and AVL



BST but not AVL

# Finding Balance Factor

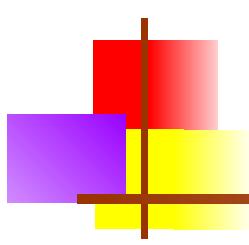




# Height of AVL Tree

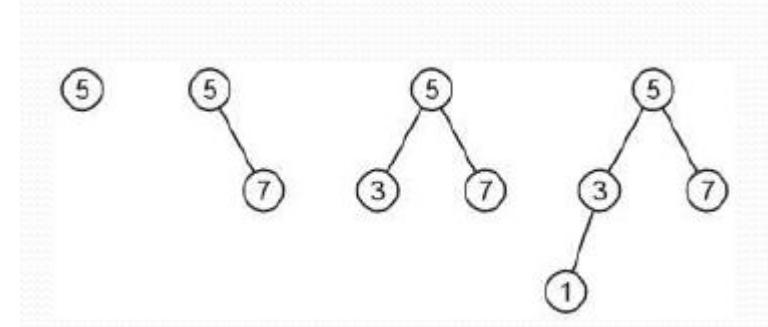
---

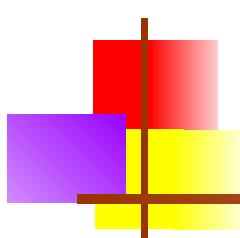
- ❑ By the definition of complete trees, any complete binary search tree is an AVL tree
- ❑ Thus, an upper bound on the number of nodes in an AVL tree of height  $h$  is a perfect binary tree with  $2^{h+1} - 1$  nodes.
- ❑ What is a lower bound?



# Height of AVL Tree

- ❑ Let  $F(h)$  be the fewest number of nodes in a tree of height  $h$ .
- ❑ From a previous slide:
  - ❑  $F(0) = 1$
  - ❑  $F(1) = 2$
  - ❑  $F(2) = 4$
- ❑ Then what is  $F(h)$  in general?





# Height of AVL Tree

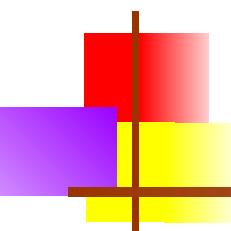
The worst-case AVL tree of height  $h$  would have:

- A worst-case AVL tree of height  $h - 1$  on one side,
- A worst-case AVL tree of height  $h - 2$  on the other, and
- The **root** node

$$\text{We get: } F(h) = F(h - 1) + F(h - 2) + 1$$

This is a recurrence relation:

$$F(h) = \begin{cases} 1 & h = 0 \\ 2 & h = 1 \\ F(h - 1) + F(h - 2) + 1 & h > 1 \end{cases}$$

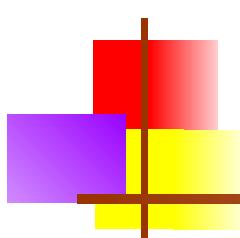


# Imbalance

---

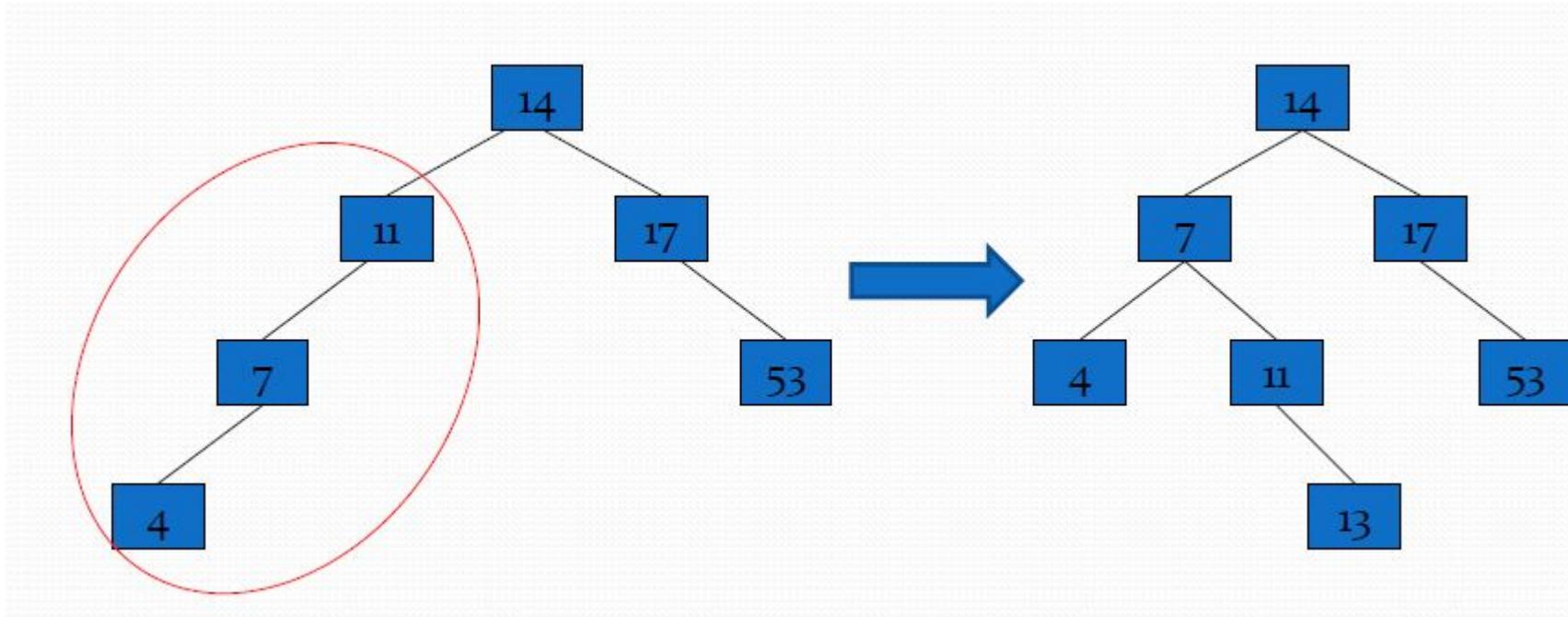
After an insertion, when the balance factor of node A is  $-2$  or  $2$ , the node A is one of the following four imbalance types

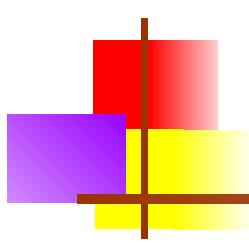
- 1. LL: new node is in the left subtree of the left subtree of A
- 1. LR: new node is in the right subtree of the left subtree of A
- 1. RR: new node is in the right subtree of the right subtree of A
- 1. RL: new node is in the left subtree of the right subtree of A



# AVL Tree Example

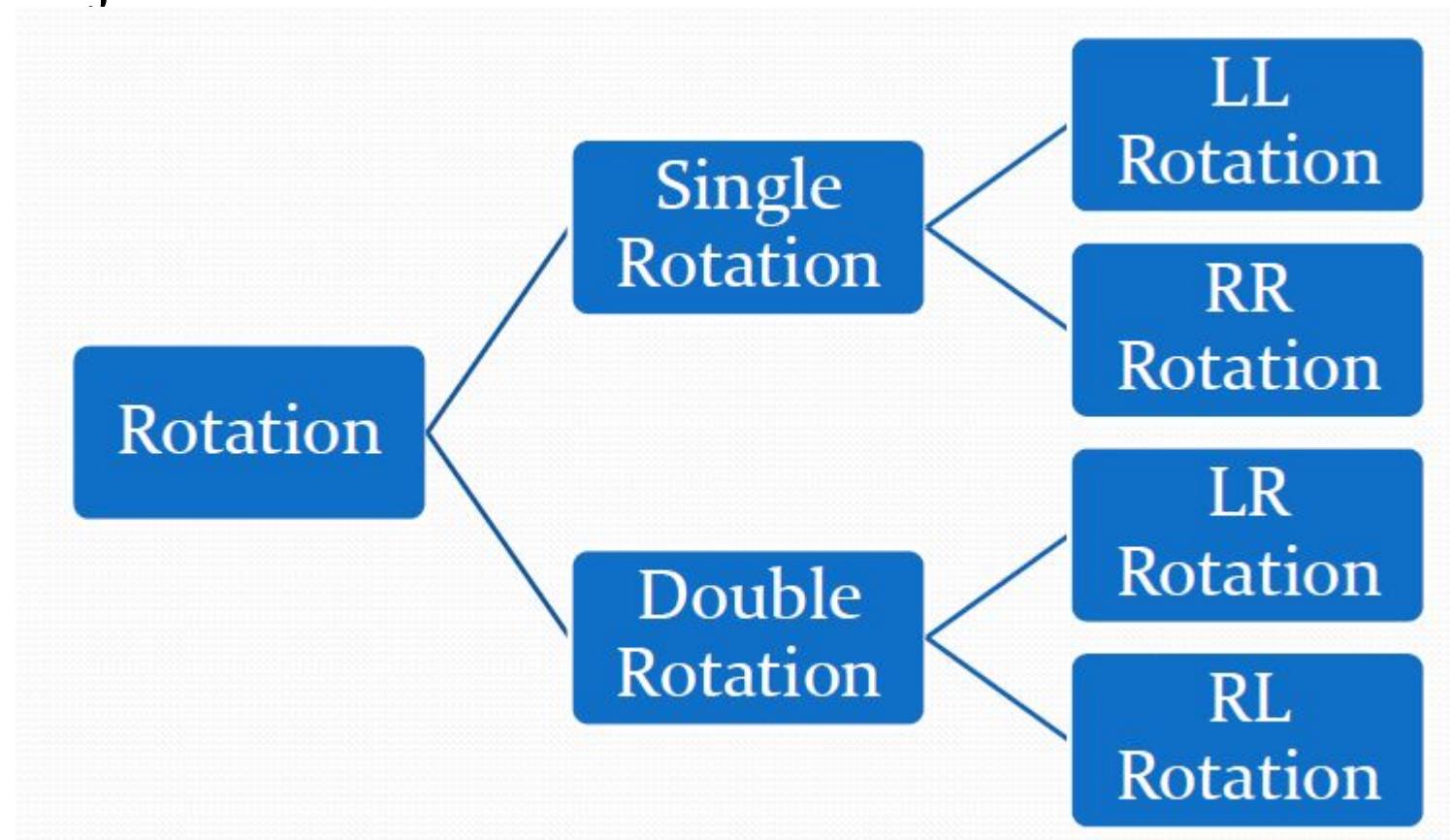
- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree

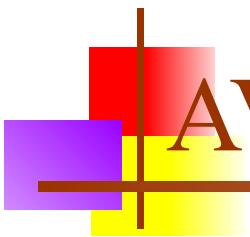




# Types of Rotation

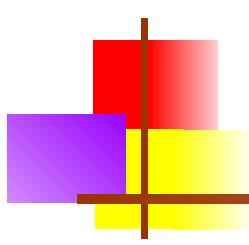
Rotation- To switch children and parents among two or three adjacent nodes to restore balance of a tree.





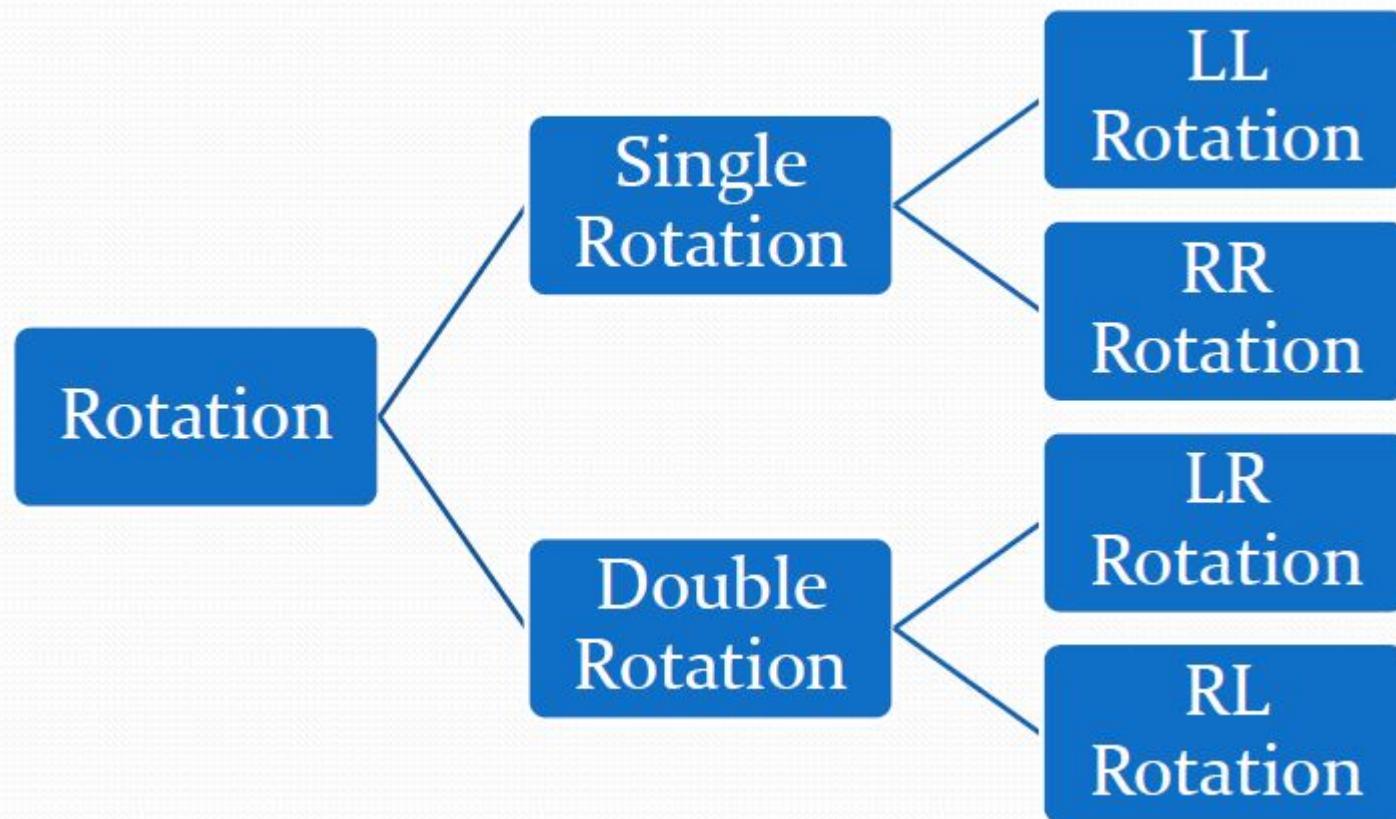
# AVL-Tree

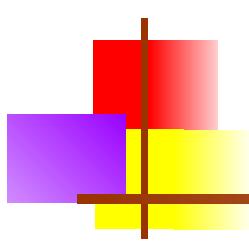




# Types of Rotation

Rotation- To switch children and parents among two or three adjacent nodes to restore balance of a tree.



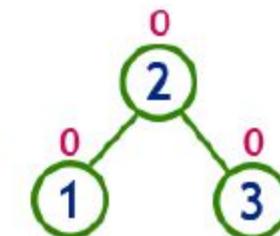
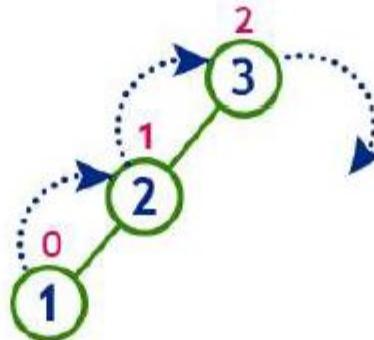
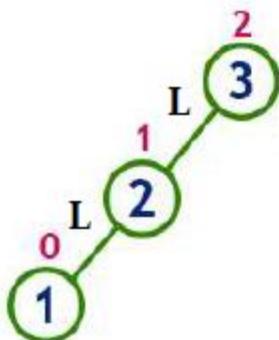


# Types of Rotation

- ❑ Single Rotation is applied when imbalanced node and child has same sign of BF (in the direction of new inserted node).
  - ❑ LL Rotation is applied in case of +ve sign. It mean left tree is heavy and so LL rotation is done.
  - ❑ RR Rotation is applied in case of -ve sign. It mean right tree is heavy and so RR rotation is done.
- ❑ Double Rotation is applied when imbalanced node and child has different signs of BF (in the direction of new inserted node).

# LL Rotation

Insert 3,2,1 in AVL Tree



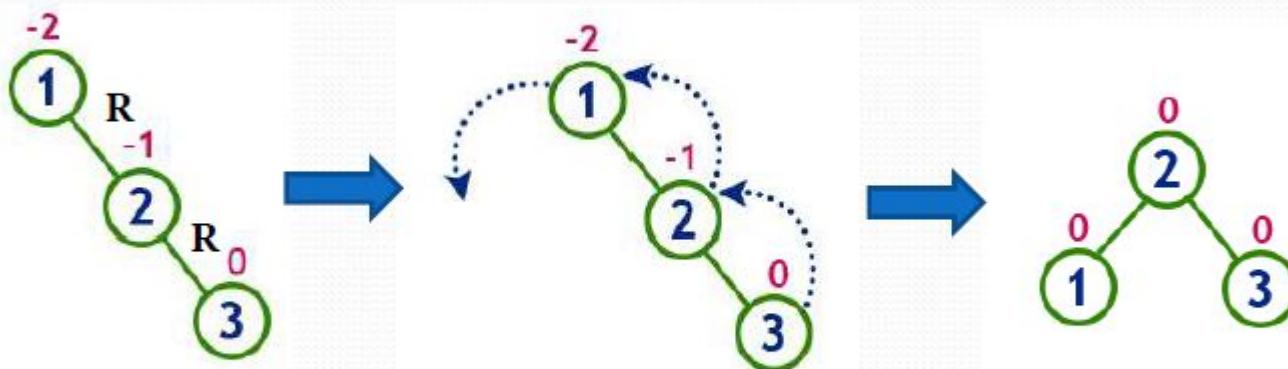
Imbalanced AVL Tree

LL Rotation

Balanced AVL Tree

# RR Rotation

Insert 1,2,3 in AVL Tree



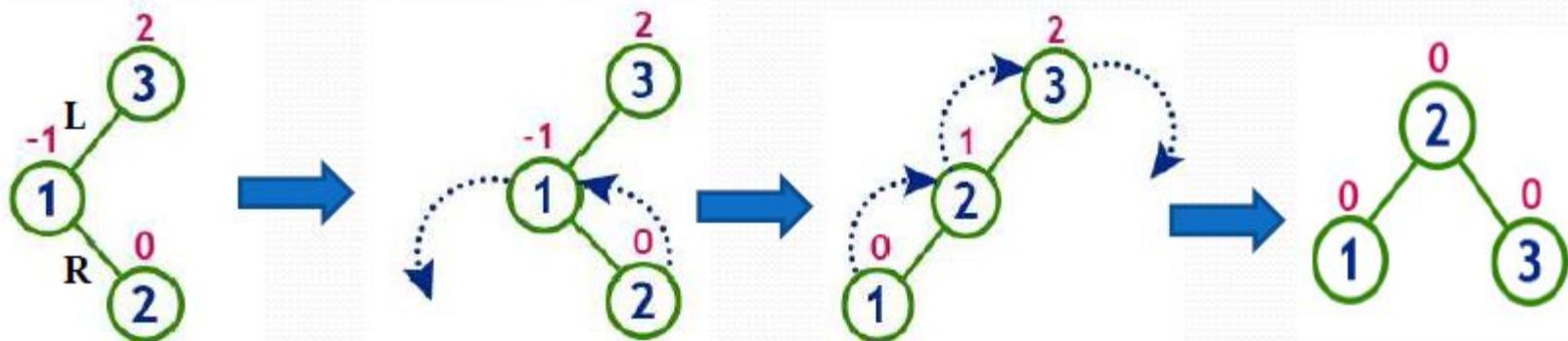
Imbalanced AVL Tree

RR Rotation

Balanced AVL Tree

# LR Rotation

Insert 3, 1, 2 in AVL Tree



Imbalanced AVL Tree

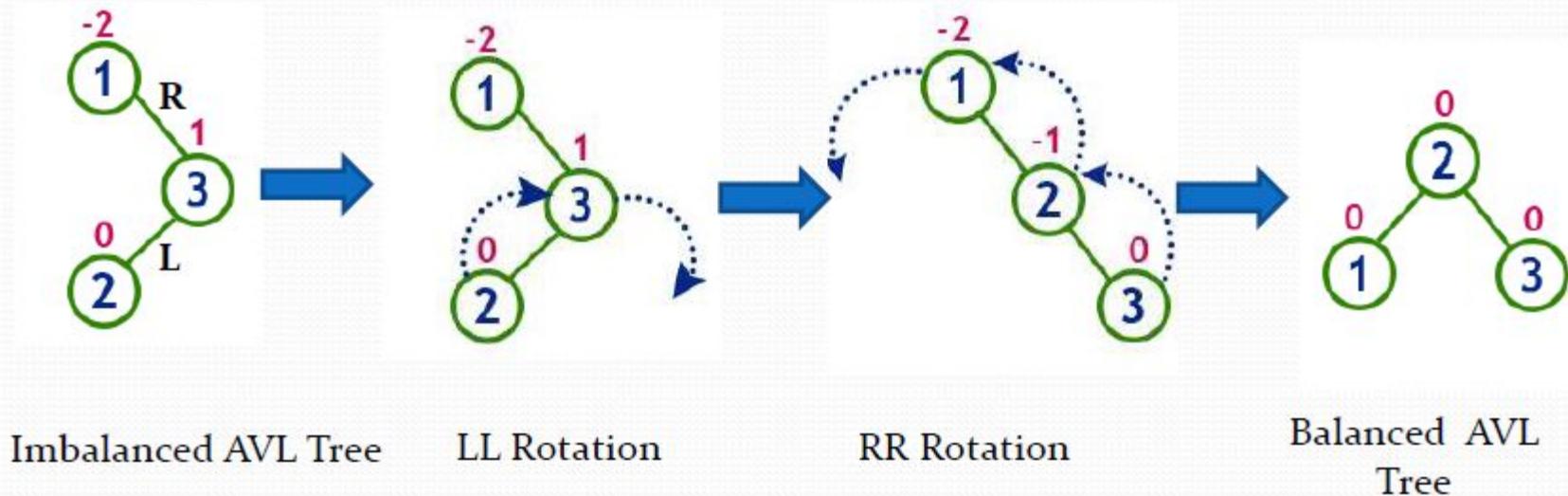
RR Rotation

LL Rotation

Balanced AVL  
Tree

# RL Rotation

Insert 1, 3, 2 in AVL Tree



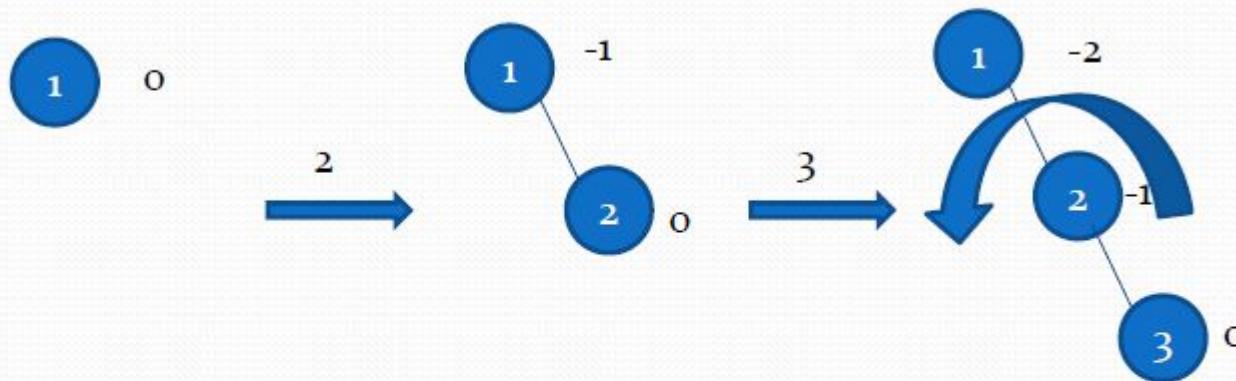
Imbalanced AVL Tree

LL Rotation

RR Rotation

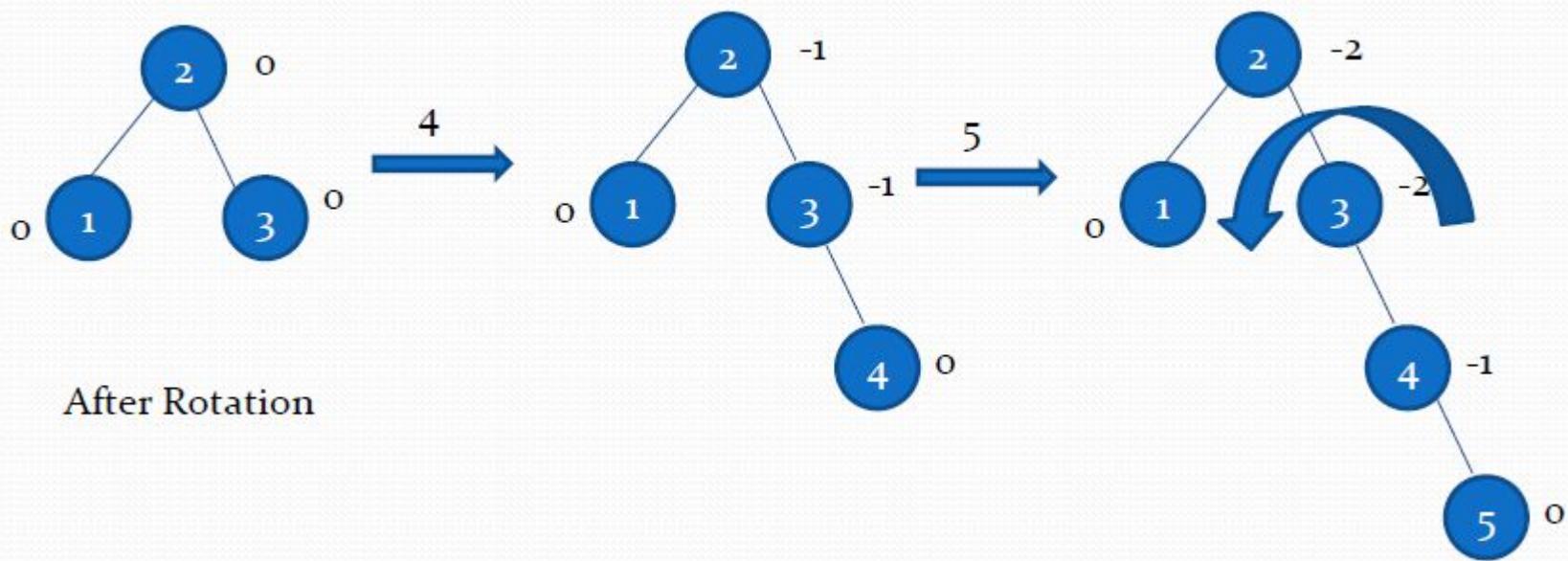
Balanced AVL  
Tree

# Construct a AVL Tree by inserting from 1 to 5 numbers

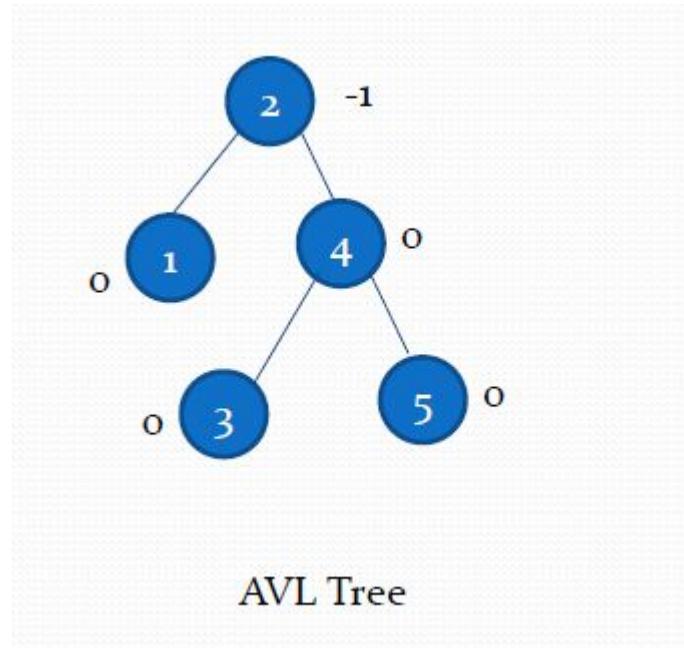


Not AVL  
Apply RR Rotation

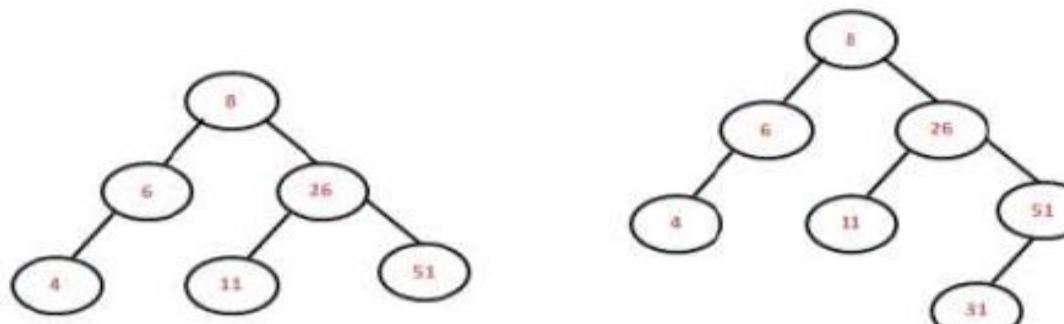
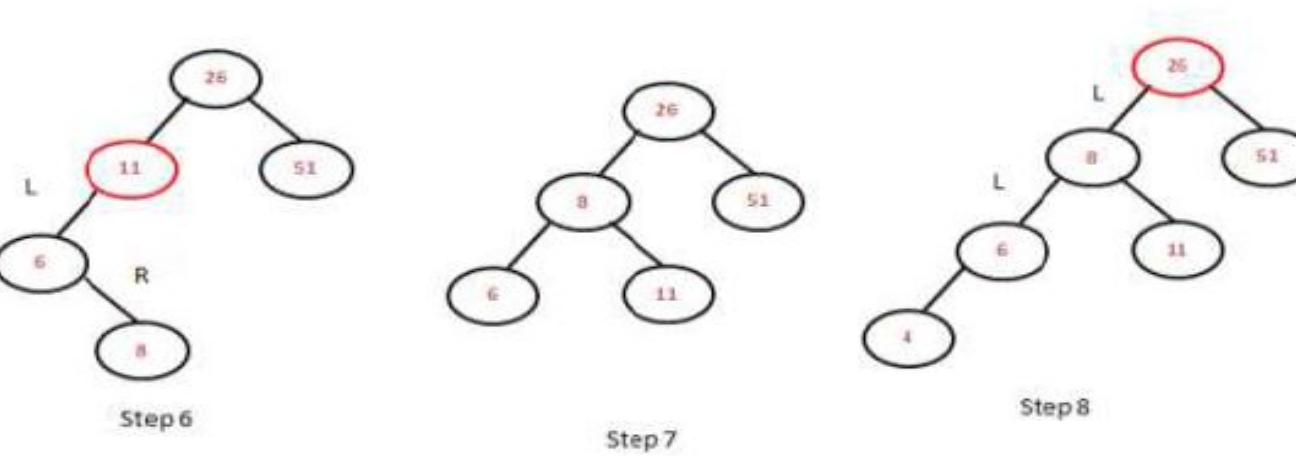
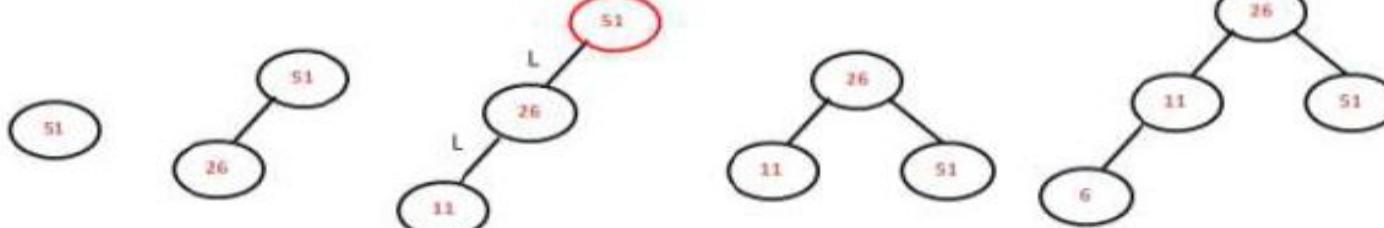
# Construct a AVL Tree by inserting from 1 to 5 numbers



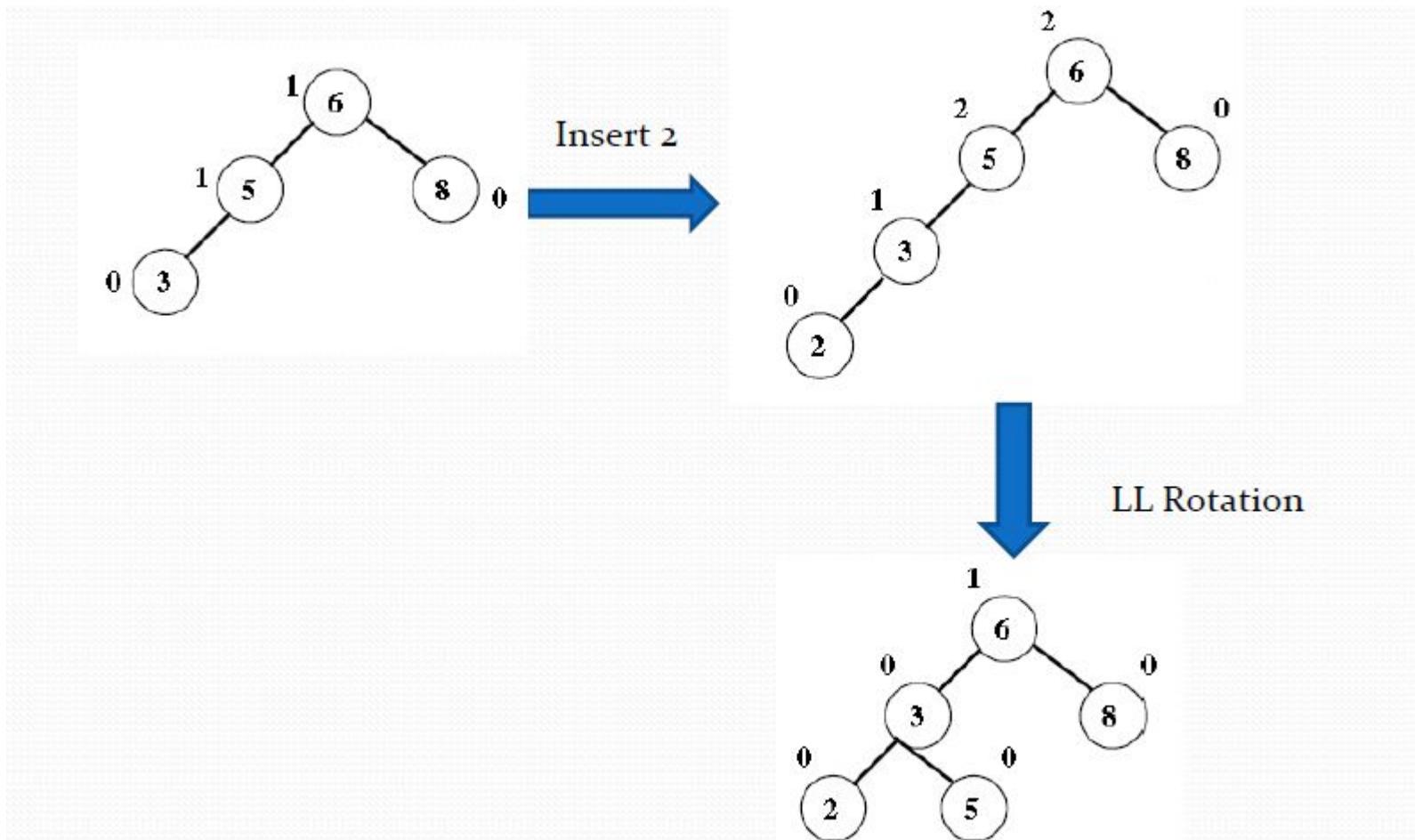
# Construct a AVL Tree by inserting from 1 to 5 numbers



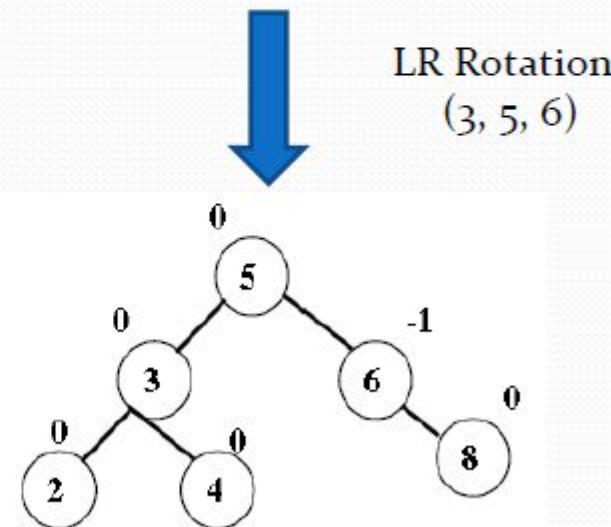
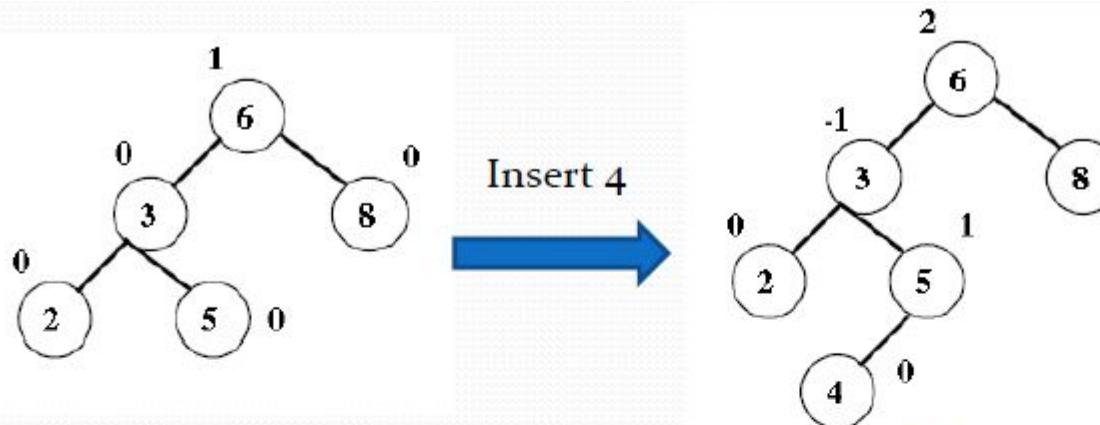
# Construct AVL Tree with data items: 51, 26, 11, 6, 8, 4, 31



# Insertion in AVL Tree

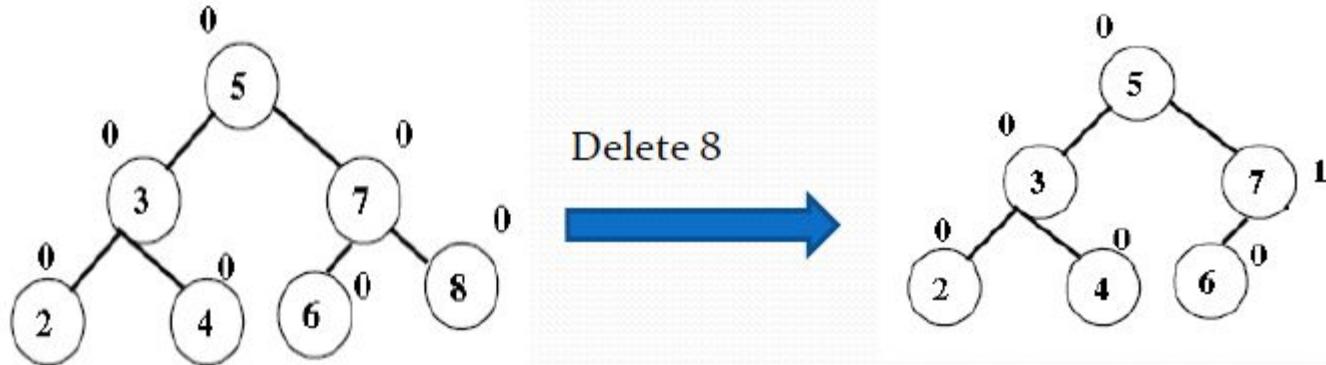


# Insertion in AVL Tree



# Deletion in AVL Tree

It is also possible to delete an item from AVL Tree.

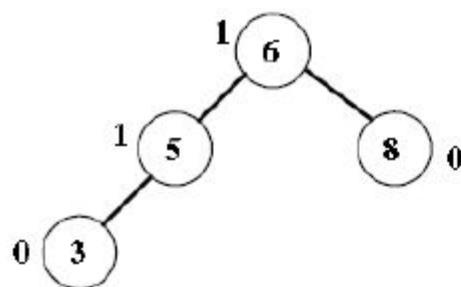


Balanced AVL

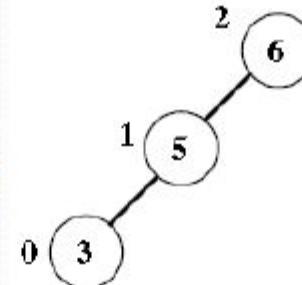
Balanced AVL

# Deletion in AVL Tree

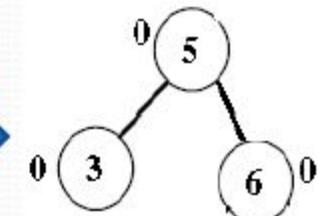
Just like insertion, deletion can cause an imbalance, which will need to be fixed by applying one of the four rotations.



Delete 8

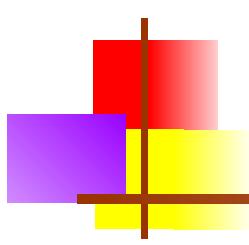


R1 Rotation



Balanced AVL

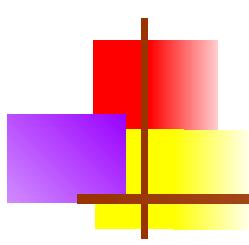
Imbalanced AVL



# Deletion in AVL Tree

---

- ❑ The deletion is also the same as in BST. However, in imbalanced tree due to deletion, one or more rotations need to be applied to balance the AVL tree.
- ❑ The Right(R) imbalance is classified into R0, R1, R-1
- ❑ The Left(L) imbalance is classified into L0, L1, L-1

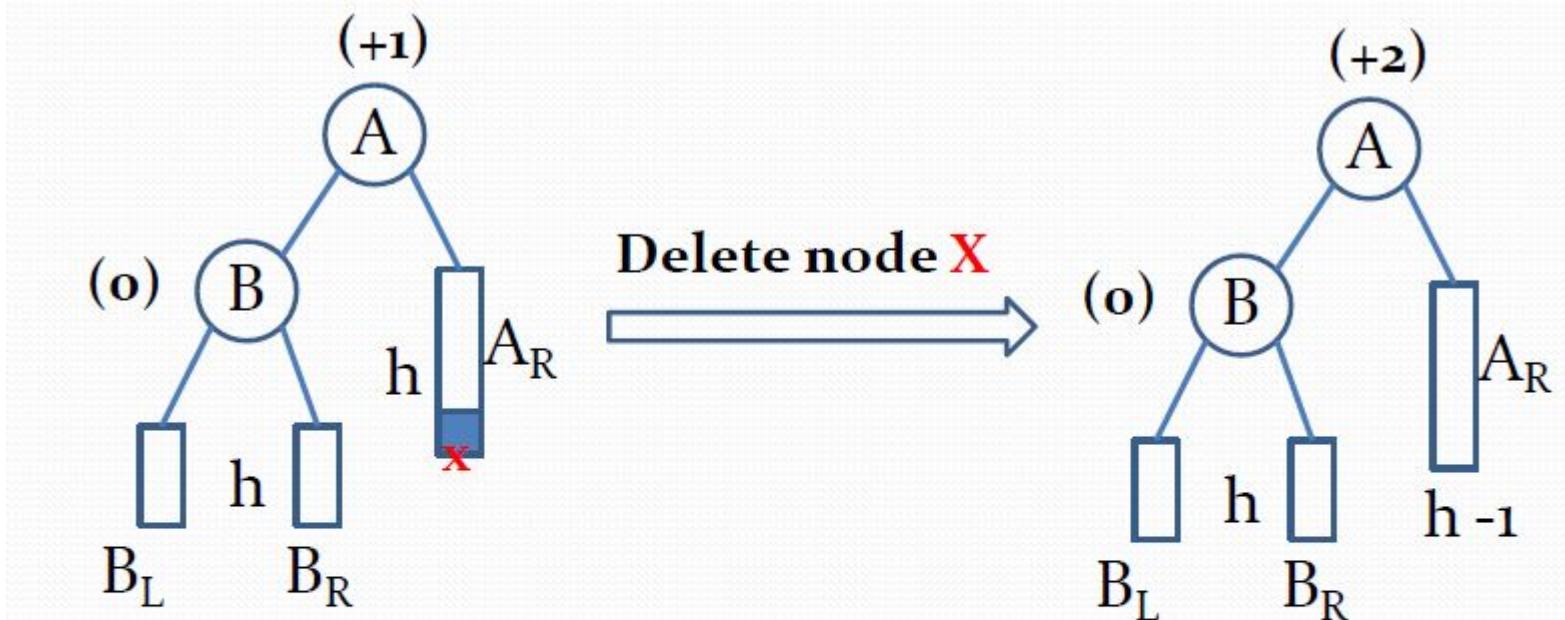


# Deletion in AVL Tree

---

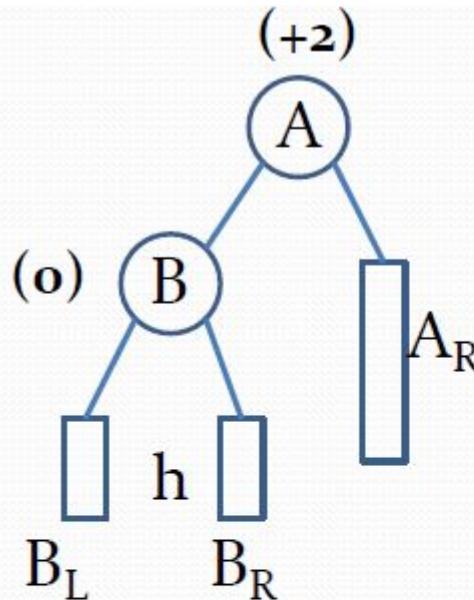
- ❑ LL Rotation is same to R0 and R1
- ❑ RR Rotation is same to L0 and L-1
- ❑ LR Rotation is same to R-1
- ❑ RL Rotation is same to L1

# R0 Rotation

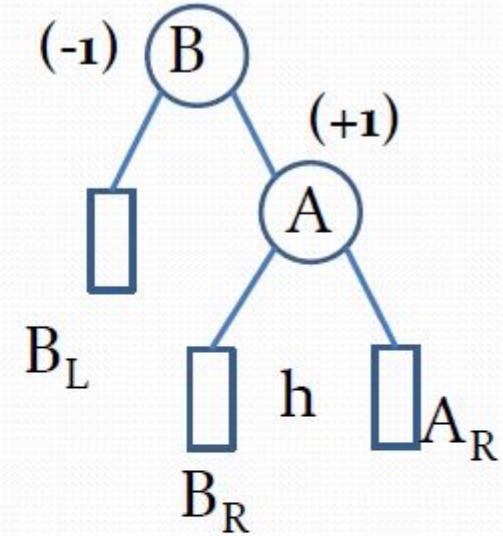


**Unbalanced AVL  
search tree after  
deletion of node x**

# R0 Rotation



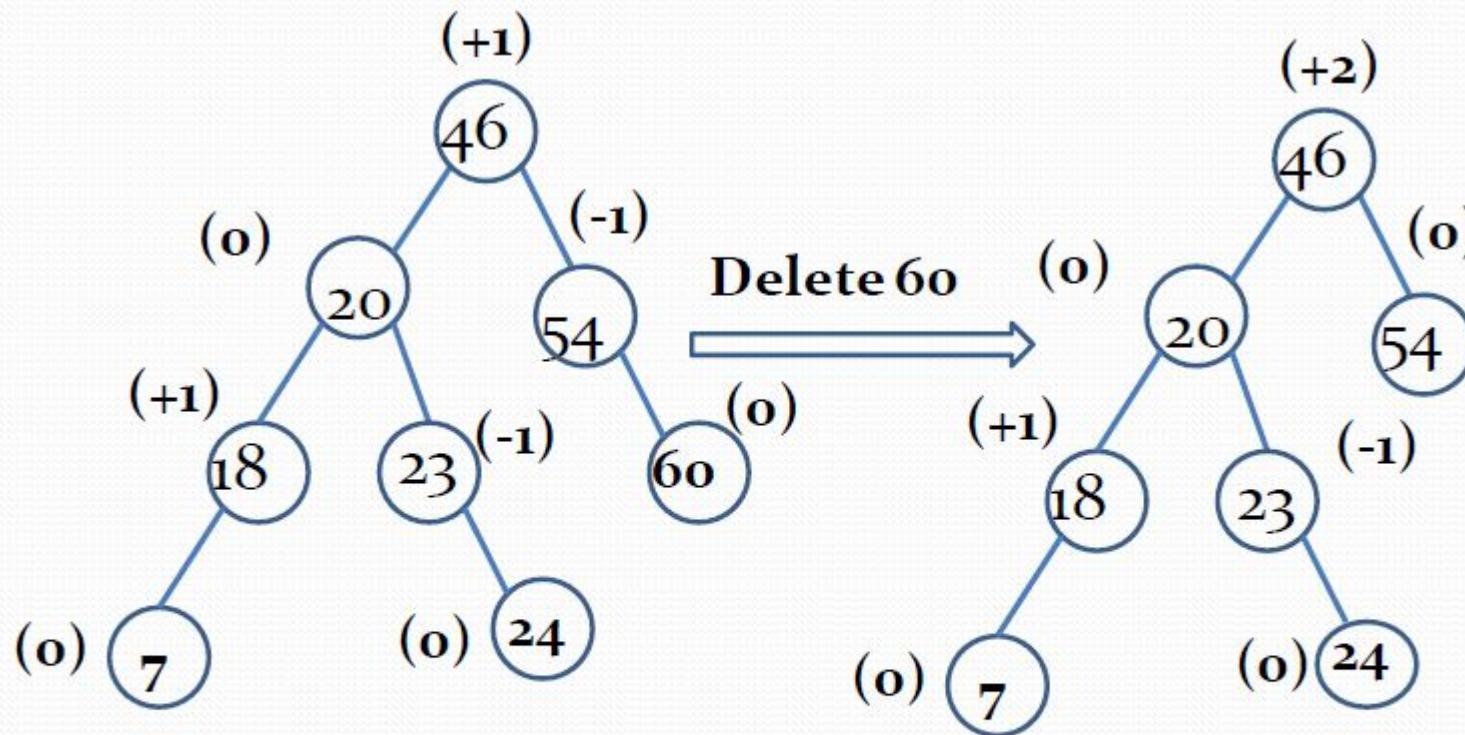
**Ro Rotation**  
BF(B) == o, use  
Ro rotation



**Unbalanced AVL  
search tree after  
deletion of x**

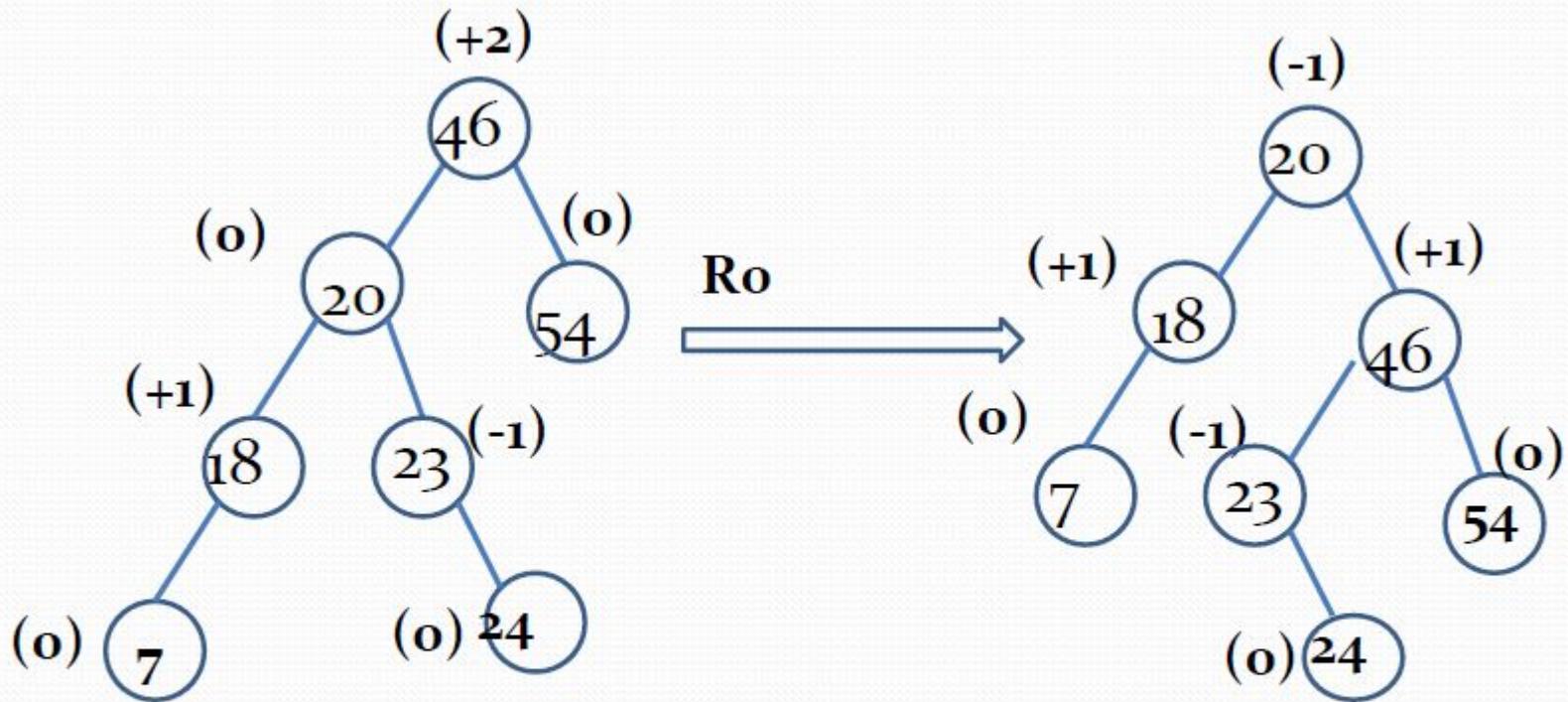
**Balanced AVL  
search tree after  
rotation**

# R0 Rotation Example



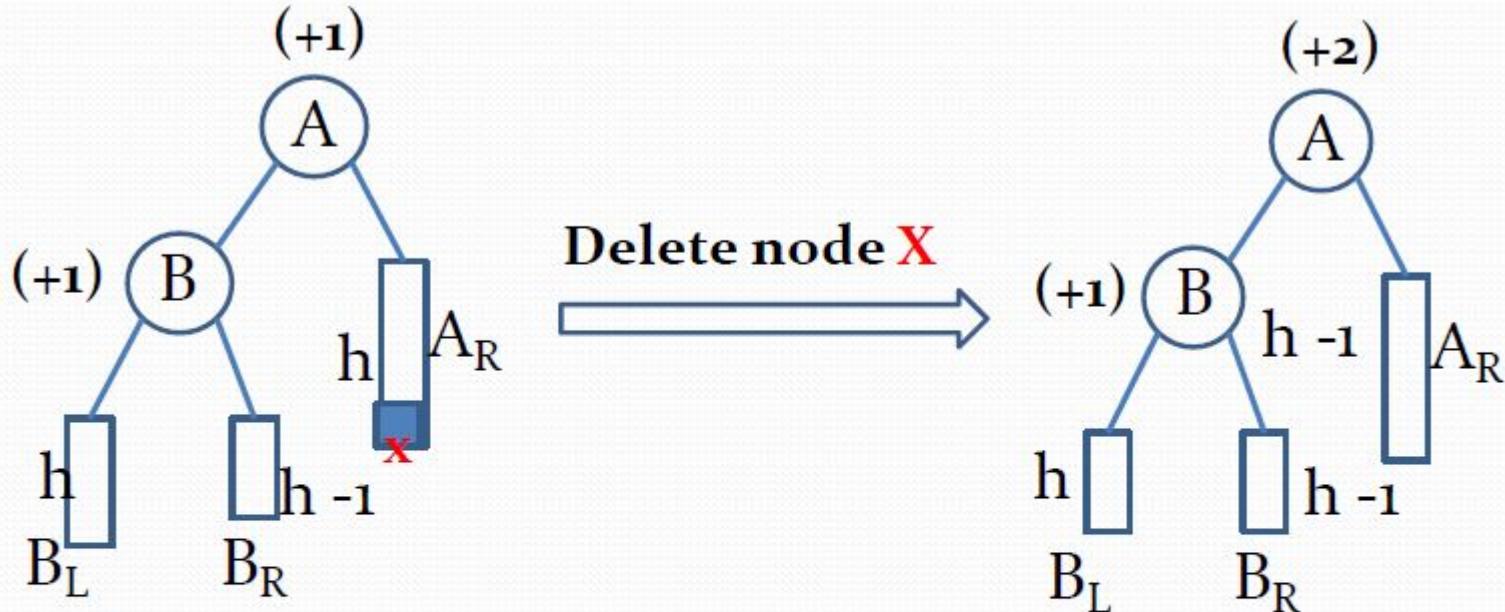
**Unbalanced AVL search  
tree after deletion**

# R0 Rotation Example



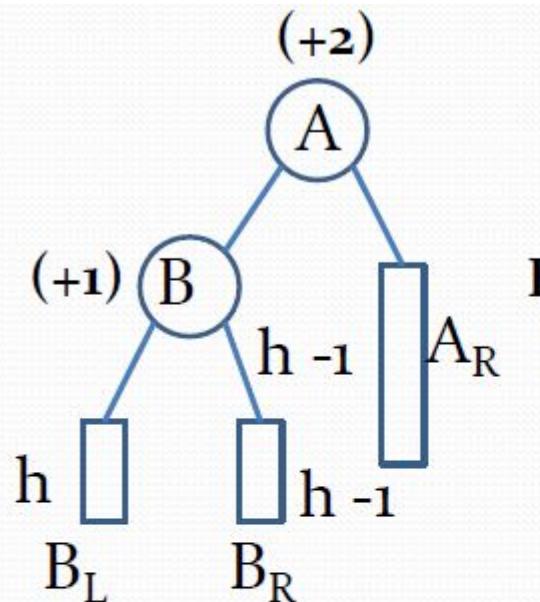
**Balanced AVL search tree  
after deletion**

# R1 Rotation



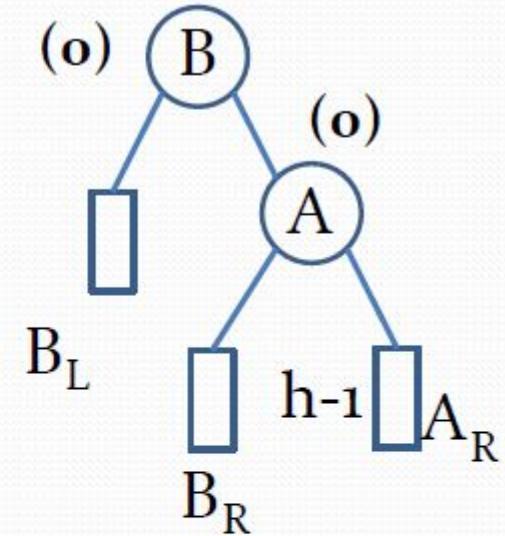
**Unbalanced AVL  
search tree after  
deletion of node x**

# R1 Rotation



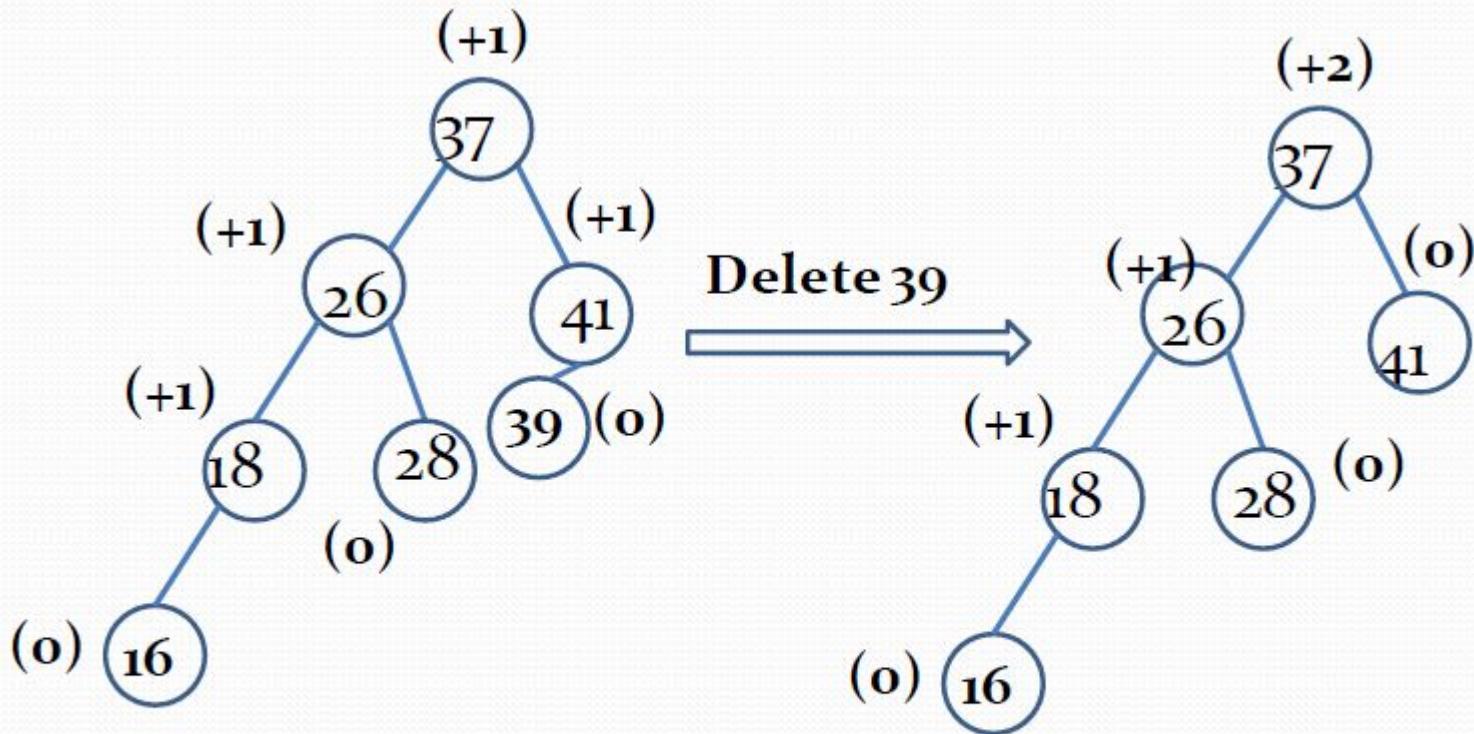
**R1 Rotation**

**BF(B) == 1, use  
R1 rotation**



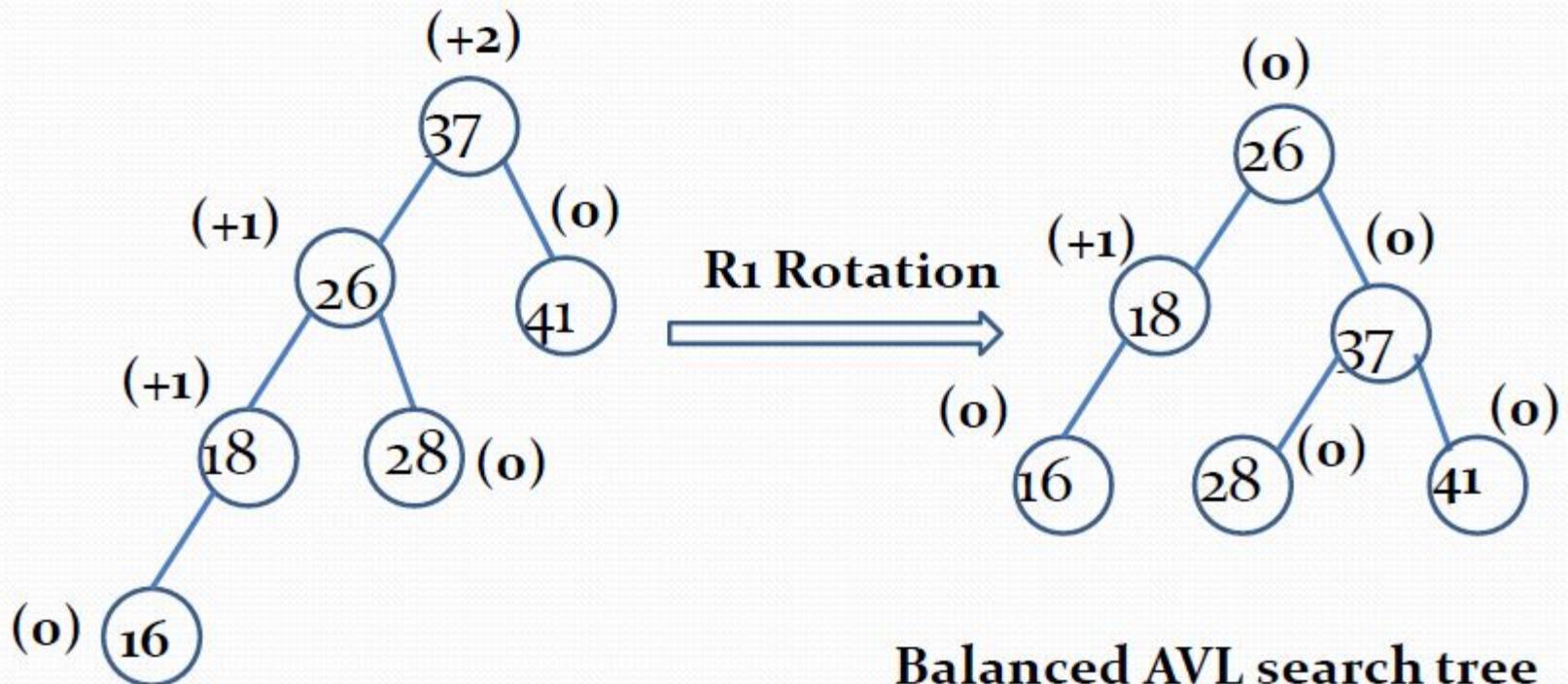
**Balanced AVL  
search tree after  
rotation**

# R1 Rotation Example

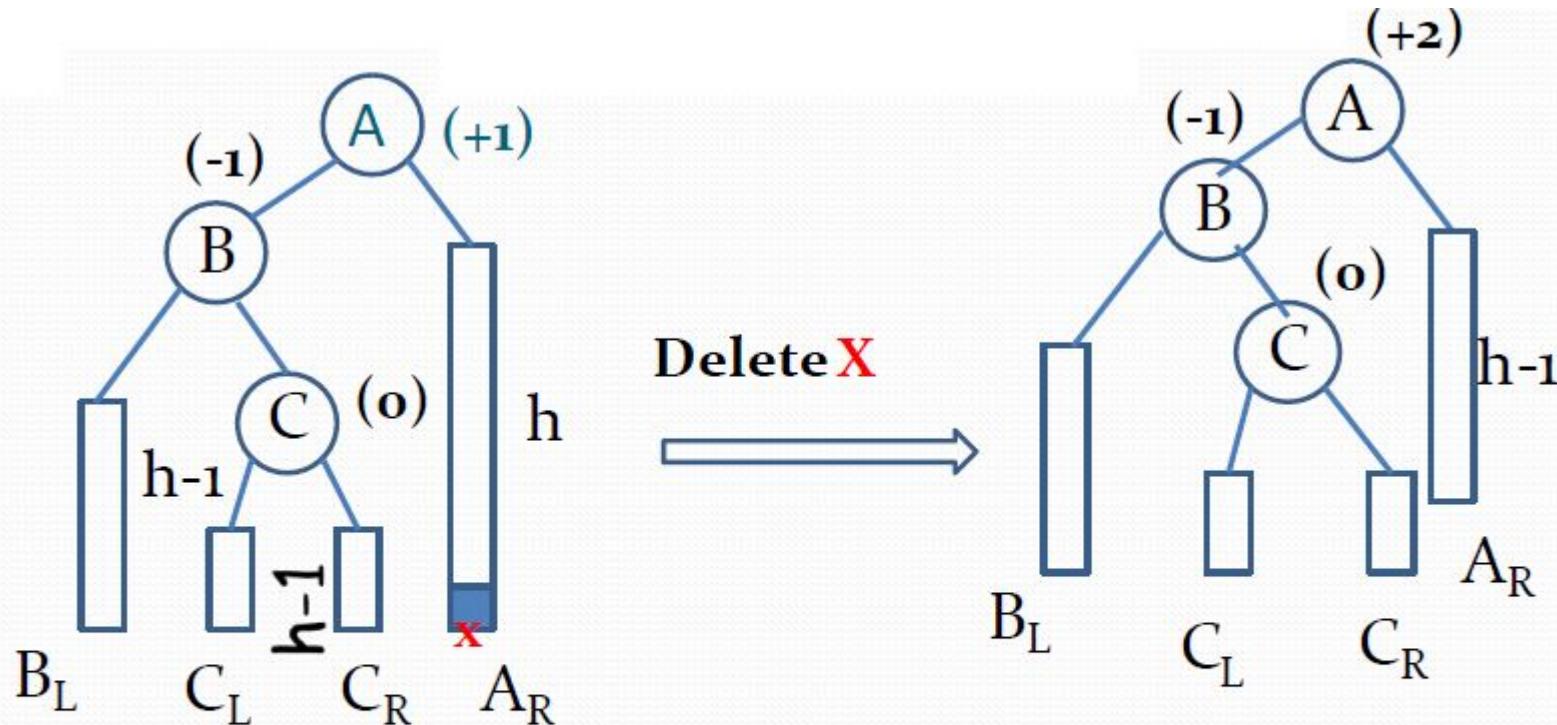


**Unbalanced AVL search tree after deletion**

# R1 Rotation Example

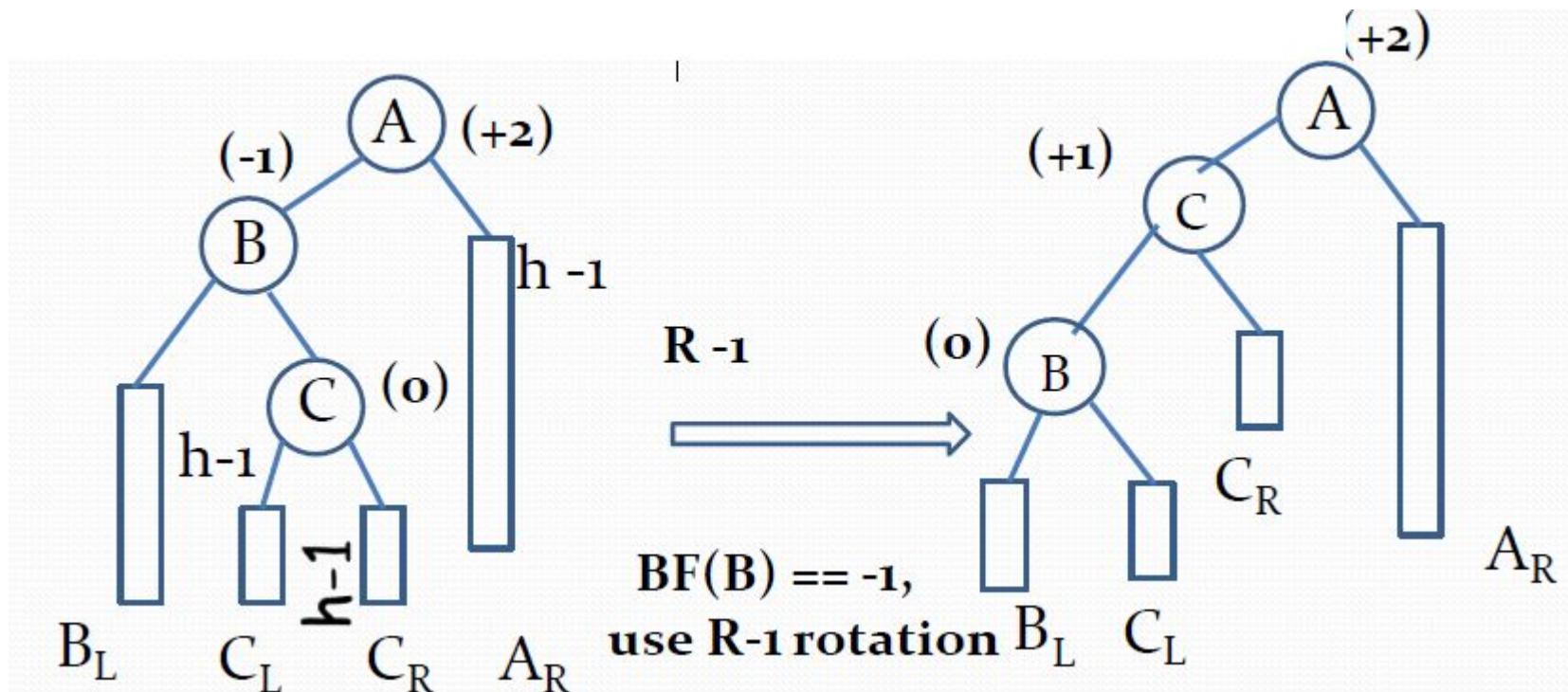


# R-1 Rotation

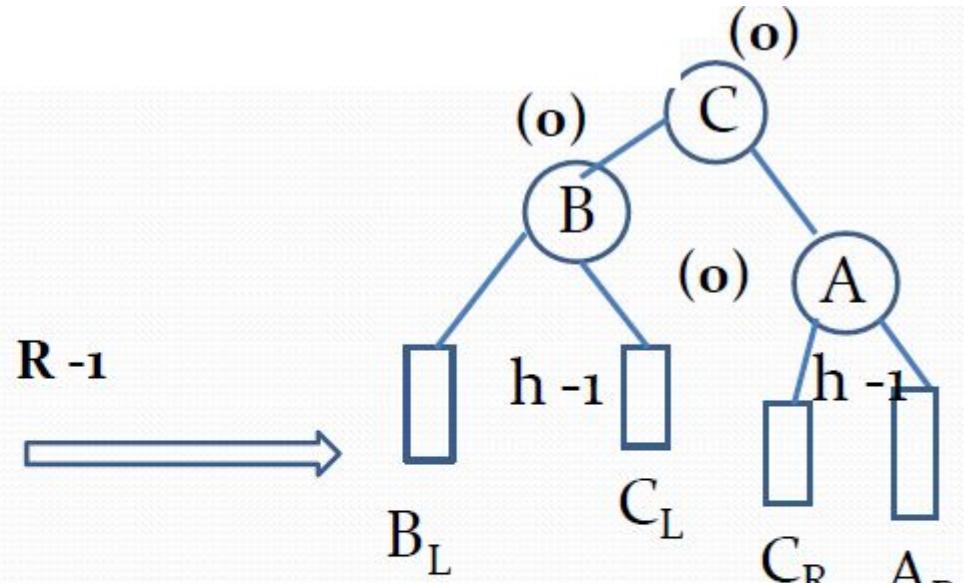
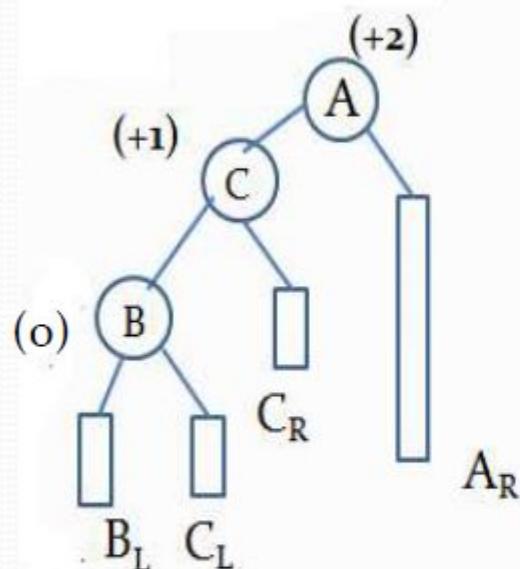


**Unbalanced AVL  
search tree after  
deletion**

# R-1 Rotation

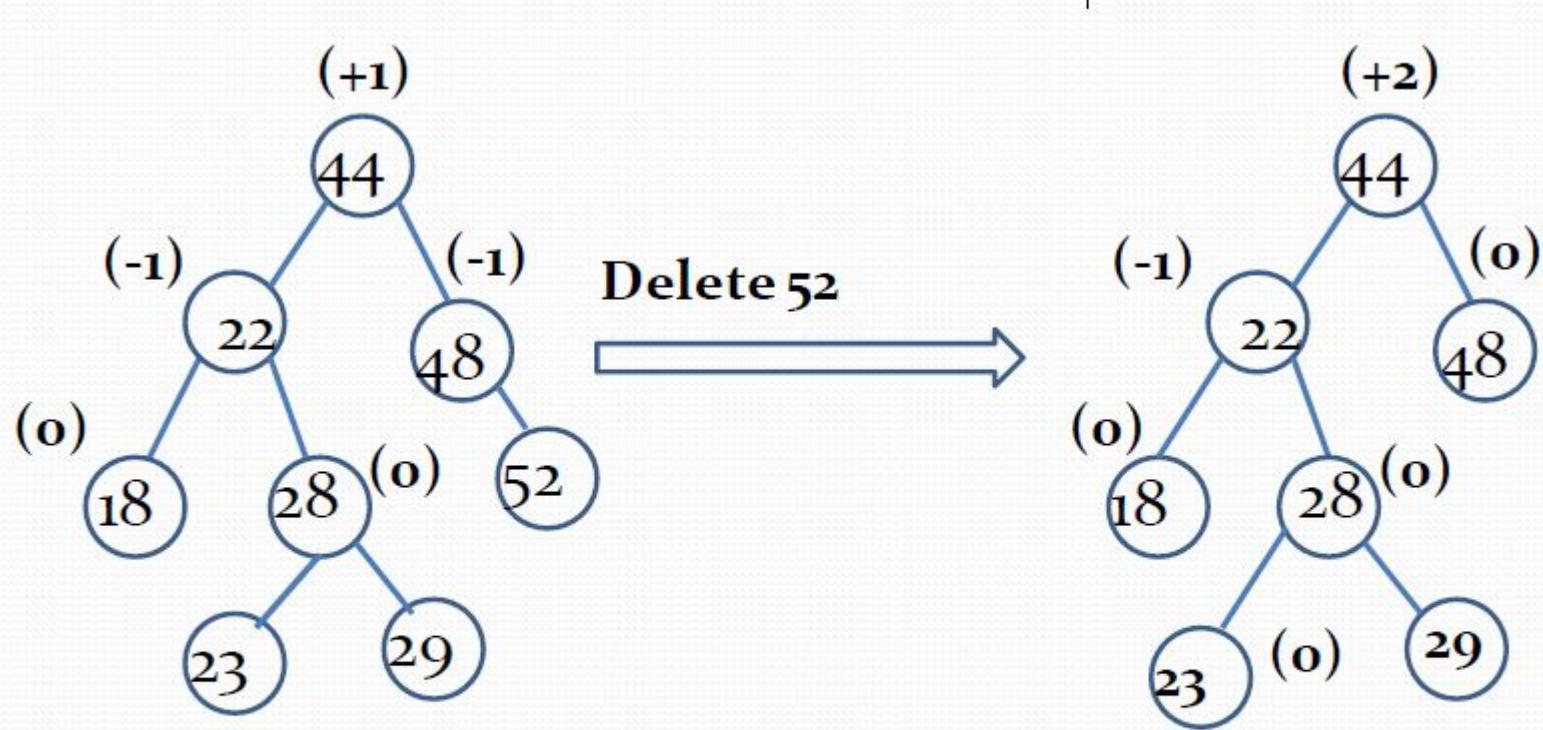


# R-1 Rotation



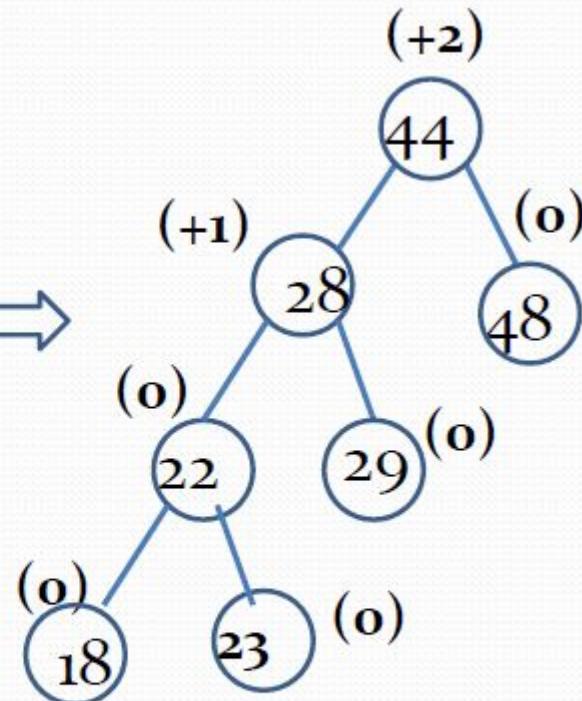
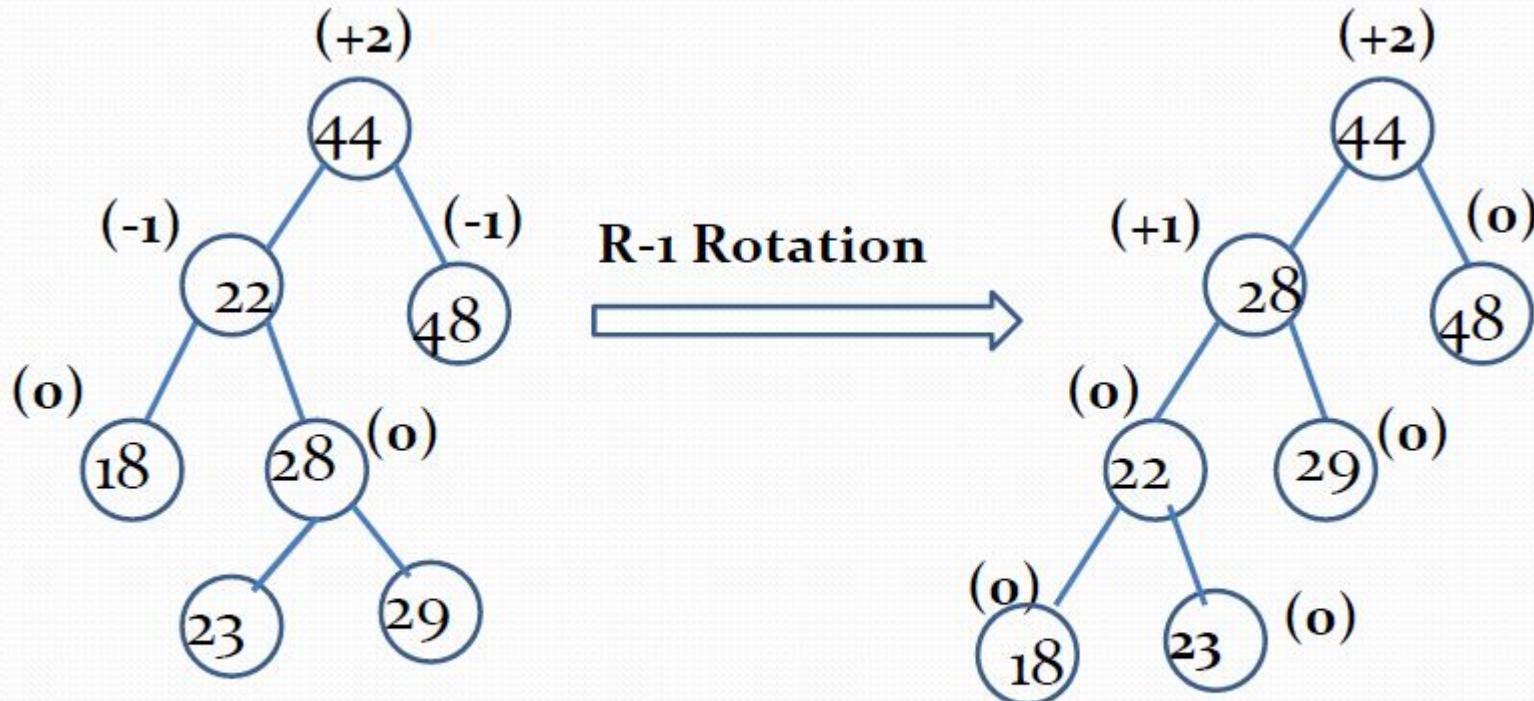
**Balanced AVL search  
tree after Rotation**

# R-1 Rotation Example



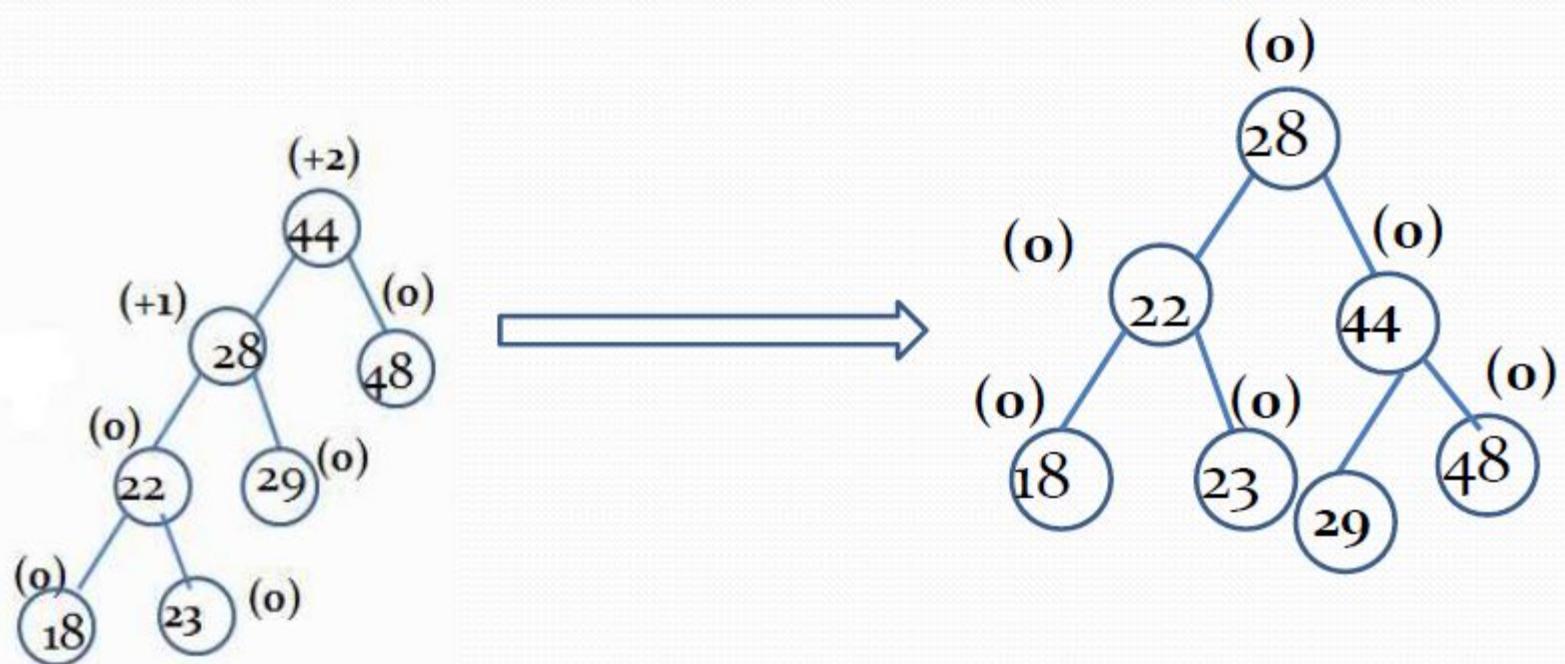
**Unbalanced AVL search  
tree after deletion**

# R-1 Rotation Example



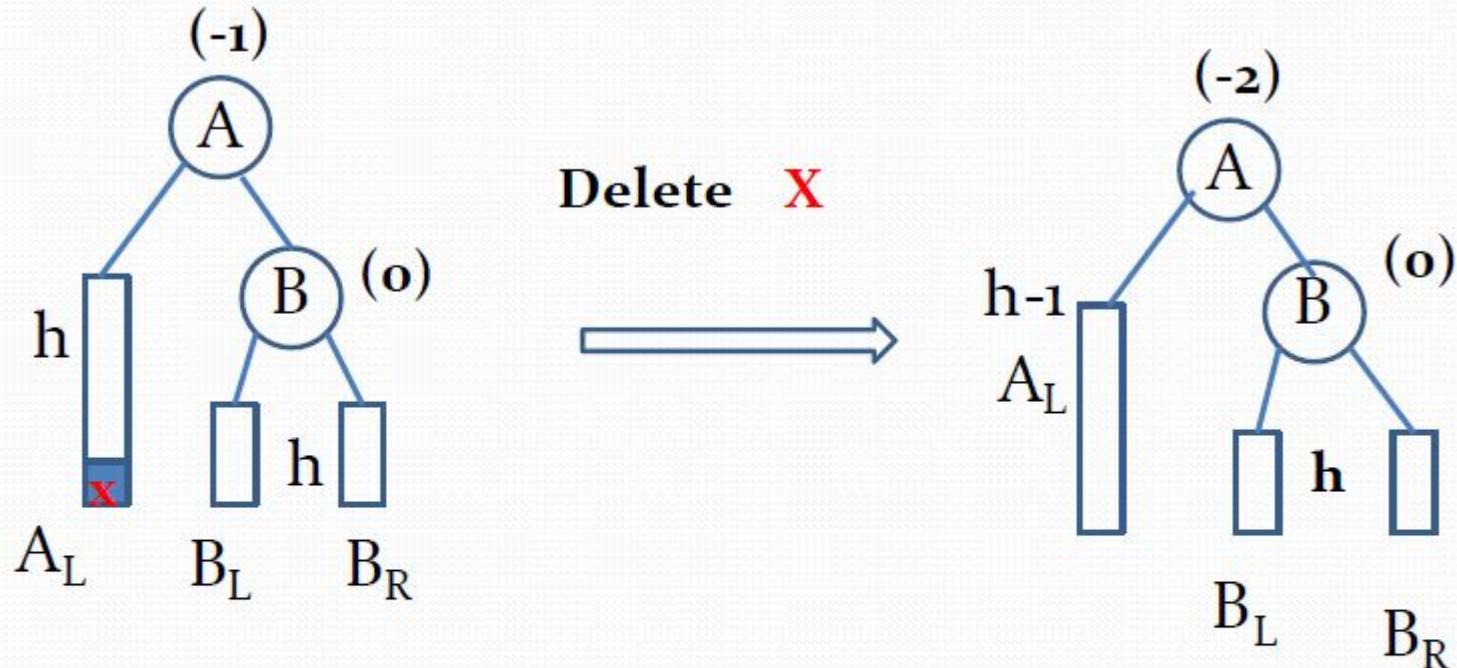
**Unbalanced AVL search  
tree after deletion**

# R-1 Rotation Example



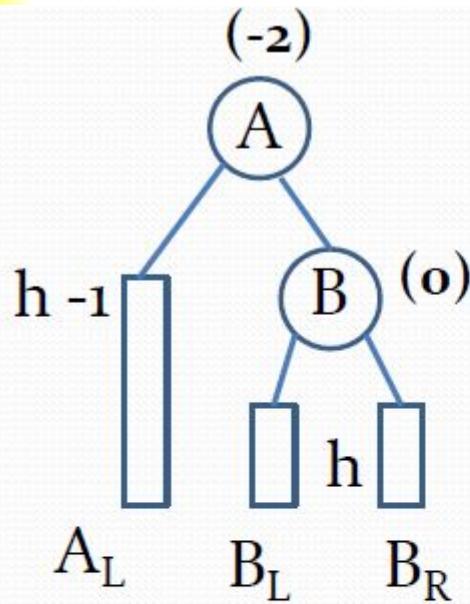
**Balanced AVL search tree  
after rotation**

# L0 Rotation

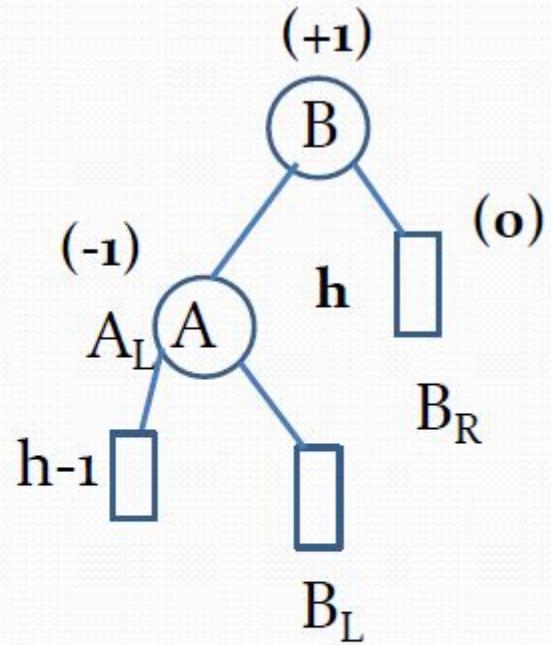


**Unbalanced AVL  
search tree after  
deletion**

# L0 Rotation

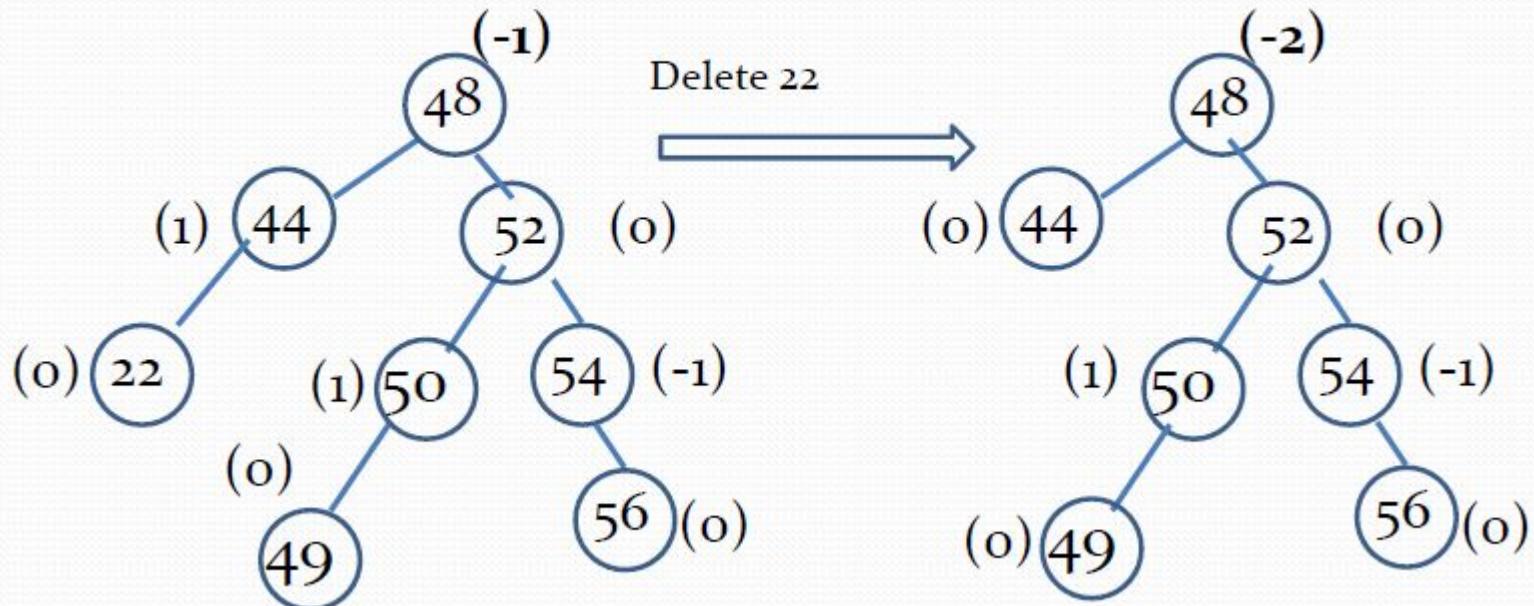


**Lo Rotation**

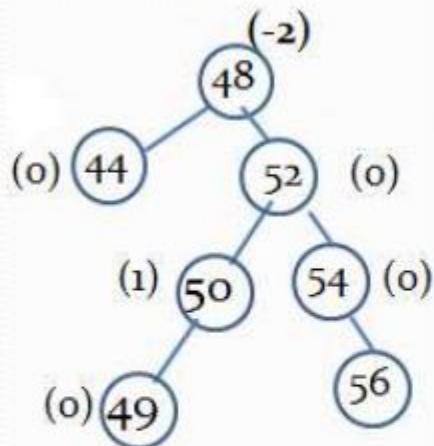


**Balanced AVL  
search tree after  
deletion**

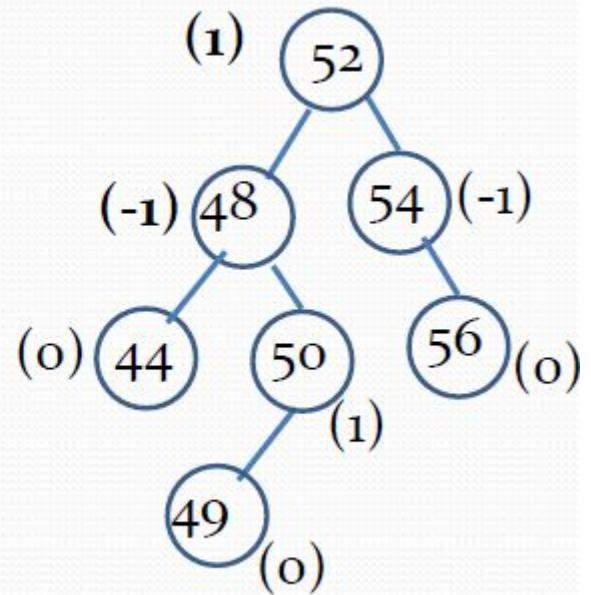
# L0 Rotation



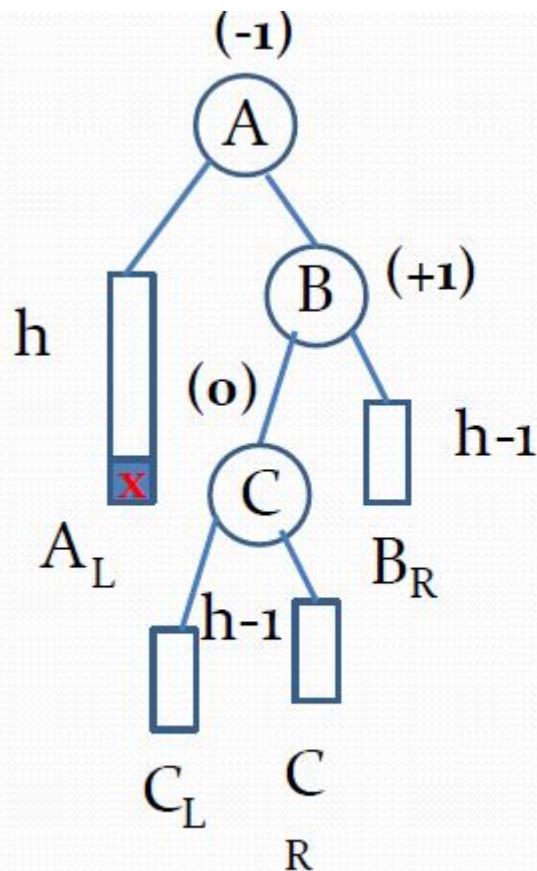
# L0 Rotation



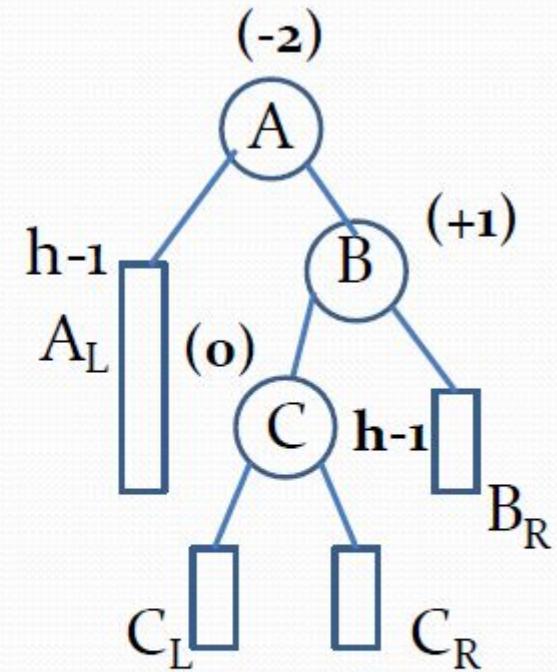
Lo Rotation



# L1 Rotation

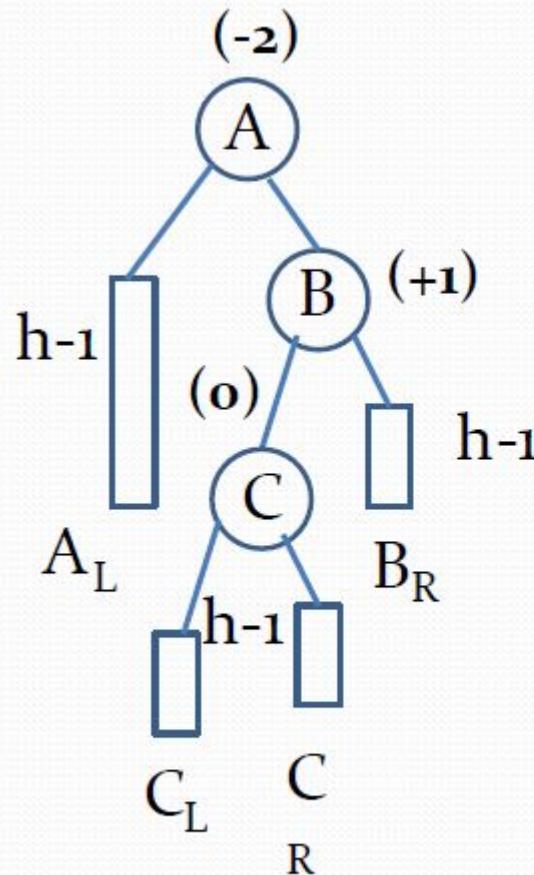


Delete X

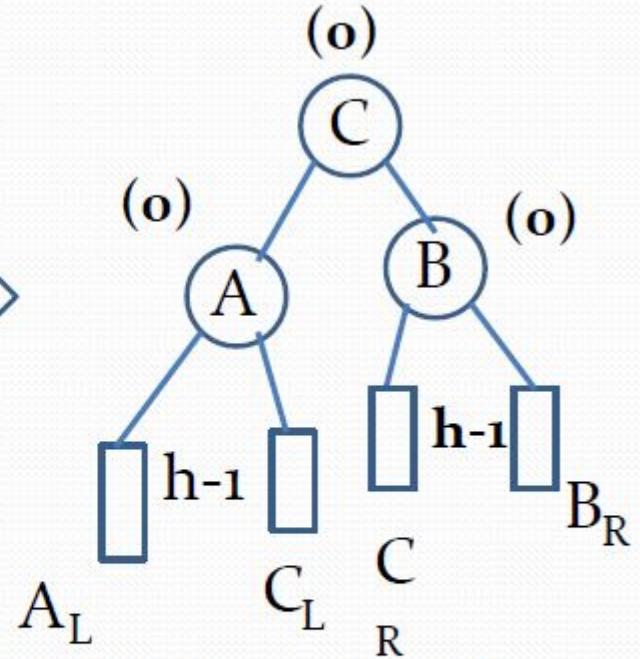


**Unbalanced AVL  
search tree after  
deletion**

# L1 Rotation

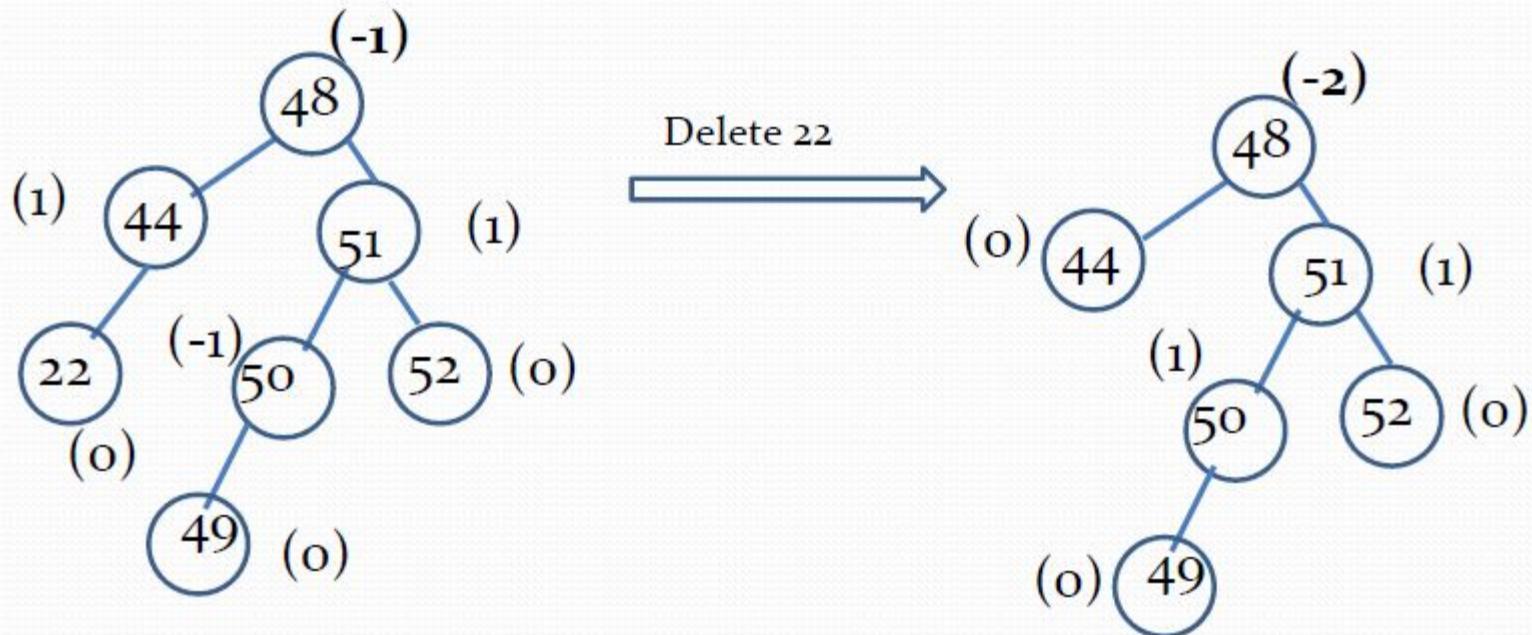


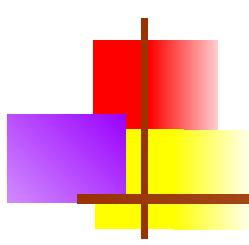
**L1 Rotation**



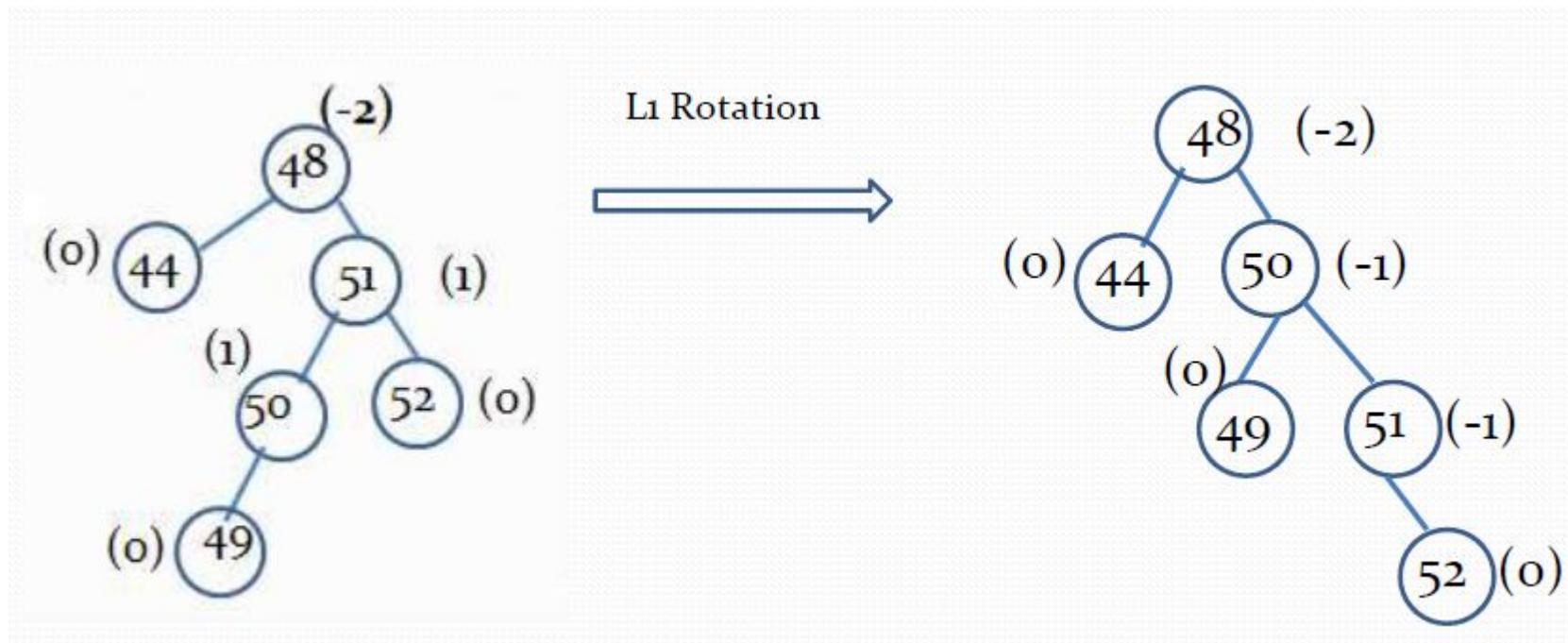
**Unbalanced AVL search tree after deletion**

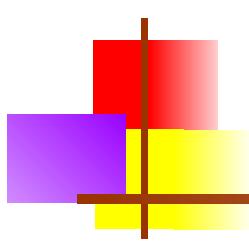
# L1 Rotation



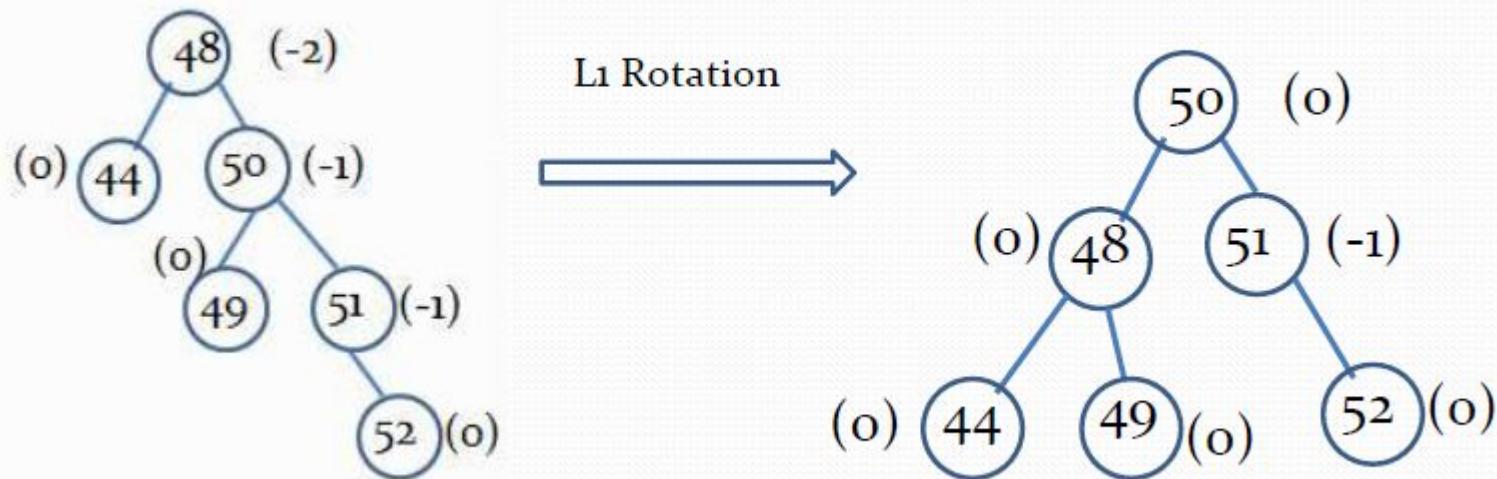


# L1 Rotation

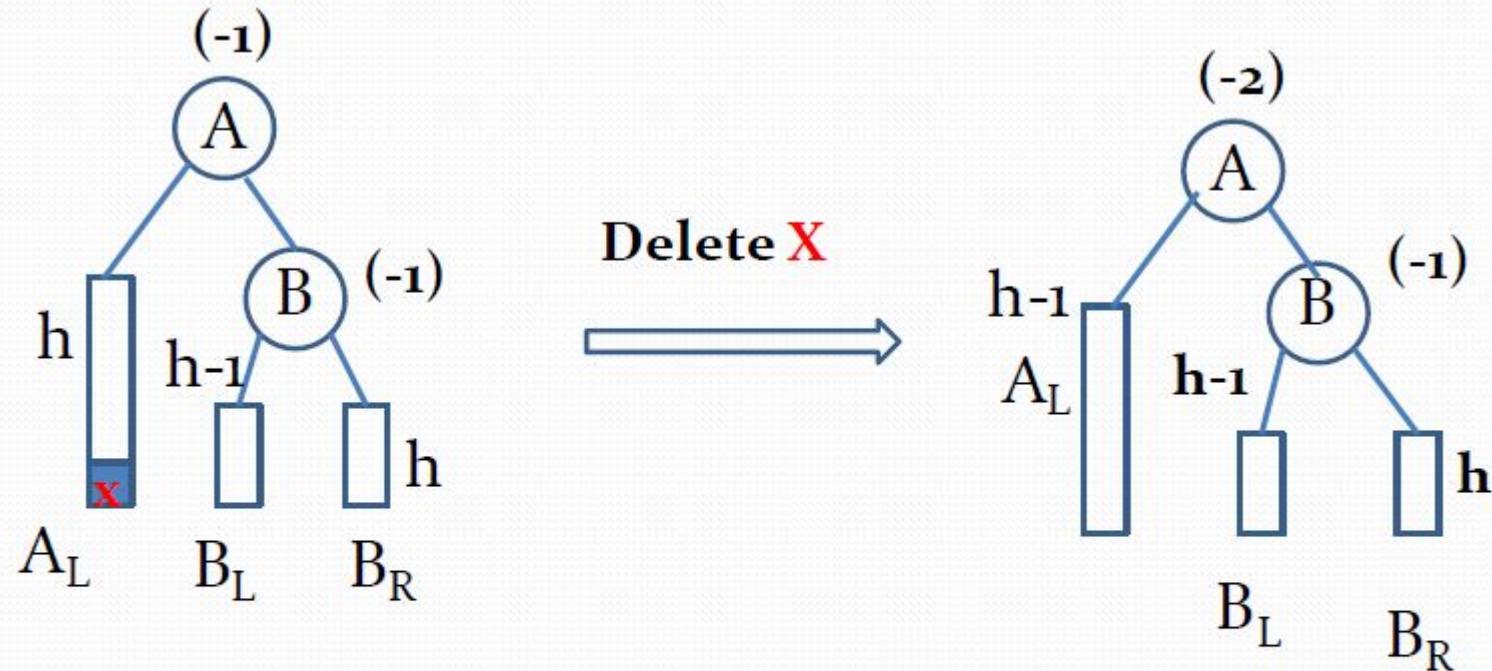




# L1 Rotation

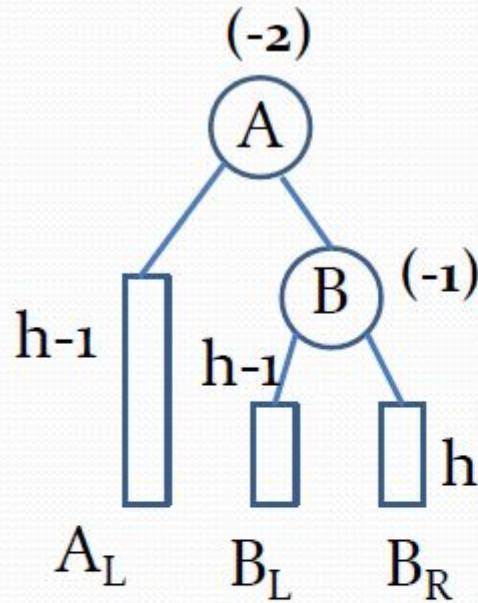


# L-1 Rotation

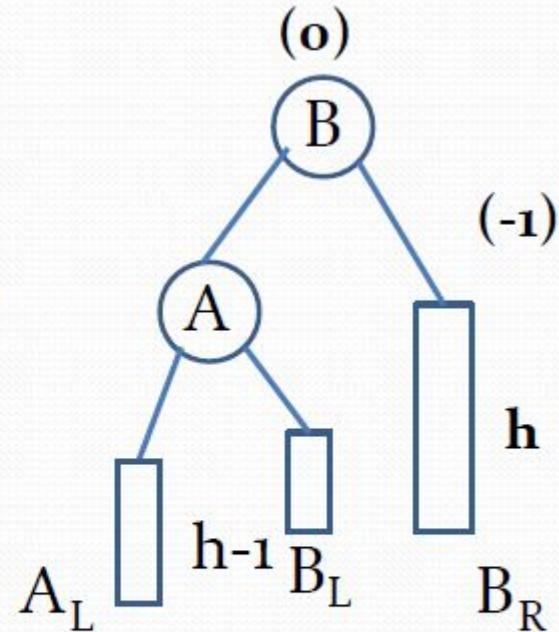


**Unbalanced AVL  
search tree after  
deletion**

# L-1 Rotation

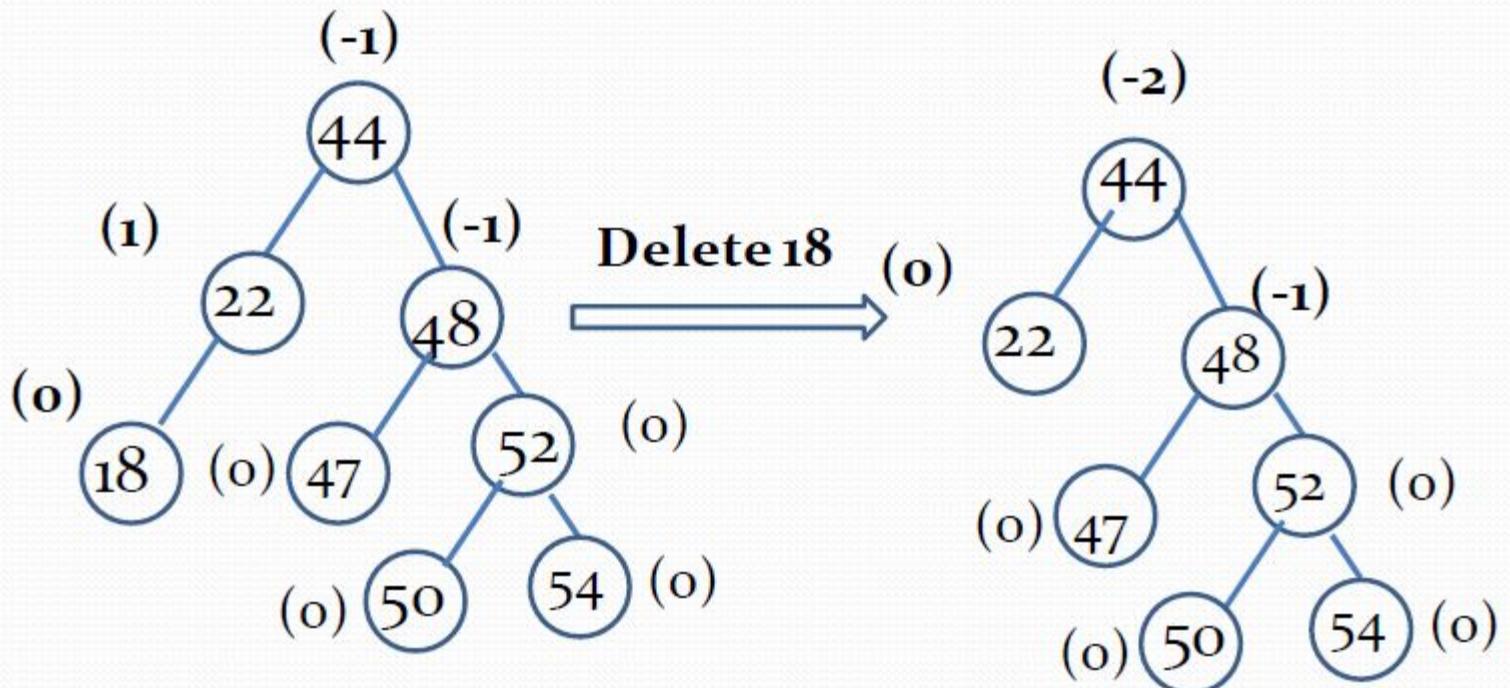


**L-1 Rotation**

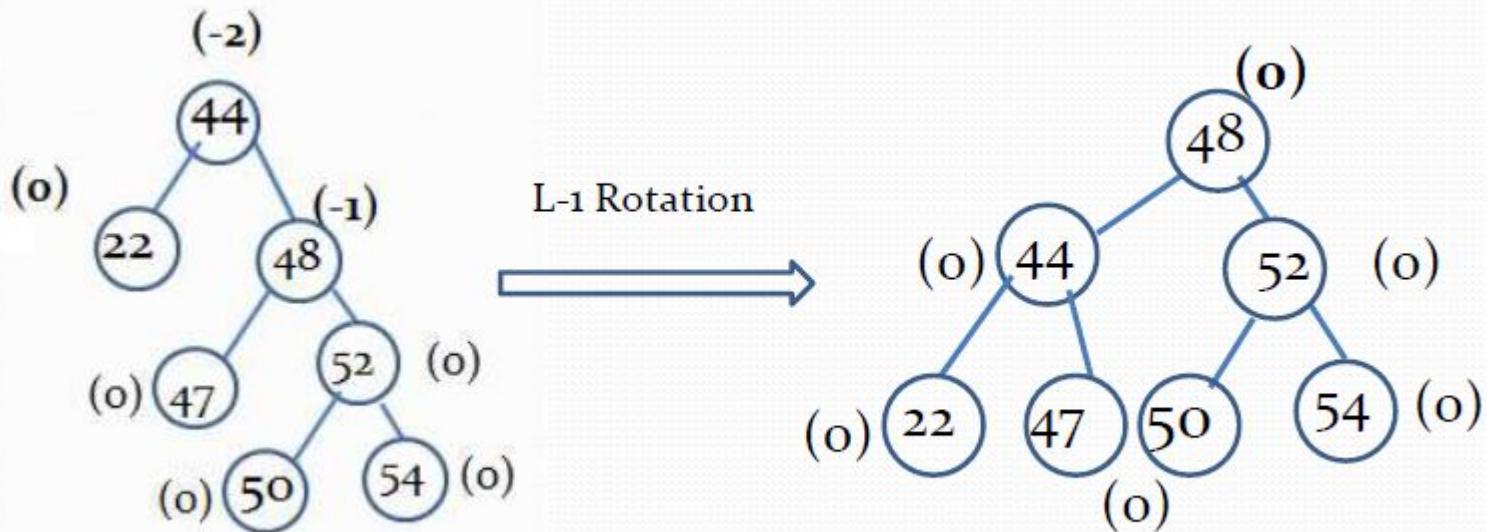


**Balanced AVL  
search tree after  
deletion**

# L-1 Rotation



# L-1 Rotation



# Advanced Tree Data Structures

---

Prepared by  
Dr Annushree Bablani

# Balanced Search Trees

- A search-tree data structure for which a height of  $O(\log n)$  is guaranteed when implementing a dynamic set of  $n$  items.
- Examples:
  - AVL trees ( Discussed in Unit-1)
  - 2-4 trees (*This Lecture*)
  - B+-trees
  - Red-black trees

# 2-4 Trees

---

- Search Trees (but not binary)
- Also known as 2-4, 2-3-4 trees
- Very important as basis for Red-Black trees

# Multi-way Search Trees

---

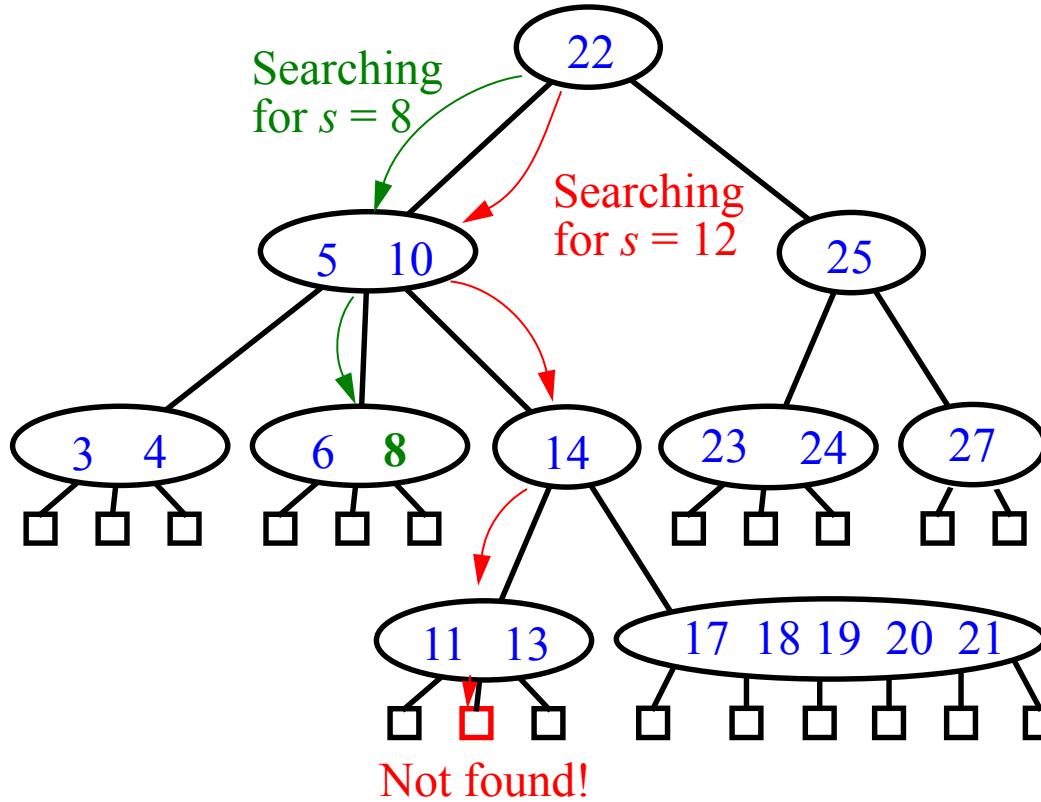
- Each internal node of a multi-way search tree  $T$ :
  - has at least two children
  - stores a collection of items of the form  $(k, x)$ , where  $k$  is a key and  $x$  is an element
  - contains  $d - 1$  items, where  $d$  is the number of children
  - “contains” 2 pseudo-items:  $k_0 = -\infty$ ,  $k_d = \infty$
- Children of each internal node are “between” items
  - all keys in the subtree rooted at the child fall between keys of those items
- External nodes are just placeholders

# Multi-way Searching

---

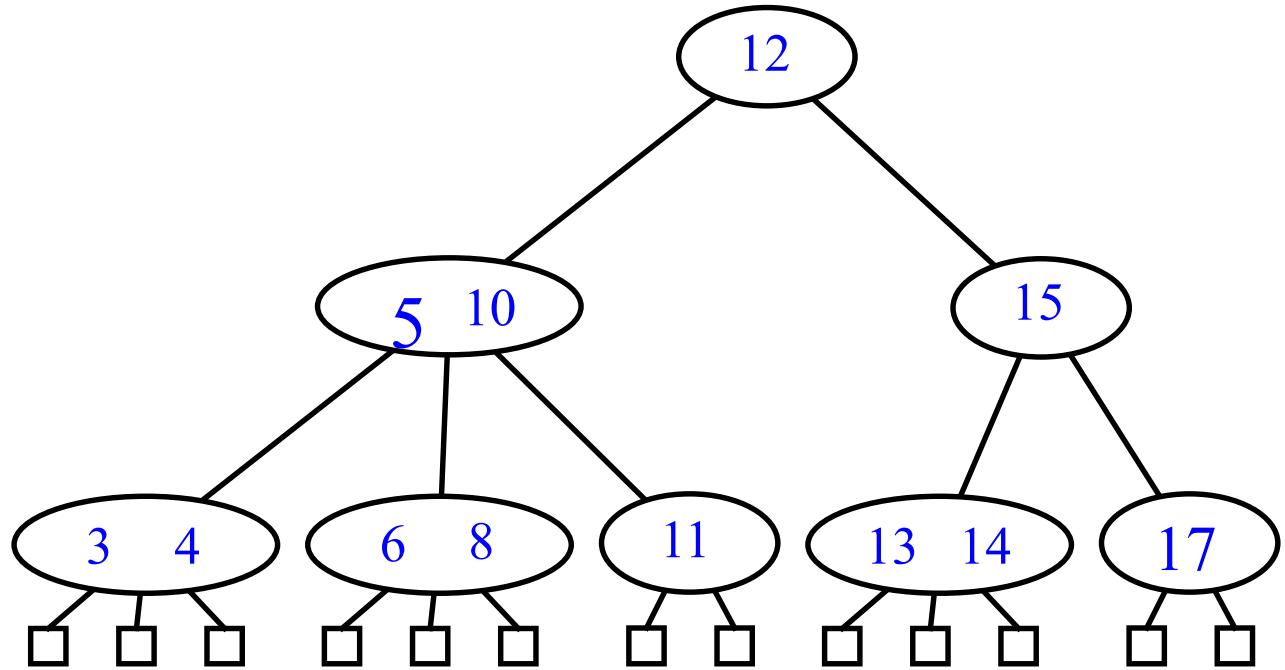
- Similar to binary searching
- If search key  $s < k_1$ , search the leftmost child
- If  $s > k_{d-1}$ , search the rightmost child
- That's it in a binary tree; what about if  $d > 2$ ?
- Find two keys  $k_{i-1}$  and  $k_i$  between which  $s$  falls, and search the child  $v_i$ .

# Multi-way Searching



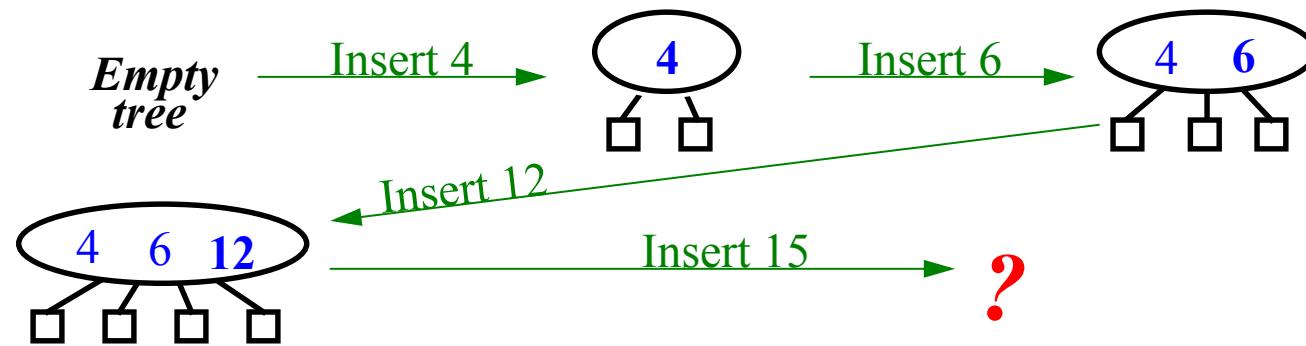
# (2,4) Trees

- At most 4 children
- All external nodes have same depth
- Height  $h$  of (2,4) tree is  $O(\log n)$ .
- How is this fact useful in searching?



# (2,4) Insertion

- Always maintain depth condition
- Add elements only to existing nodes

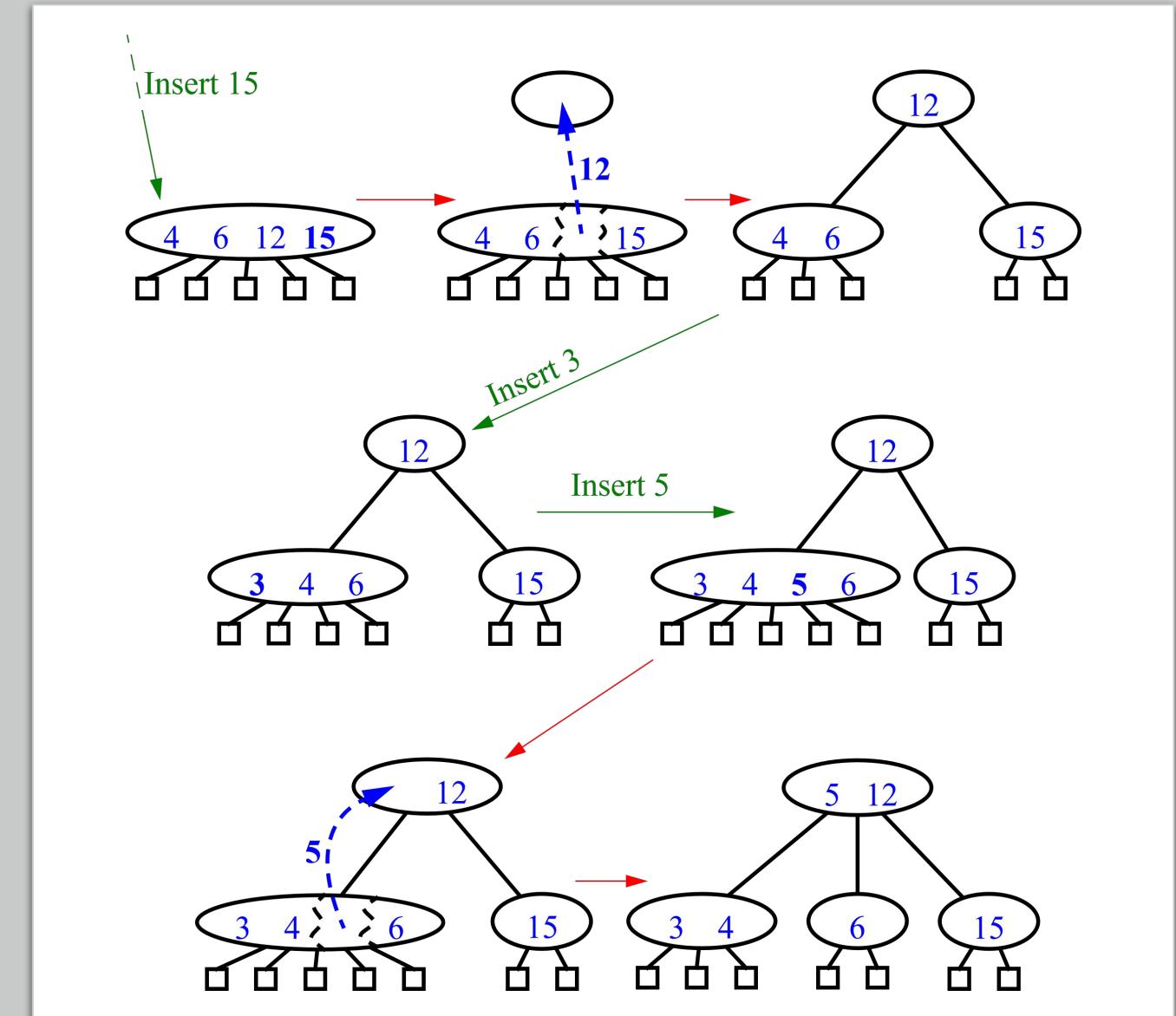


# (2,4) Insertion

---

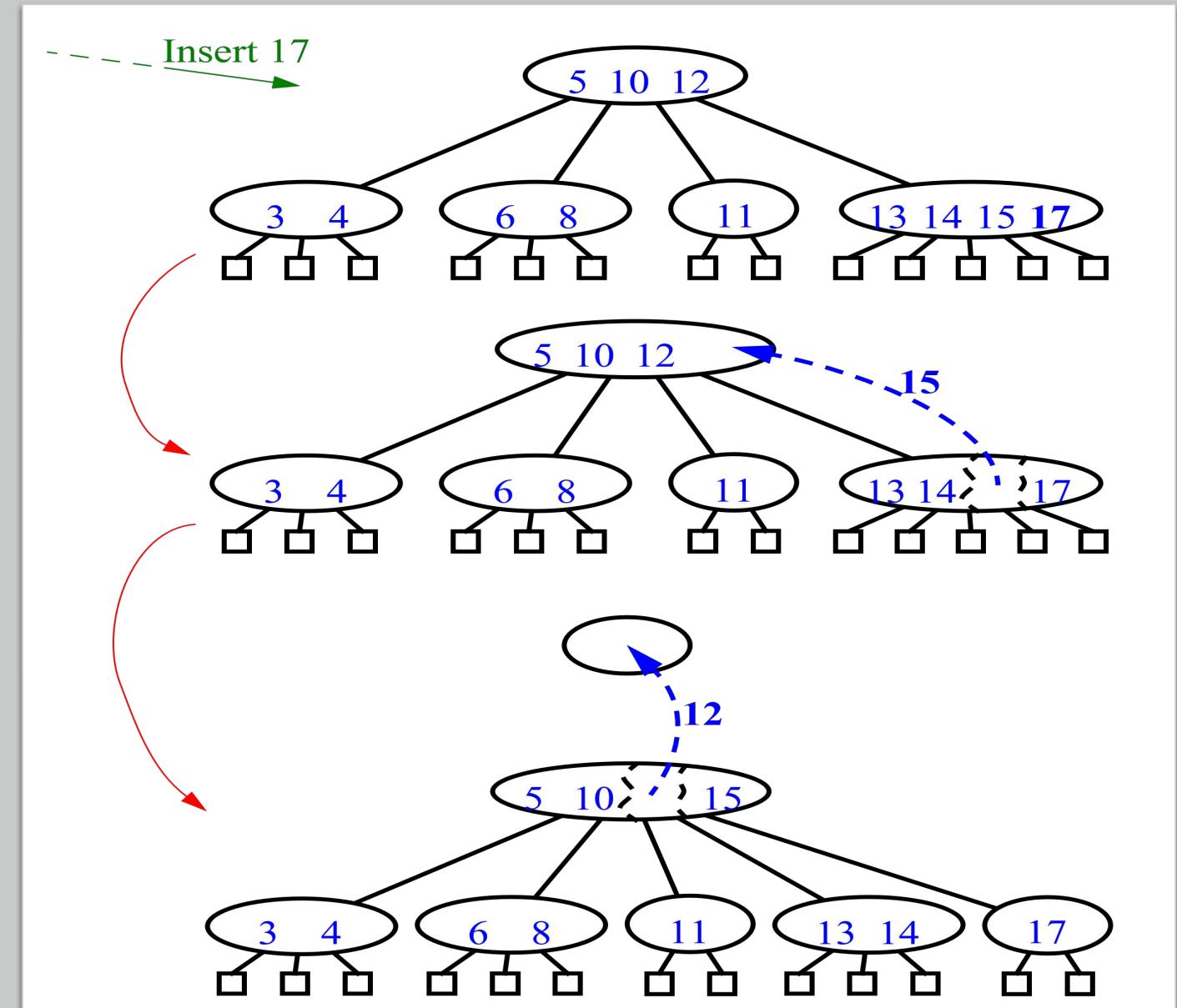
- What if that makes a node too big?
  - *overflow*
- Must perform a *split* operation
  - replace node  $v$  with two nodes  $v'$  and  $v''$
  - $v'$  gets the first two keys
  - $v''$  gets the last key - send the other key up the tree
    - if  $v$  is root, create new root with third key
    - otherwise just add third key to parent

# (2,4) Insertion (cont.)



# (2,4) Insertion (cont.)

- Tree always grows from the top, maintaining balance
- What if parent is full?
  - Do the same thing
- Overflow cascade all the way up to the root
  - still at most  $O(\log n)$

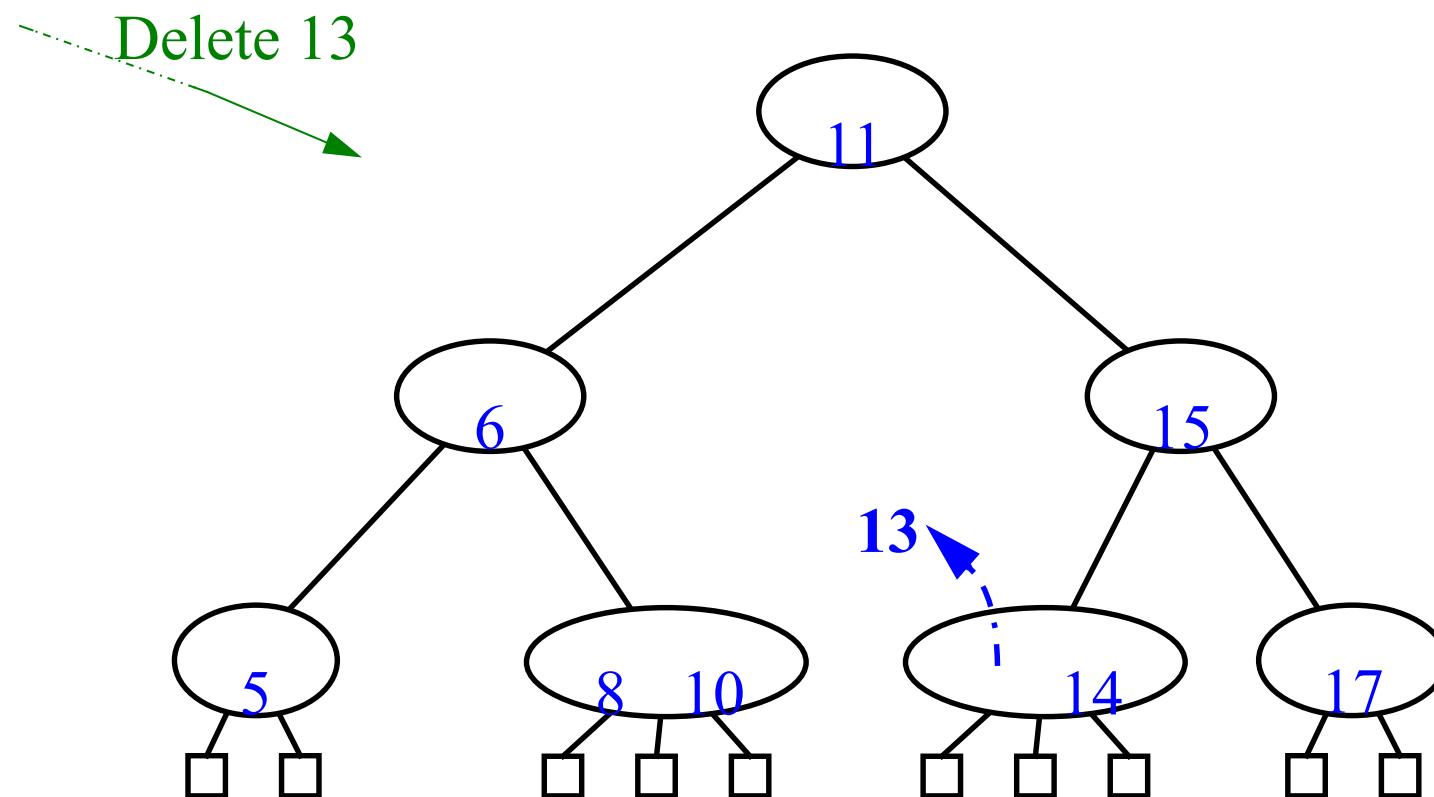


# (2,4) Deletion

---

- First of all, find the key
  - simple multi-way search
- Then, reduce to the case where item to be deleted is at the bottom of the tree
  - Find item which precedes it in in-order traversal
  - Swap them
- Remove the item

# (2,4) Deletion

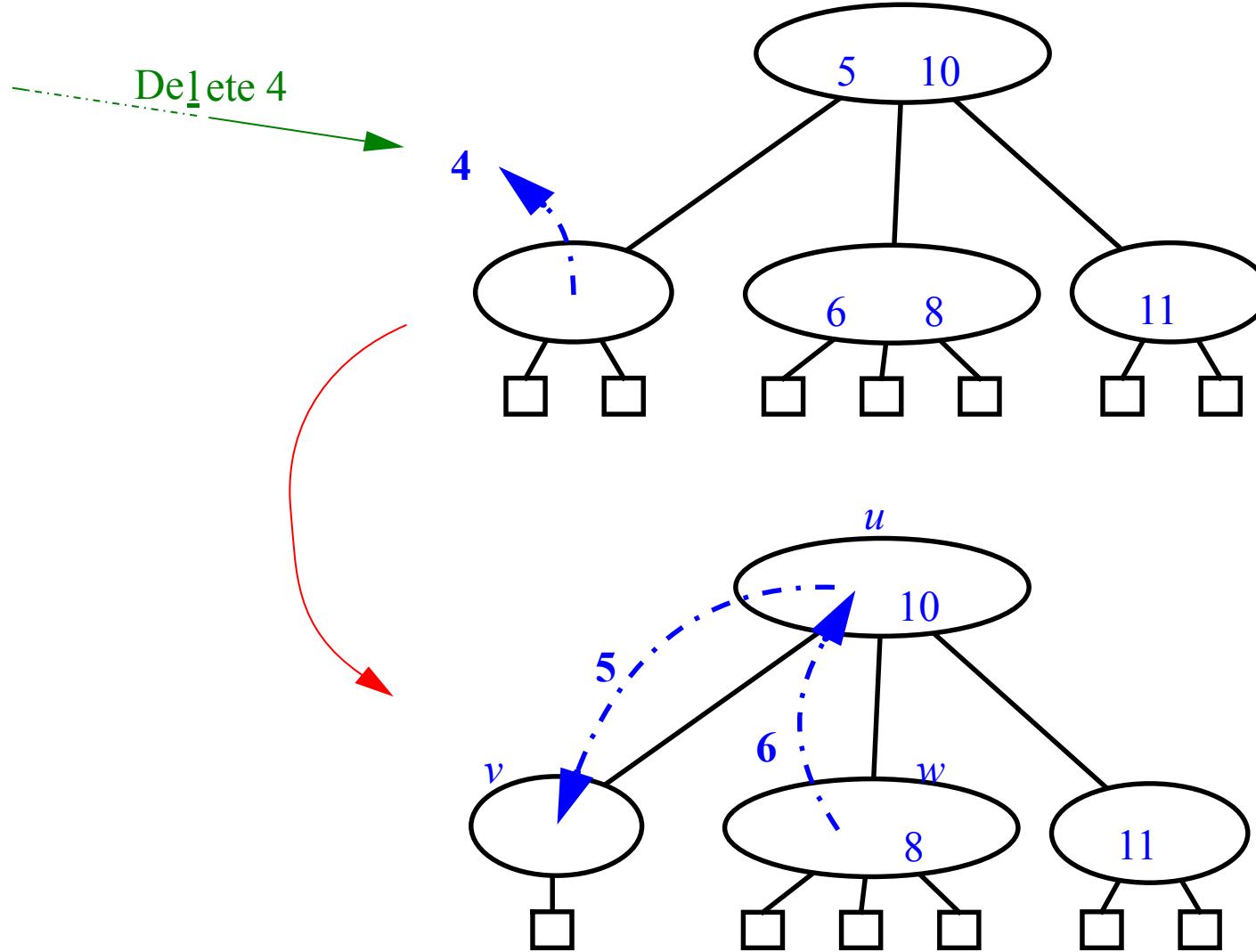


# (2,4) Deletion

---

- Removing from 2-nodes
- Not enough items in the node
  - *underflow*
- Pull an item from the parent, replace it with an item from a sibling
  - **called *transfer***

# (2,4) Deletion

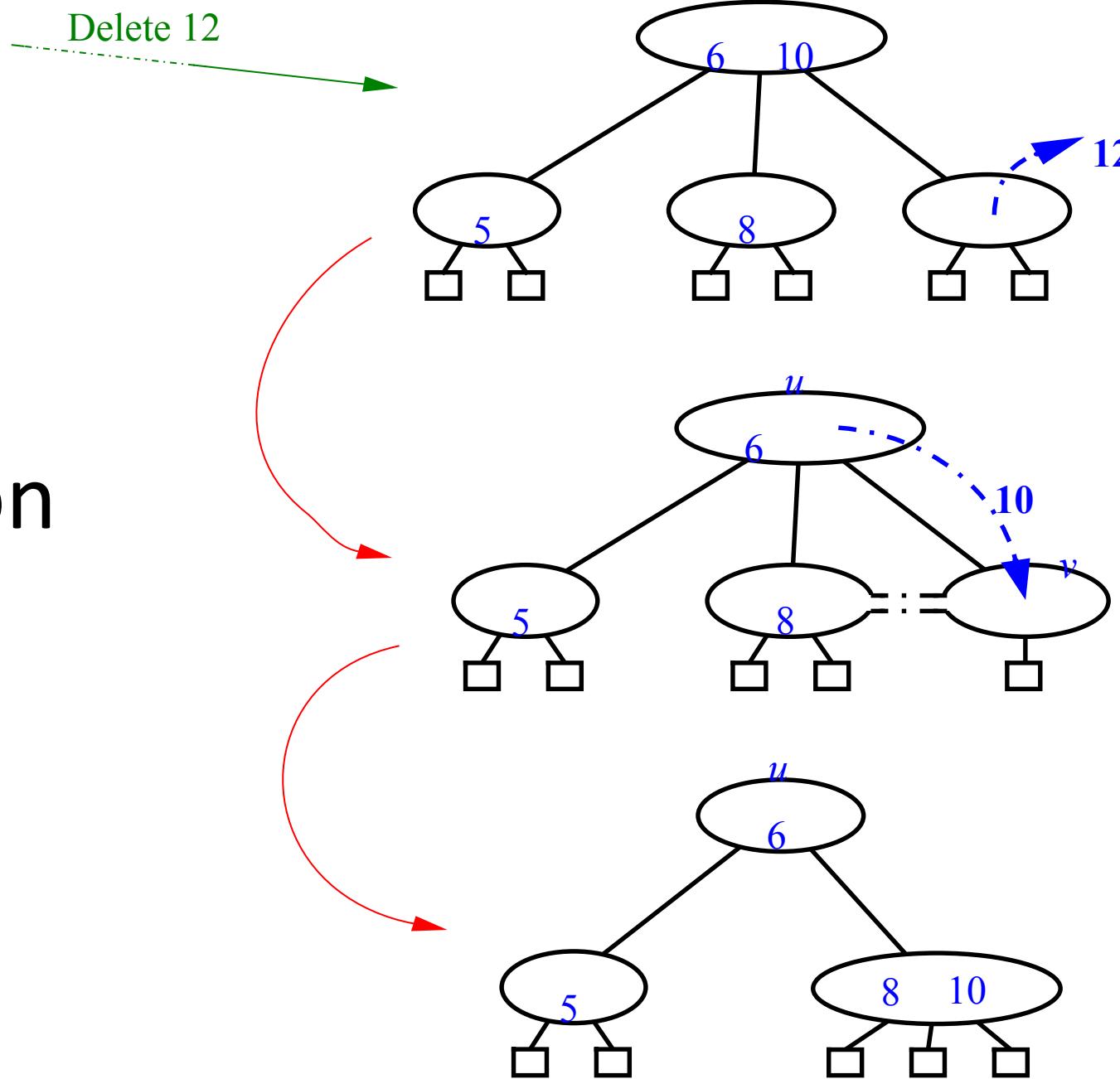


# (2,4) Deletion

---

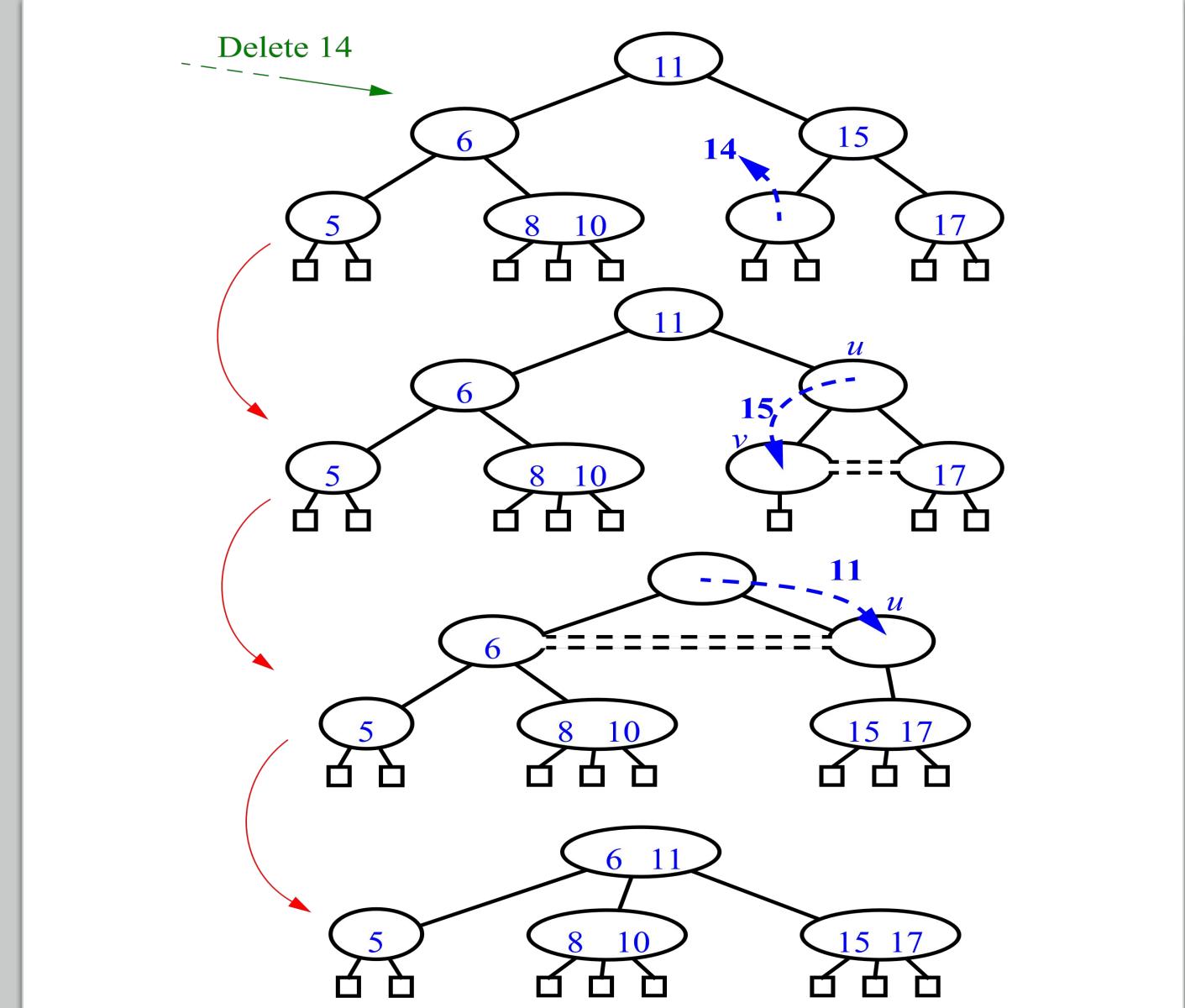
- What happens if siblings are 2-nodes?
- Could we just pull one item from the parent?
  - too many children
- But maybe...
- We know that the node's sibling is just a 2-node
- So we *fuse* them into one after removing an item from the parent,

## (2,4) Deletion



# (2,4) Deletion

- what if the parent was a 2-node?
- Underflow can cascade up the tree, too.



# (2-4) Trees Conclusion

---

- The height of a (2,4) tree is  $O(\log n)$ .
- Split, transfer, and fusion each take  $O(1)$ .
- Search, insertion and deletion each take  $O(\log n)$ .

# *Red Black Trees*

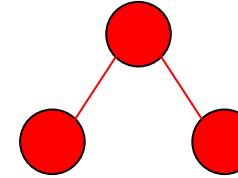
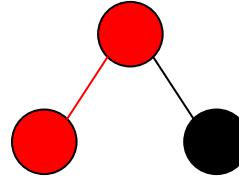
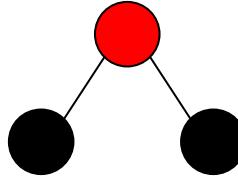
Prepared by  
Dr. Annushree Bablani

# A balanced binary search tree- Review

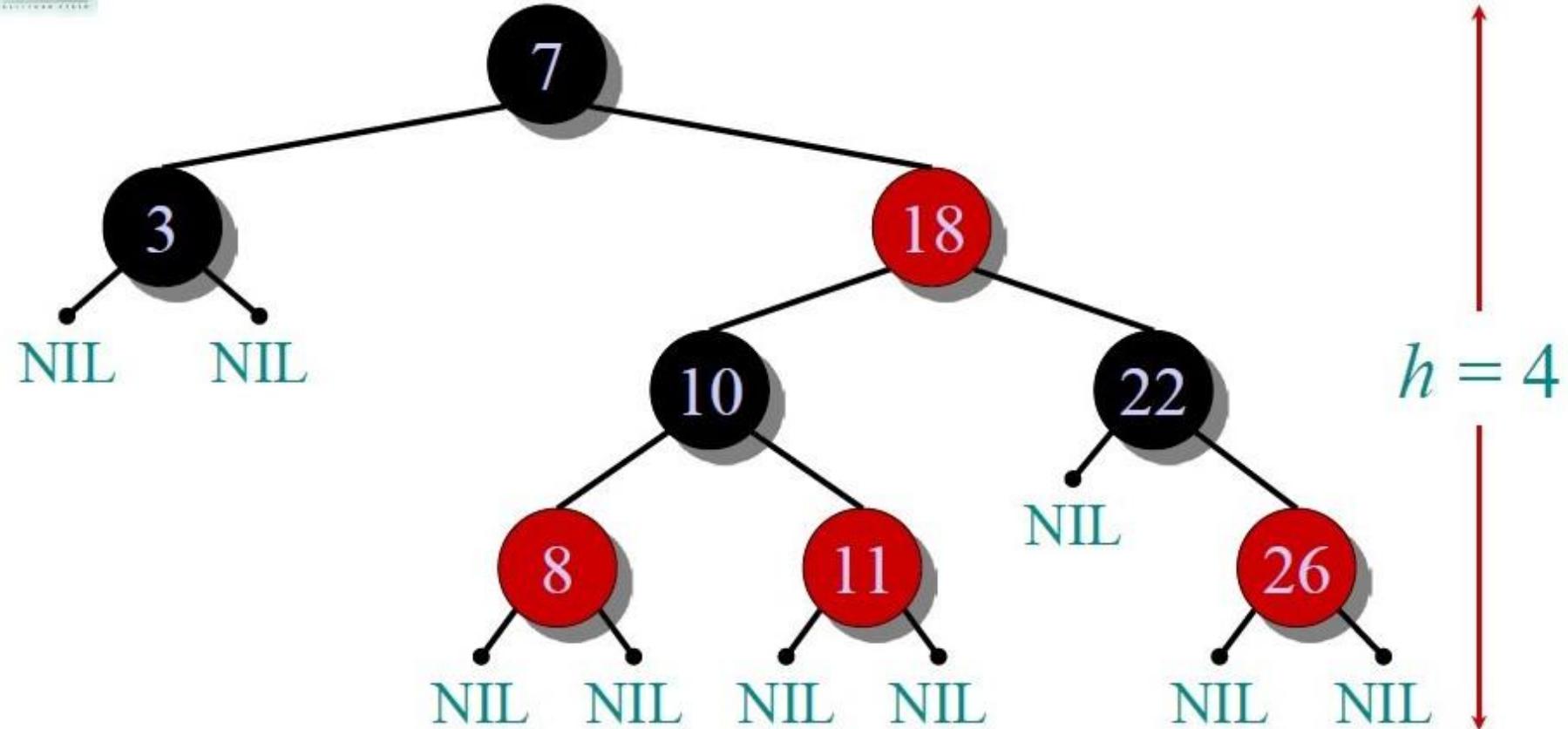
- Binary Search Tree (BST) is a good data structure for searching algorithm
- It supports
  - Search, find predecessor, find successor, find minimum, find maximum, insertion, deletion
- The performance of BST is related to its height  $h$ 
  - All the operations are  $O(h)$
- We want a balanced binary search tree
  - Height of the tree is  $O(\log n)$
- Red-Black Tree is one of the balanced binary search tree

# Properties of Red-Black Trees

- Every node is either red or black
- The root is black
- If a node is red, then both its children are black

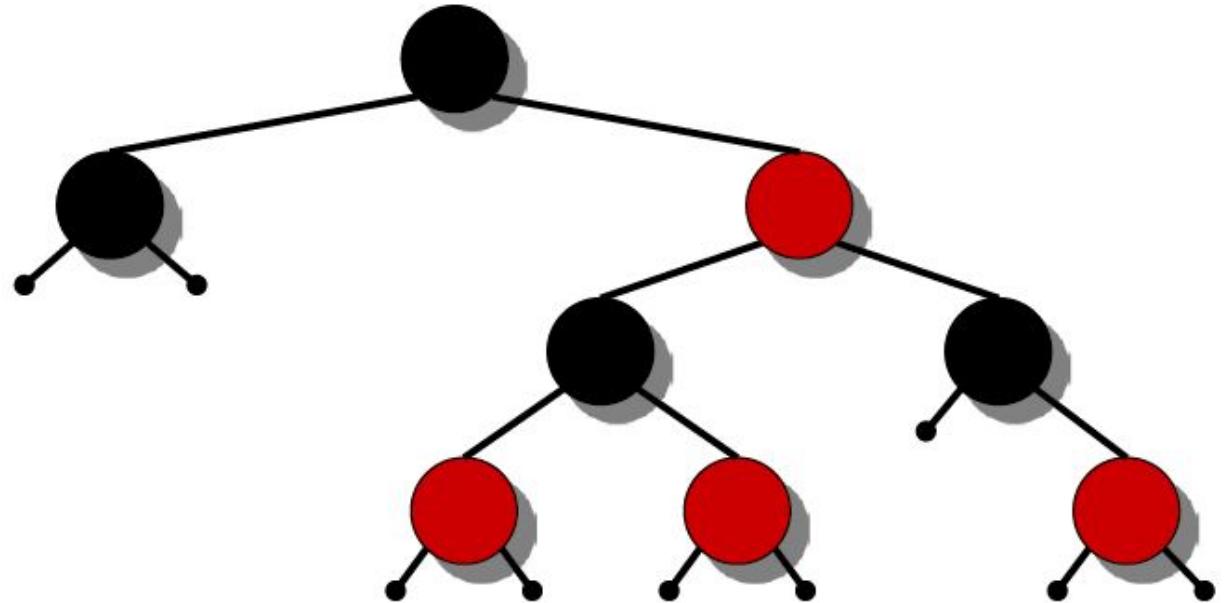


- For each node, all path from the node to descendant leaves contain the same number of black nodes
  - All path from the node have the same black height



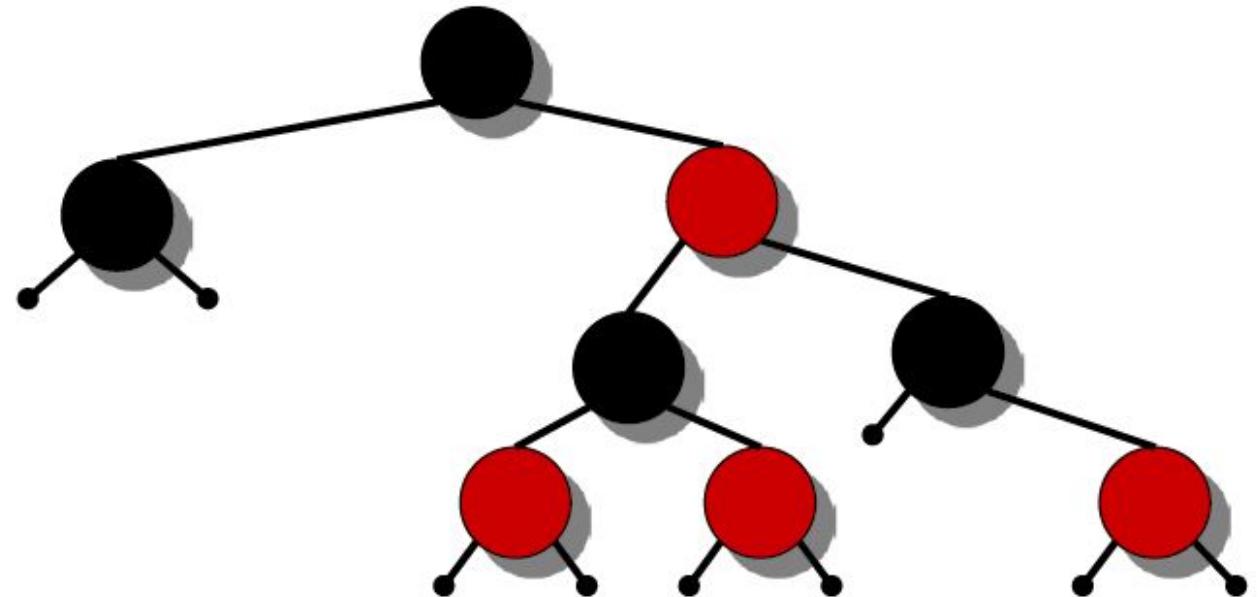
# Height of a red-black tree

- A red-black tree with  $n$  keys has height  $h \leq 2 \lg(n + 1)$ .
- **INTUITION:**
  - Merge red nodes into their black parents.



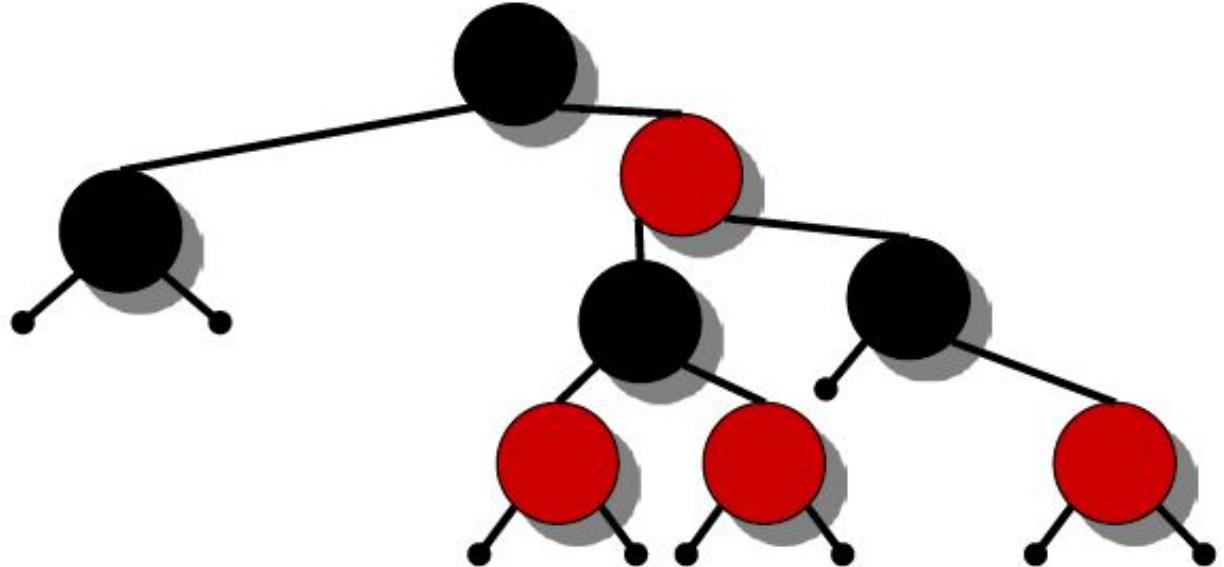
# Height of a red-black tree

- A red-black tree with  $n$  keys has height  $h \leq 2 \lg(n + 1)$ .
- **INTUITION:**
  - Merge red nodes into their black parents.



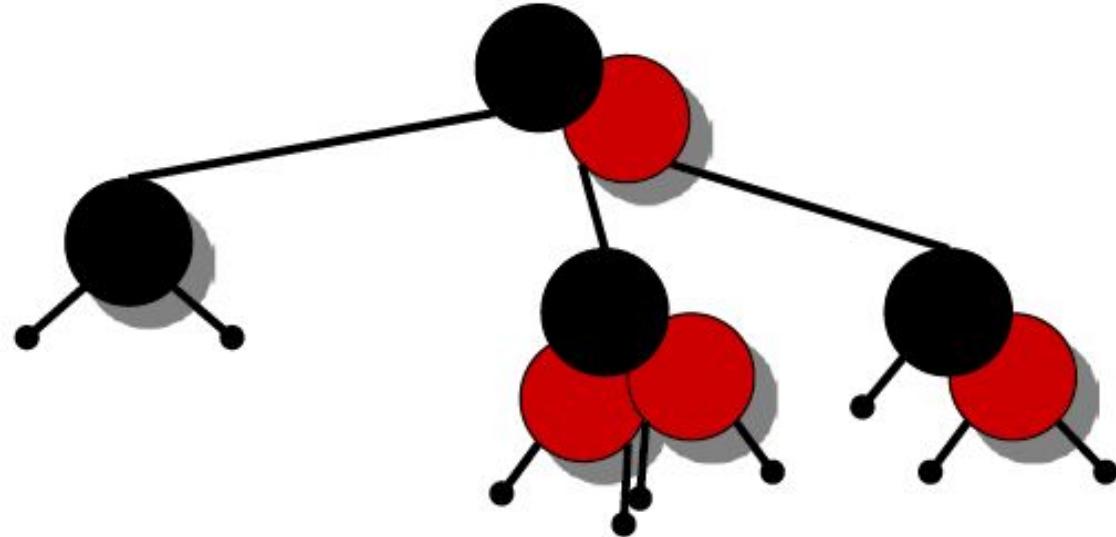
# Height of a red-black tree

- A red-black tree with  $n$  keys has height  $h \leq 2 \lg(n + 1)$ .
- **INTUITION:**
  - Merge red nodes into their black parents.



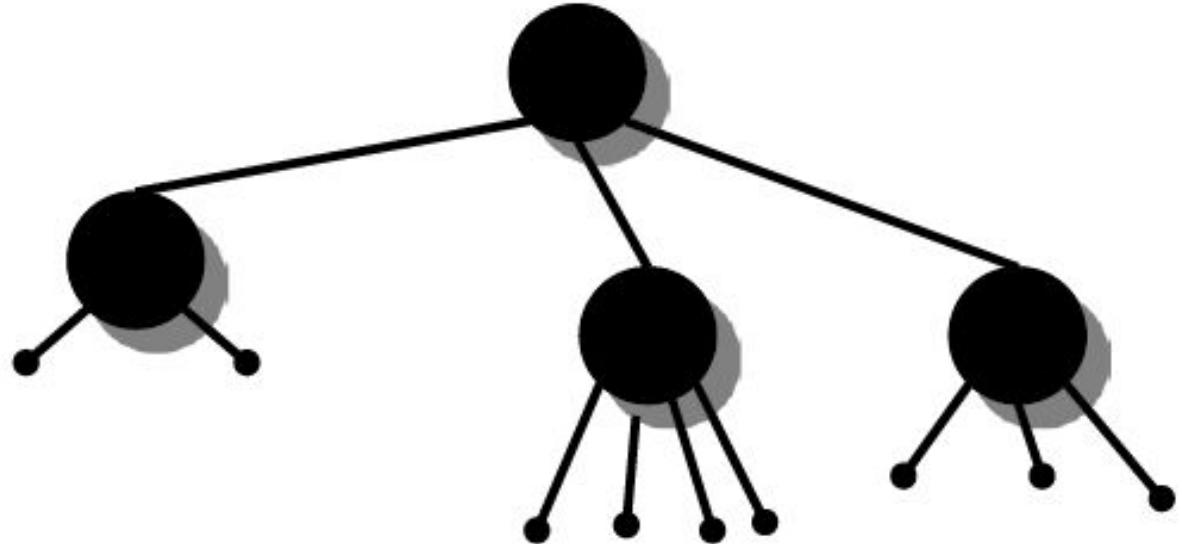
# Height of a red-black tree

- A red-black tree with  $n$  keys has height  $h \leq 2 \lg(n + 1)$ .
- **INTUITION:**
  - Merge red nodes into their black parents.



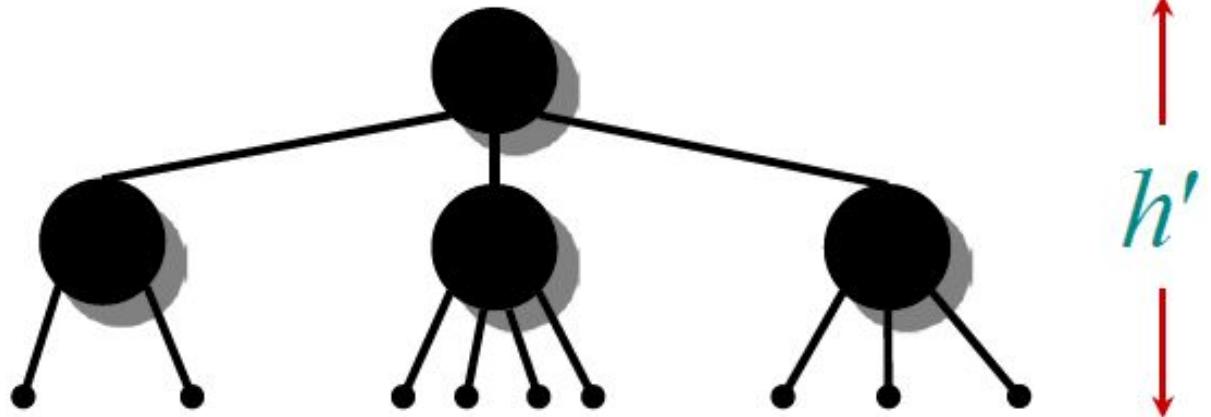
# Height of a red-black tree

- A red-black tree with  $n$  keys has height  $h \leq 2 \lg(n + 1)$ .
- **INTUITION:**
  - Merge red nodes into their black parents.



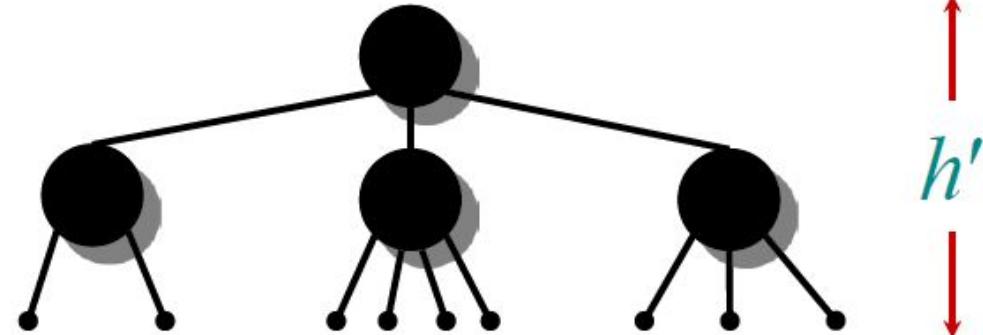
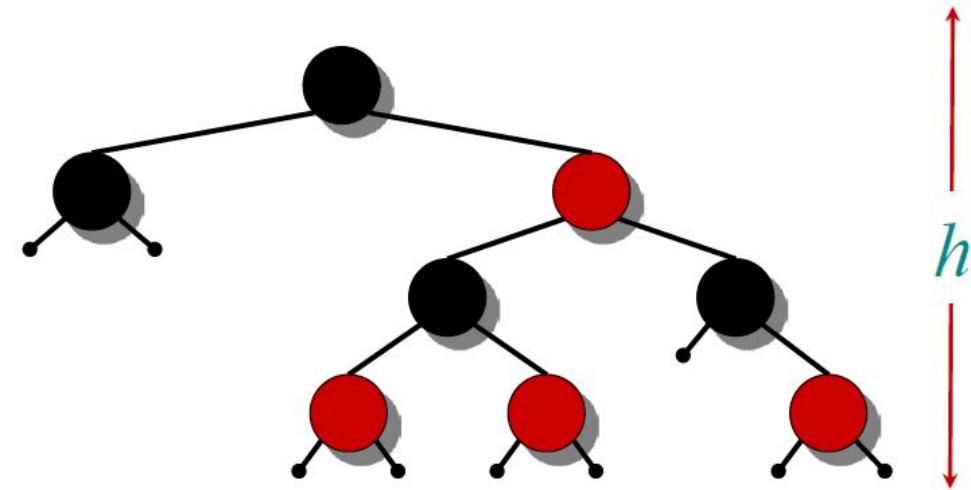
# Height of a red-black tree

- A red-black tree with  $n$  keys has height  $h \leq 2 \lg(n + 1)$ .
- **INTUITION:**
  - Merge red nodes into their black parents.
  - This process produces a tree in which each node has 2, 3, or 4 children.
  - The 2-3-4 tree has uniform depth  $h'$  of leaves.



# Height of a red-black tree

- We have  $h' \geq h/2$ , since at most half the leaves on any path are red.
- The number of leaves in each tree is  $n + 1 \Rightarrow n + 1 \geq 2h' \Rightarrow \lg(n + 1) \geq h' \geq h/2 \Rightarrow h \leq 2 \lg(n + 1)$ .
- The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in  $O(\lg n)$  time on a red-black tree with  $n$  nodes.

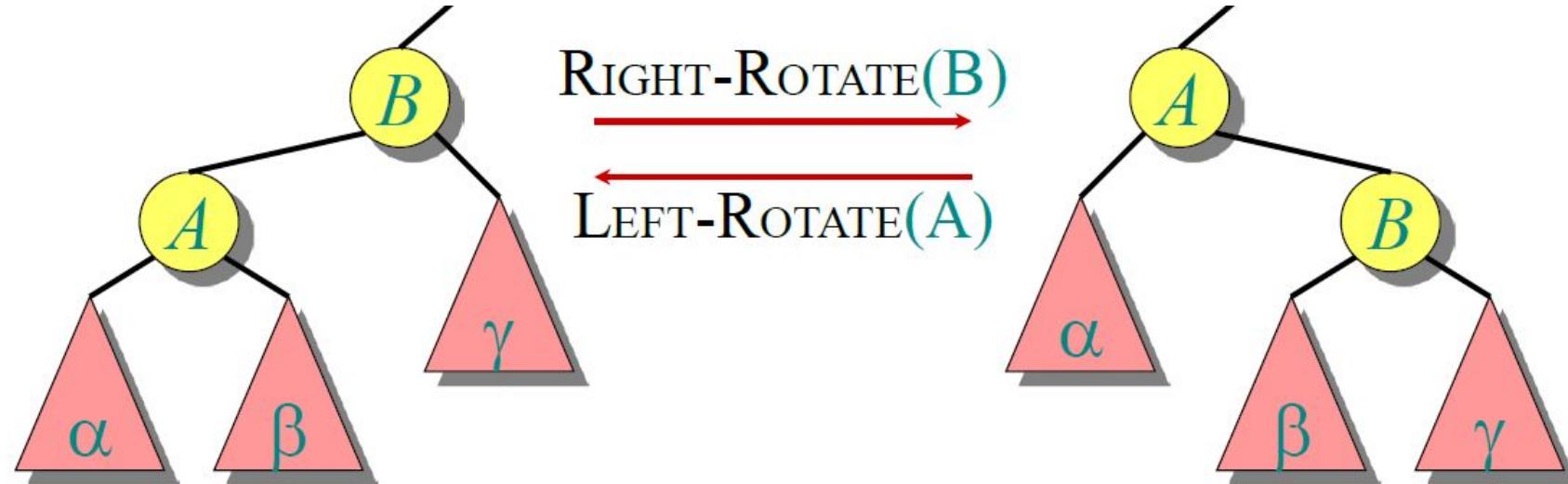


# Modifying operations

- The operations **INSERT** and **DELETE** cause modifications to the red-black tree:
  - the operations causes violation of the red-black properties
  - Colour of some nodes to be changed
  - restructuring the links of the tree via “*rotations*”.

# Rotations

- Rotations maintain the in-order ordering of keys:
  - $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c.$
- A rotation can be performed in  $O(1)$  time.



# Rotations

**Left Rotate ( $T, A$ ) pseudoocode**

$B=A.right$

$A.right=B.left$

*If*  $B.left \neq T.nil$

$B.left.p=A$

$B.p=A.p$

*If*  $A.p==T.nil$

$T.root=B$

*Elseif*  $A==A.p.left$

$A.p.left=A$

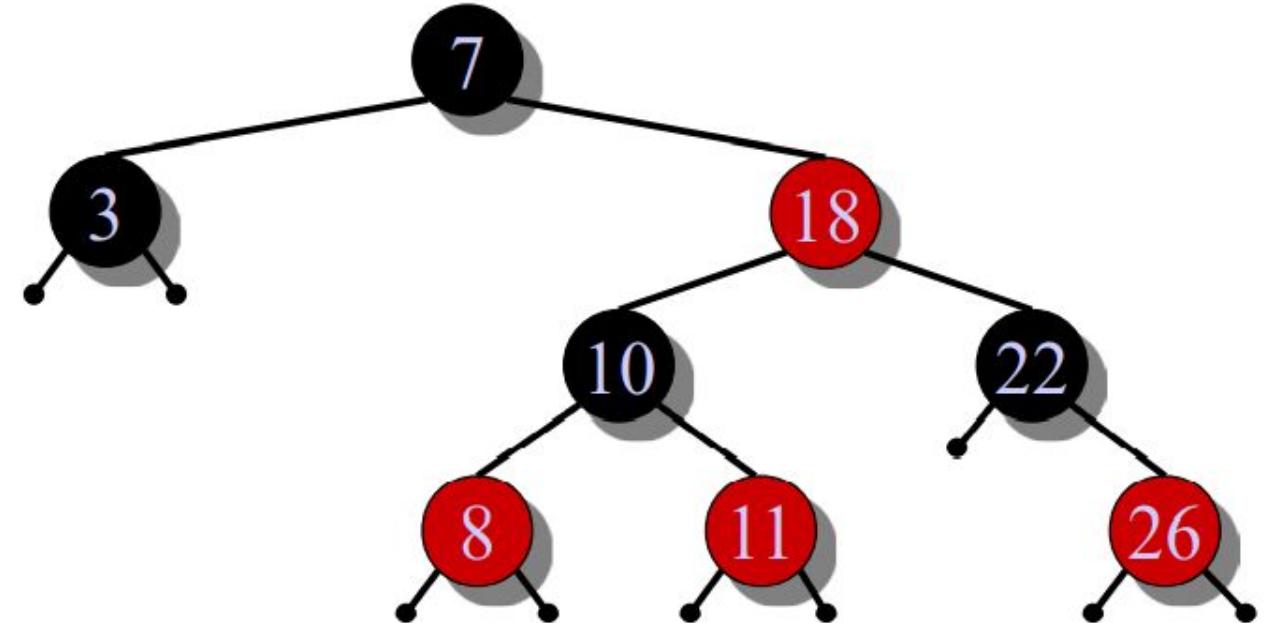
*Else*  $A.p.right=B$

$B.left=A$        *// A on B's Left*

$A.p=B$

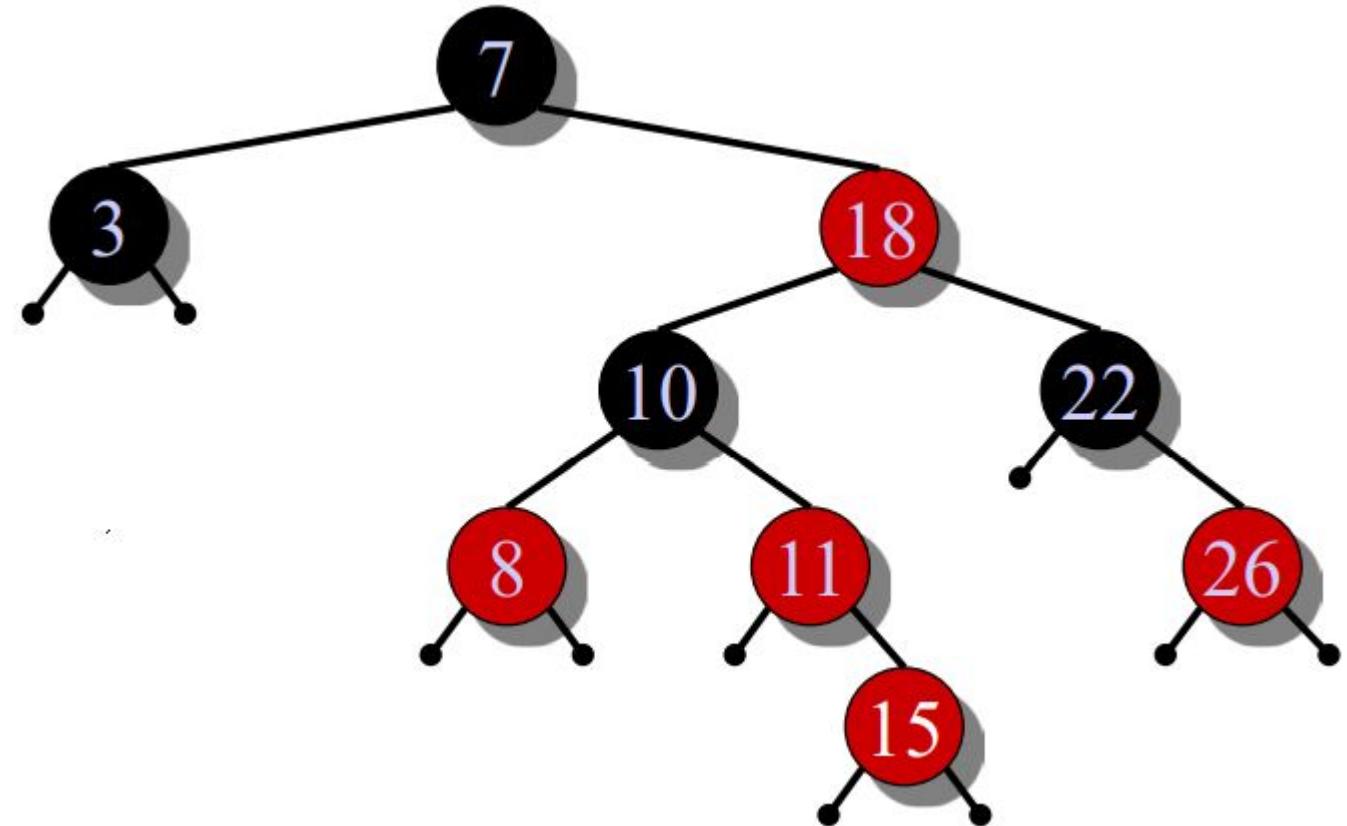
# Insertion into a red-black tree

- IDEA:
  - Insert x in tree. Color x red. Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring



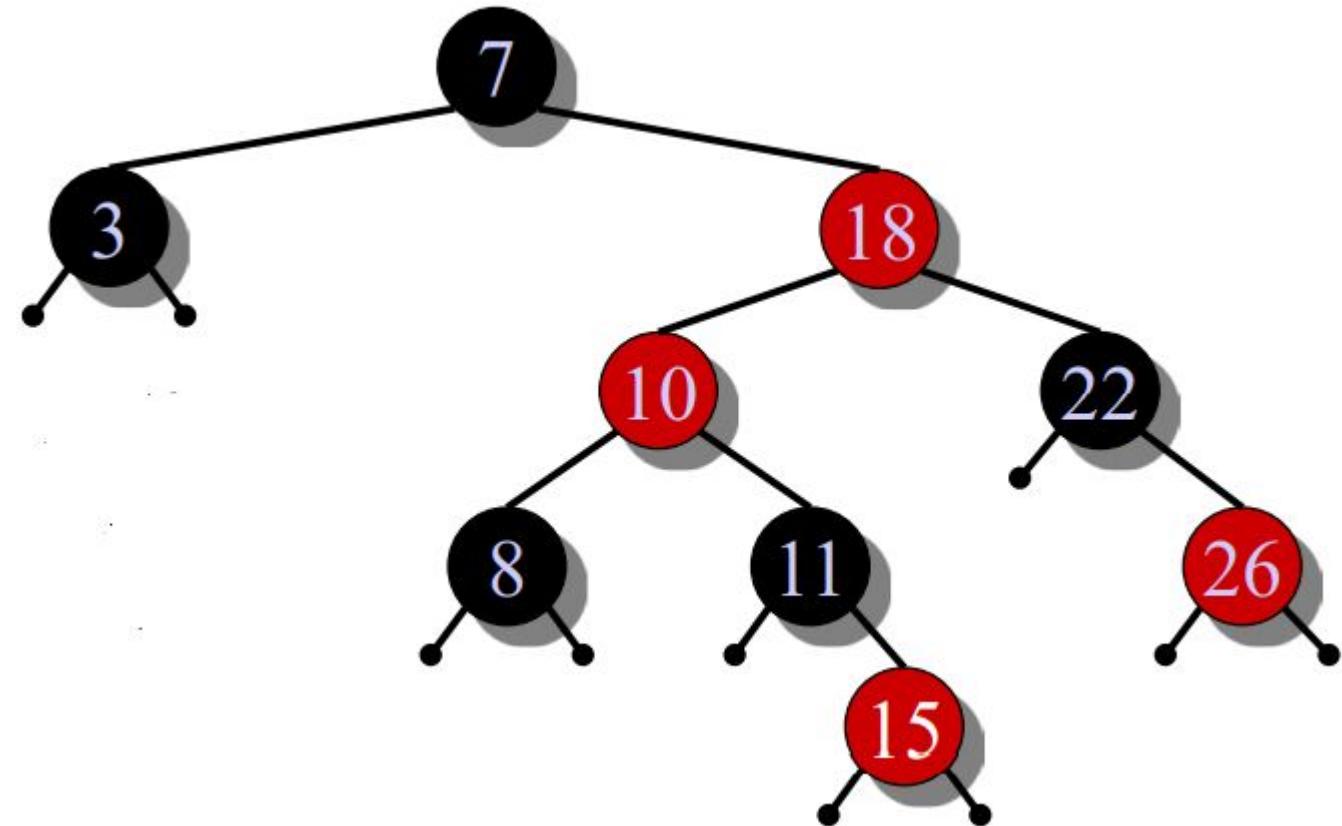
# Insertion into a red-black tree

- IDEA:
  - Insert x in tree. Color x red. Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring
- Example:
  - Insert x =15.



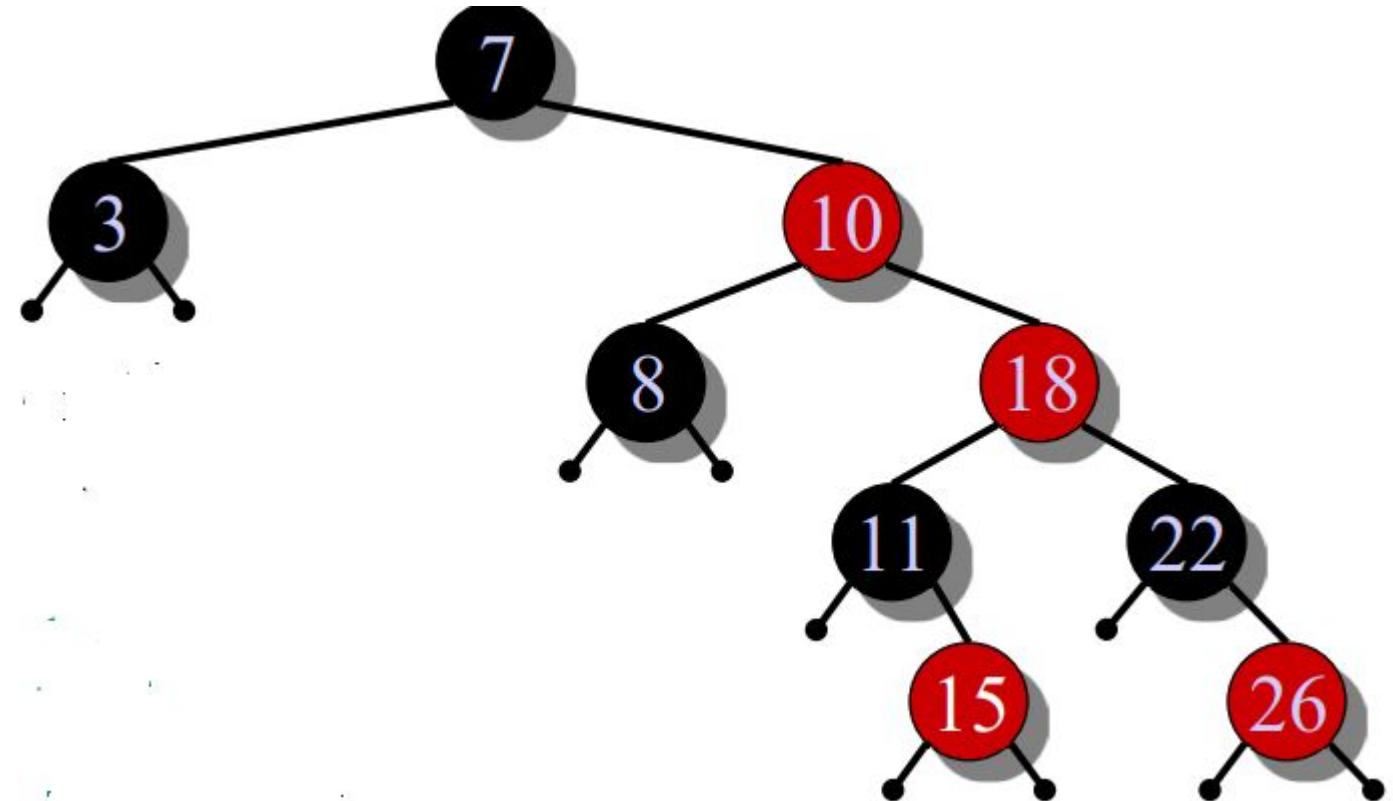
# Insertion into a red-black tree

- IDEA:
  - Insert x in tree. Color x red.
  - Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring
- Example:
  - Insert x =15.
  - Recolor, moving the violation up the tree.
  - RIGHT-ROTATE(18).



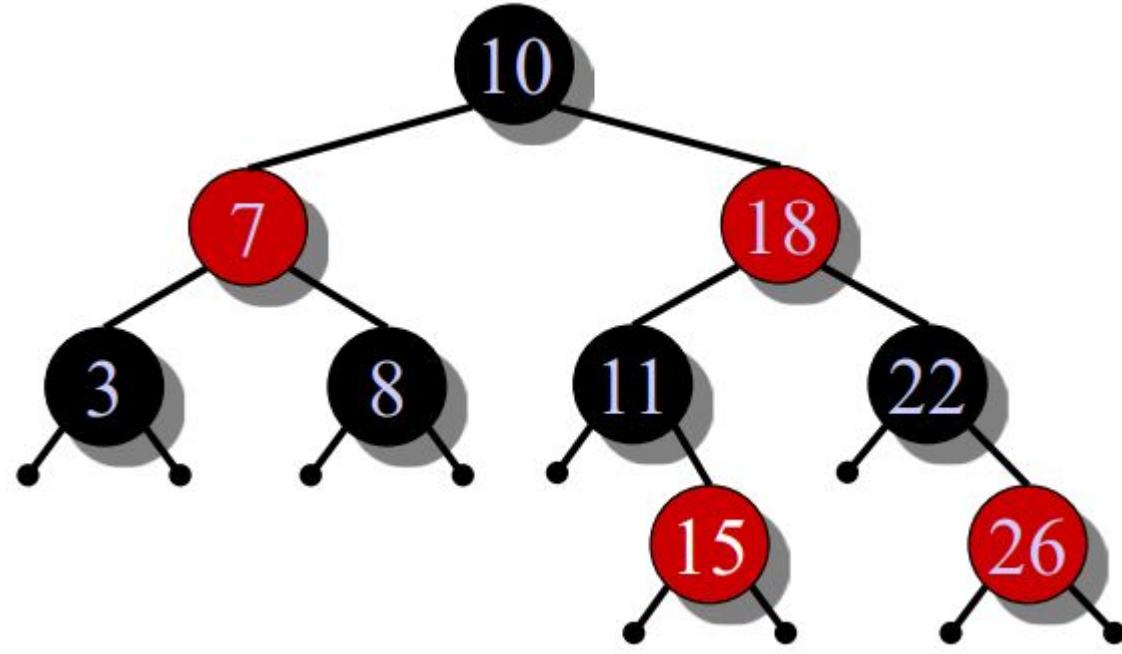
# Insertion into a red-black tree

- IDEA:
  - Insert x in tree. Color x red.
  - Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring
- Example:
  - Insert x =15.
  - Recolor, moving the violation up the tree.
  - RIGHT-ROTATE(18).
  - LEFT-ROTATE(7) and recolor.



# Insertion into a red-black tree

- IDEA:
  - Insert x in tree. Color x red.
  - Only redblack property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring
- Example:
  - Insert x =15.
  - Recolor, moving the violation up the tree.
  - RIGHT-ROTATE(18).
  - LEFT-ROTATE(7) and recolor.



# Insertion into a red-black tree- Pseudocode

RB-INSERT( $T, x$ )

  TREE-INSERT( $T, x$ )

  color[ $x$ ]  $\leftarrow$  RED                                  //only RB property 3 can be violated

  while  $x \neq \text{root}[T]$  and color[ $p[x]$ ] = RED

    do if  $p[x] = \text{left}[p[p[x]]]$

      then  $y \leftarrow \text{right}[p[p[x]]]$                           //  $y$  = aunt/uncle of  $x$

      if color[ $y$ ] = RED

        then ⟨Case 1⟩

        else if  $x = \text{right}[p[x]]$

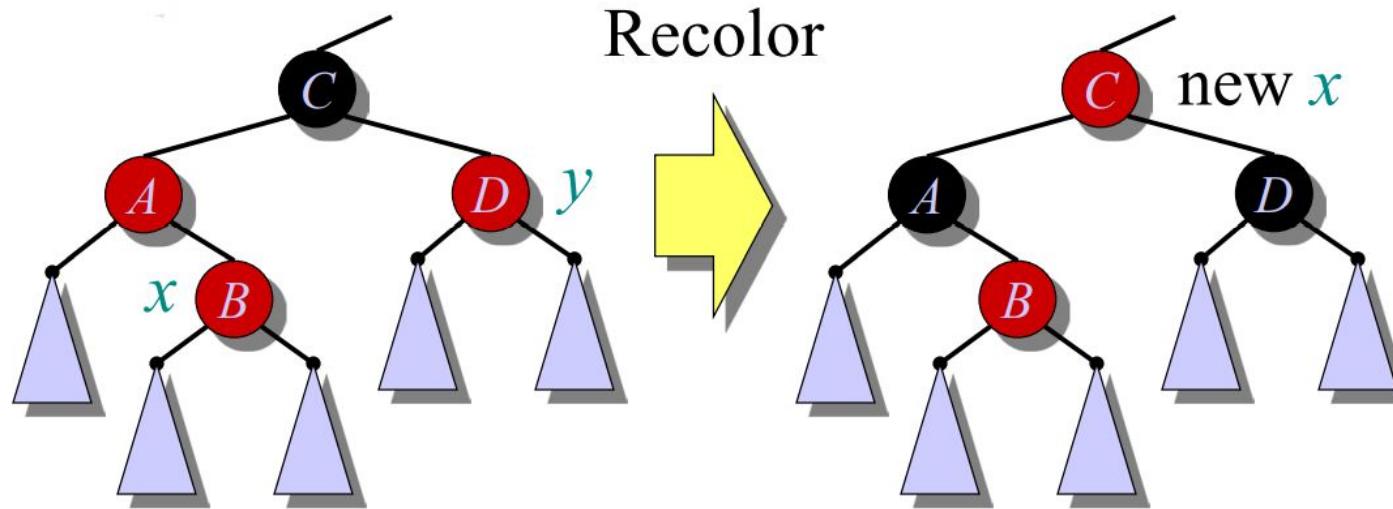
          then ⟨Case 2⟩                                  // Case 2 falls into Case 3

          ⟨Case 3⟩

        else ⟨“then” clause with “left” and “right” swapped⟩

  color[ $\text{root}[T]$ ]  $\leftarrow$  BLACK

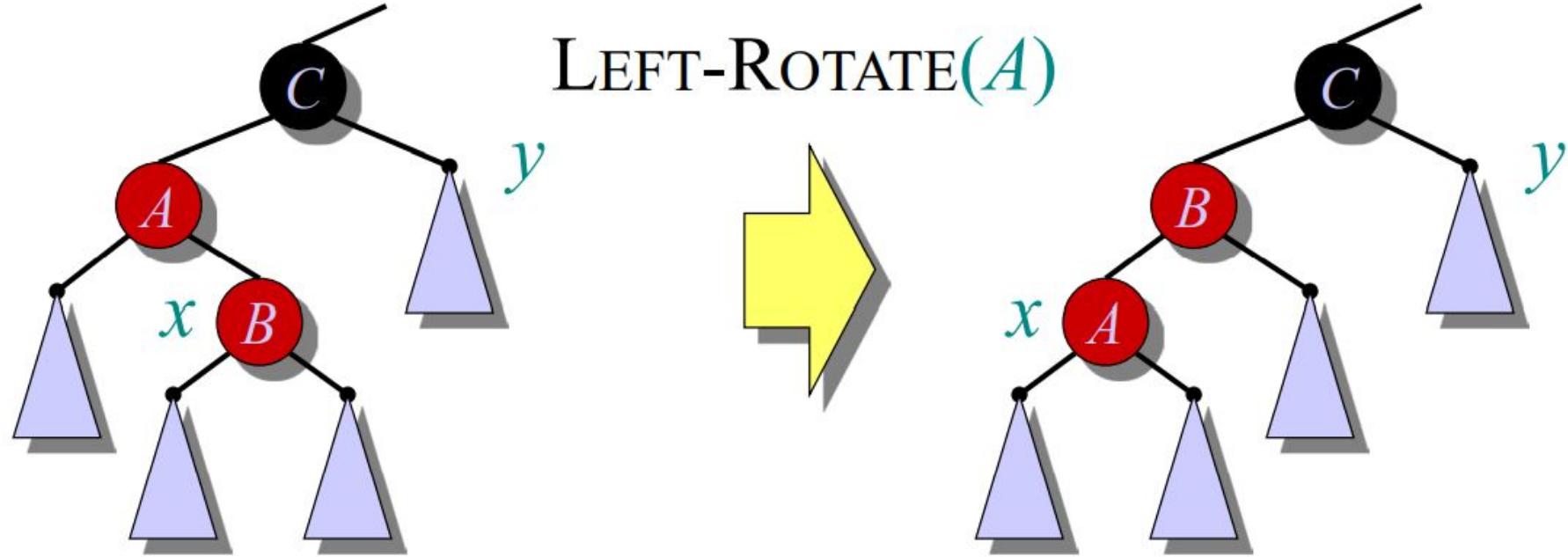
# Insertion into a red-black tree- Case-1



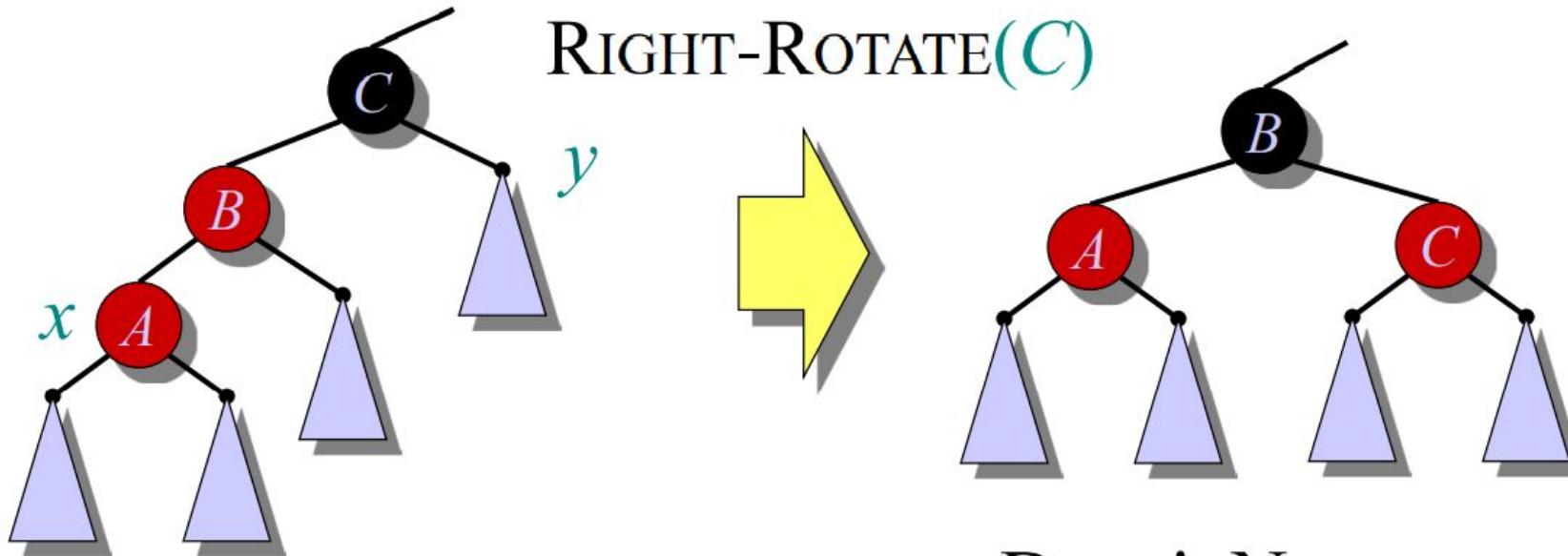
(Or, children of  
 $A$  are swapped.)

Push  $C$ 's black onto  
 $A$  and  $D$ , and recurse,  
since  $C$ 's parent may  
be red.

# Insertion into a red-black tree- Case-2



# Insertion into a red-black tree- Case-3



Done! No more violations of RB property 3 are possible.

# Analysis

- Go up the tree performing Case 1, which only recolors nodes.
- If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.
- Running time:  $O(\lg n)$  with  $O(1)$  rotations.
- RB-DELETE — same asymptotic running time and number of rotations as RB-INSERT

# Red-Black Tree- Deletion

Prepared by  
Dr Annushree Bablani

# Red Black Tree- Deletion

- To perform operation, we first execute the deletion algorithm for binary search trees
- Thus, the node which is deleted is the parent of an external node.
- If node is red, it won't violate any property
- If node is a leaf, it won't violate any property
- Otherwise, if node is black and has a child, it will violate property 2, 3, and 4
- For property 2, set the color of root to black after deletion

# Red Black Tree- Deletion

- To fix property 3 and 4:
  - From now on, lets call the deleted node to be z
  - If z's child x (which is the replacing node) is red, set x to black. Done!
  - If x is black, add another black to x, so that x will be a doubly black node, and property 3 and 4 are fixed. But property 1 is violated

# Red Black Tree- Deletion

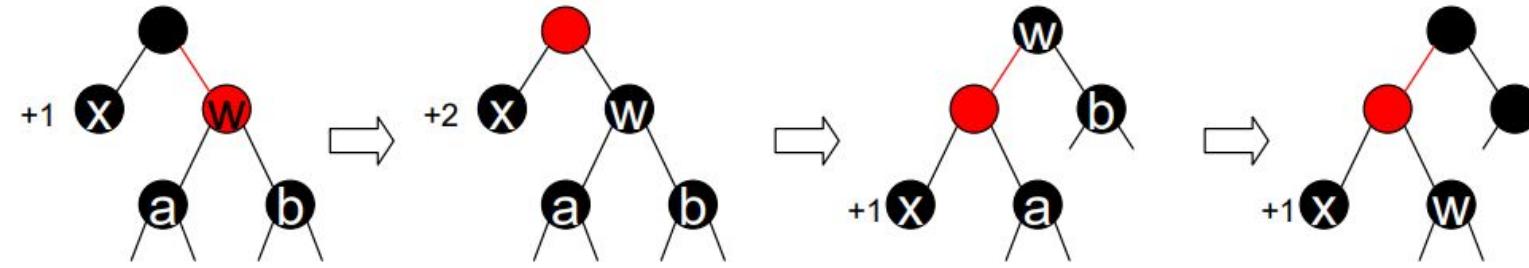
- To fix property 1, we will consider if
  - $x$  is a left child or right child
  - The color of  $x$ 's sibling  $w$  is red or black
  - The colors of  $w$ 's children
- We consider  $x$  is a left child first, the other case can be done by symmetric operation

# Red Black Tree- Deletion

- There are 4 cases:
  - Case 1: w is red
  - Case 2: w is black, both w's children are black
  - Case 3: w is black, w's left child is red, w's right child is black
  - Case 4: w is black, w's right child is red

# Red Black Tree - Deletion

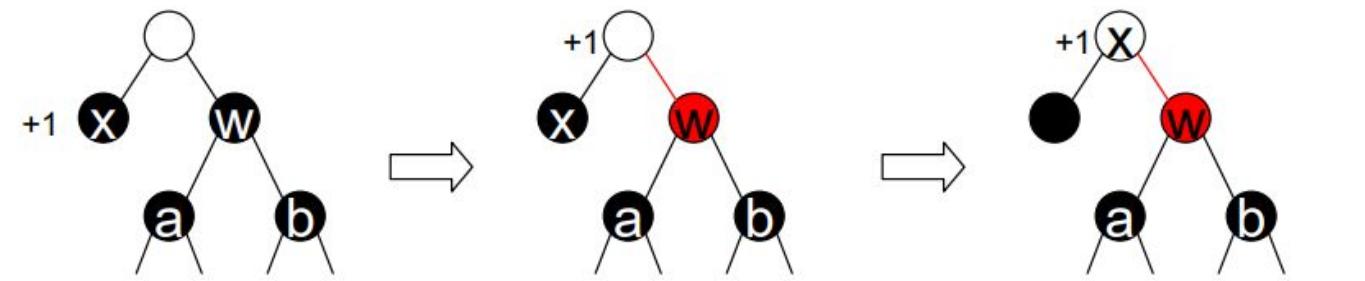
- Case 1: w is red



Case 2, 3, 4

# Red Black Tree - Deletion

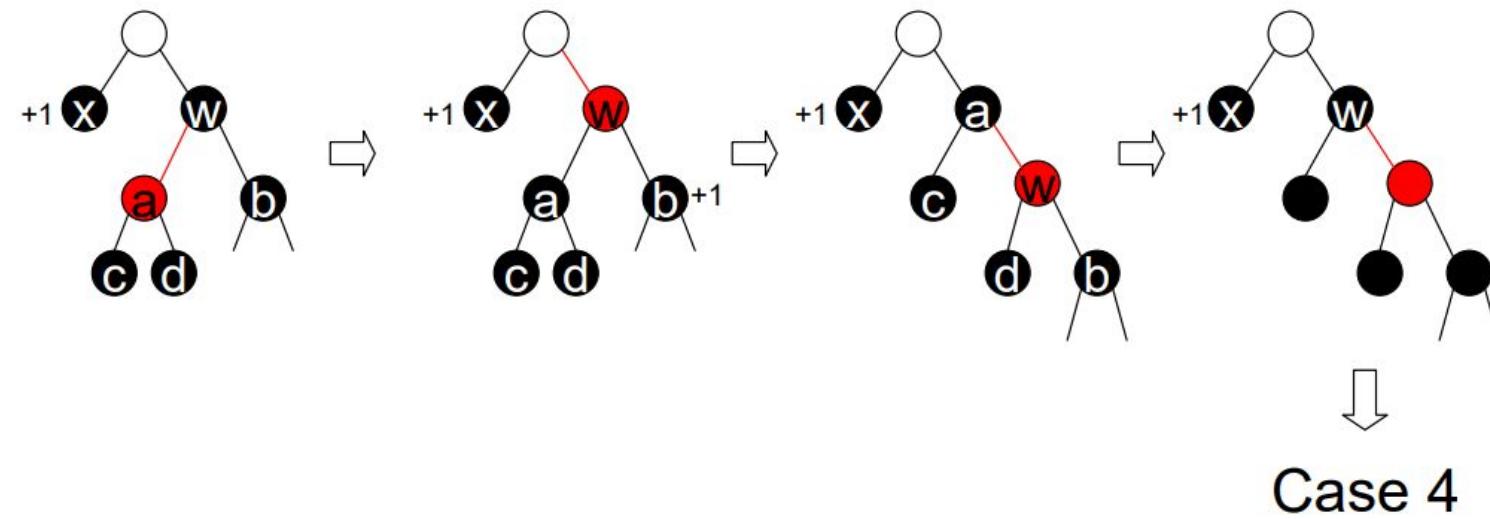
- Case 2: w is black, both w's children are black



Recursively delete x

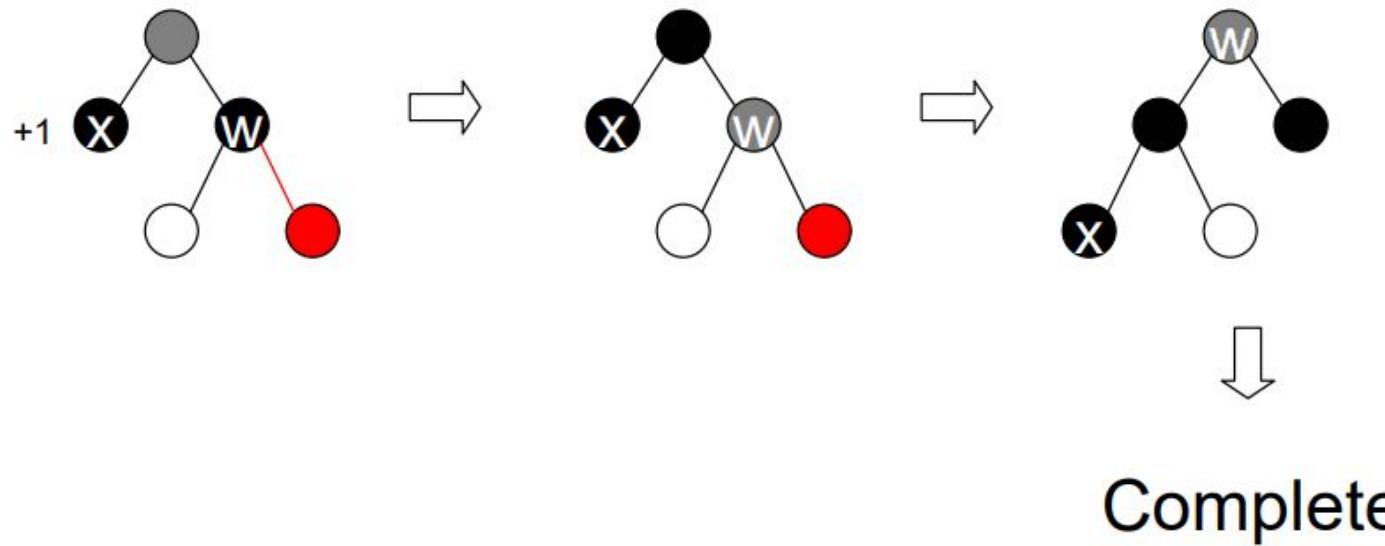
# Red Black Tree - Deletion

- Case 3: w is black, w's left child is red, w's right child is black



# Red Black Tree - Deletion

- Case 4: w is black, w's right child is red



# Red-Black Tree Deletion - Pseudocode

- RB-Delete( $T, z$ )
  1. if  $z \rightarrow \text{left} = \text{null}$  or  $z \rightarrow \text{right} = \text{null}$
  2.     then  $y \leftarrow z$
  3. else  $y \leftarrow \text{TREE-SUCCESSOR}(z)$
  4. if  $y \rightarrow \text{left} \neq \text{null}$
  5.     then  $x \leftarrow y \rightarrow \text{left}$
  6. else  $x \leftarrow y \rightarrow \text{right}$
  7.  $x \rightarrow p \leftarrow y \rightarrow p$
  8. if  $y \rightarrow p = \text{null}$
  9.     then  $T \rightarrow \text{root} \leftarrow x$
  10. else if  $y = y \rightarrow p \rightarrow \text{left}$
  11.     then  $y \rightarrow p \rightarrow \text{left} \leftarrow x$
  12. else  $y \rightarrow p \rightarrow \text{right} \leftarrow x$
  13. if  $y \neq z$
  14.     then  $z \rightarrow \text{key} \leftarrow y \rightarrow \text{key}$
  15. copy  $y$ 's data into  $z$
  16. if  $y \rightarrow \text{color} = \text{BLACK}$
  17.     then **RB-DELETE-FIXUP( $T, x$ )**
  18. return  $y$

# Red-Black Tree Deletion - Pseudocode

- **RB-DELETE-FIXUP( $T, x$ )**
1.    while  $x \neq T \rightarrow \text{root}$  and  $x \rightarrow \text{color} = \text{BLACK}$
  2.       do if  $x = x \rightarrow p \rightarrow \text{left}$
  3.          then  $w \leftarrow x \rightarrow p \rightarrow \text{right}$
  4.          if  $w \rightarrow \text{color} = \text{RED}$
  5.             then  $w \rightarrow \text{color} \leftarrow \text{BLACK}$  Case 1
  6.              $x \rightarrow p \rightarrow \text{color} \leftarrow \text{RED}$  Case 1
  7.             LEFT-ROTATE( $T, x \rightarrow p$ ) Case 1
  8.              $w \leftarrow x \rightarrow p \rightarrow \text{right}$  Case 1
  9.             if  $w \rightarrow \text{left} \rightarrow \text{color} = \text{BLACK}$  and  
 $w \rightarrow \text{right} \rightarrow \text{color} = \text{BLACK}$
  10.             then  $w \rightarrow \text{color} \leftarrow \text{RED}$  Case 2
  11.              $x \leftarrow x \rightarrow p$  Case 2
  12.             else if  $w \rightarrow \text{right} \rightarrow \text{color} = \text{BLACK}$
  13.                then  $w \rightarrow \text{left} \rightarrow \text{color} \leftarrow \text{BLACK}$  Case 3
  14.                 $w \rightarrow \text{color} \leftarrow \text{RED}$  Case 3
  15.                RIGHT-ROTATE( $T, w$ ) Case 3
  16.                 $w \leftarrow x \rightarrow p \rightarrow \text{right}$  Case 3
  17.                 $w \rightarrow \text{color} \leftarrow x \rightarrow p \rightarrow \text{color}$  Case 4
  18.                 $x \rightarrow p \rightarrow \text{color} \leftarrow \text{BLACK}$  Case 4
  19.                 $w \rightarrow \text{right} \rightarrow \text{color} \leftarrow \text{BLACK}$  Case 4
  20.                LEFT-ROTATE( $T, x \rightarrow p$ ) Case 4
  21.                 $x \leftarrow T \rightarrow \text{root}$  Case 4
  22.                else (same as then clause with "right" and  
"left" exchanged)
  23.                 $x \rightarrow \text{color} \leftarrow \text{BLACK}$

# Red Black Tree – Deletion Summary

In all cases, except 2,  
deletion can be completed by  
a simple rotation/ recoloring.

In case 2, the height of the  
subtree reduces and so we  
need to proceed up the tree

- If we proceed up the tree,  
we only need to  
recolor/rotate.

Complexity-  $O(\log n)$

# Red-Black Trees to 2-4 Trees

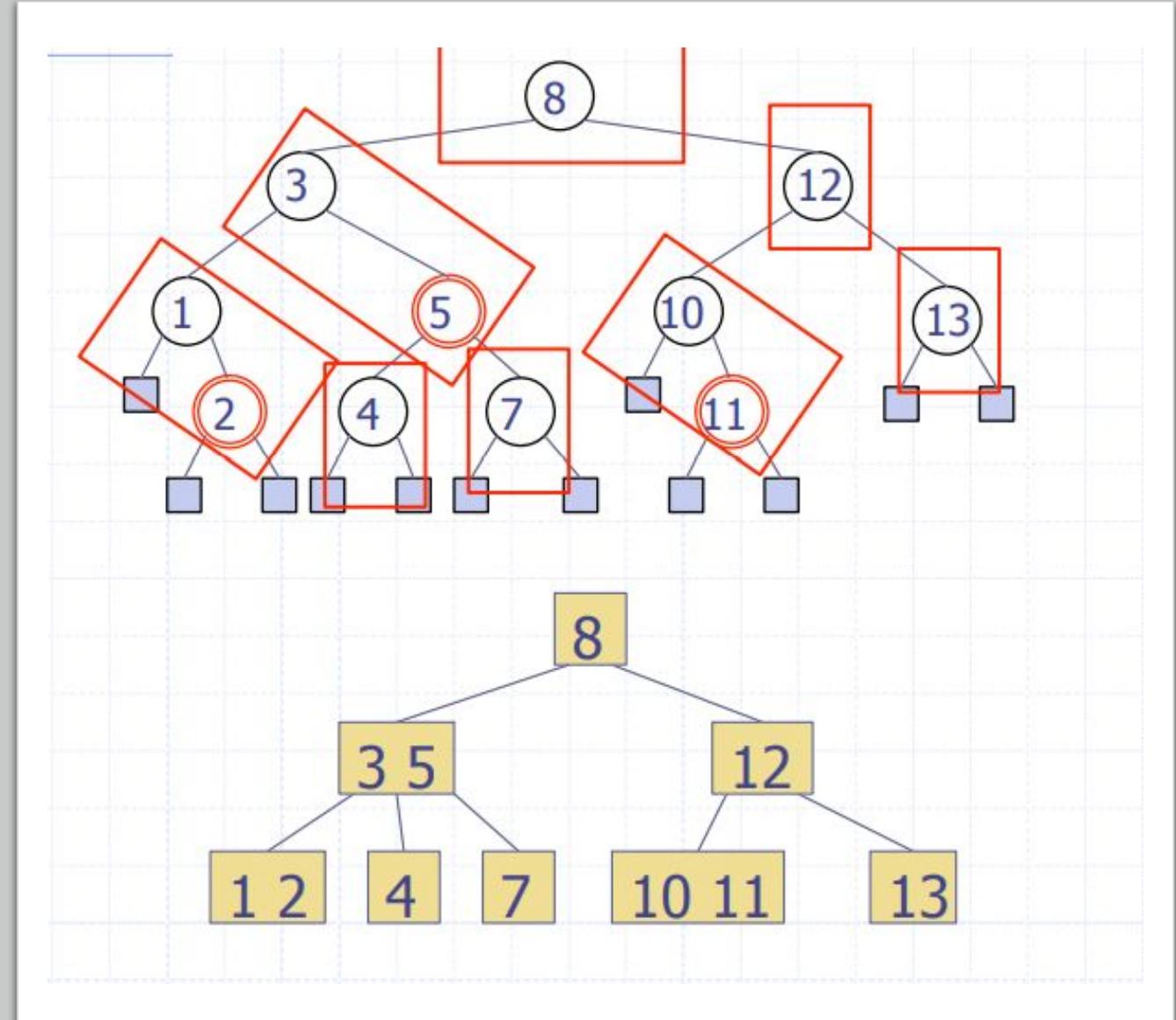
Any red-black tree can be converted into a 2-4 tree

Take a black node and its red children (at most 2) and combine them into one node of a 2-4 tree.

Each node thus formed has at least 1 and at most 3 keys

Since black depth of all external nodes is the same, in the resulting 2-4 tree all the external nodes will be at the same level.

# Red-Black Tree to 2-4 Tree - Example



# 2-4 Trees to Red-Black Trees

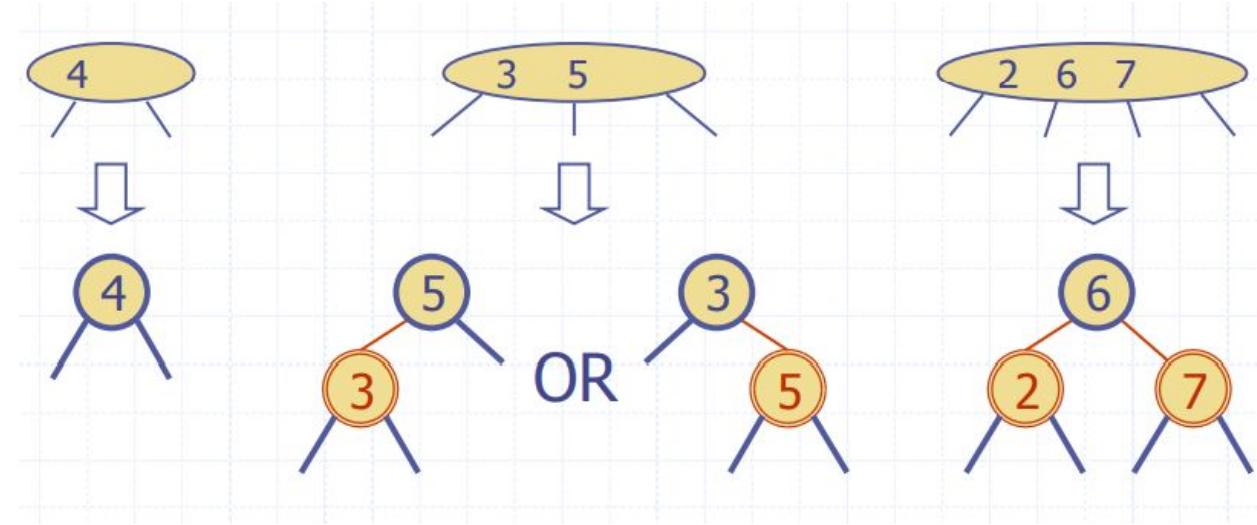
Any 2-4 tree can be converted into a red-black tree

We replace a node of the 2-4 tree with one black node and 0/1/2 red nodes which are children of the black node.

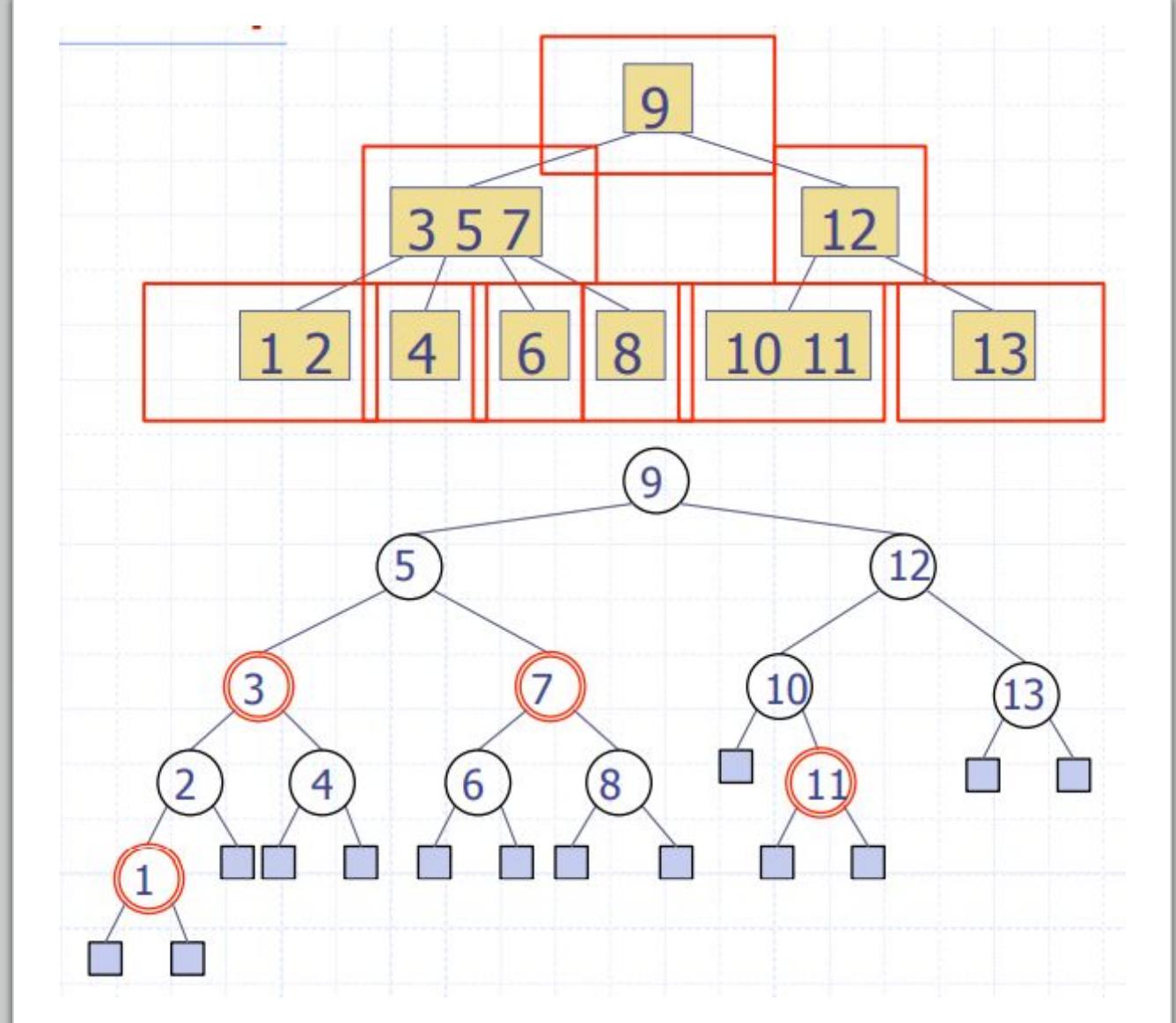
The height of 2-4 tree is the black depth of the red-black tree created.

Every red node has a black child.

# 2-4 Tree to Red-Black Trees



# 2-4 Trees to Red-Black Trees - Example



# **B+ Trees**

Prepared by:

Dr. Annushree Bablani

# B+ Trees

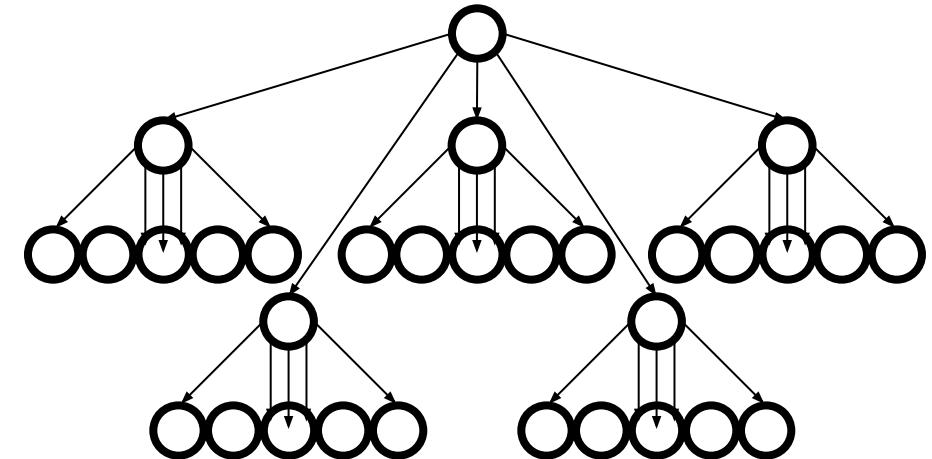
- What are B+ Trees used for
- What is a B Tree
- What is a B+ Tree
- Searching
- Insertion
- Deletion

# What are B+ Trees Used For?

- When we store data in a table in a DBMS we want
  - Fast lookup by primary key
    - Just this – hashtable  $O(c)$
  - Ability to add/remove records on the fly
    - Dynamic tree **on disk**
  - Sequential access to records (physically sorted by primary key on disk)
    - Tree structured keys (hierarchical index for searching)
    - Records all at leaves in sorted order

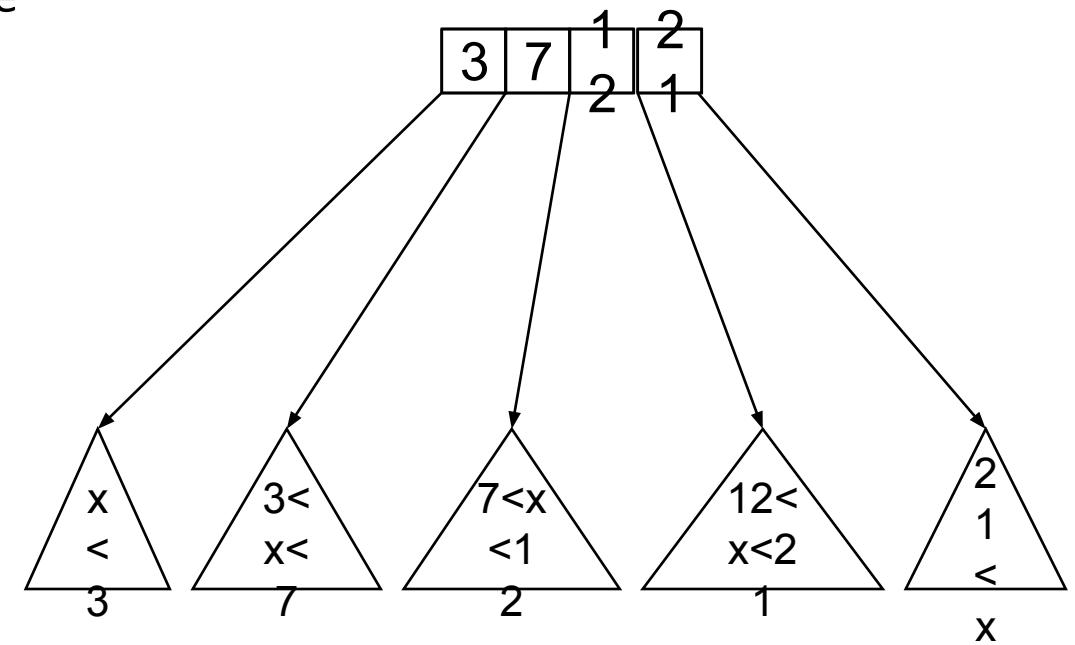
# M-ary Search Tree- Revision

- Maximum branching factor of  $M$
- Complete tree has depth =  $\log_M N$
- Each internal node in a complete tree has
- $M - 1$  keys
- Binary search tree is a B Tree where  $M$  is 2



# B Trees

- B-Trees are specialized  $M$ -ary search trees
- Each node has many keys
  - subtree between two keys  $x$  and  $y$  contains values  $v$  such that  $x \leq v < y$
  - binary search within a node to find correct subtree
- Each node takes one
  - full  $\{page, block, line\}$  of memory (disk)



# B-Tree Properties

- Properties
  - maximum branching factor of  $M$
  - the root has between 2 and  $M$  children *or* at most  $M-1$  keys
  - All other nodes have between  $\lceil M/2 \rceil$  and  $M$  records
    - Keys+data
- Result
  - tree is  $O(\log M)$  deep
  - all operations run in  $O(\log M)$  time
  - operations pull in about  $M$  items at a time

# What is a B+ Tree?

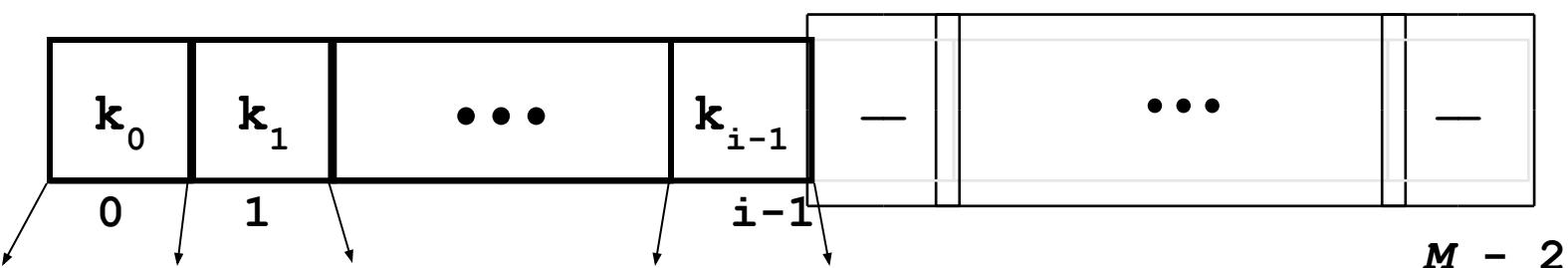
- A variation of B trees in which
  - internal nodes contain only search keys (no data)
  - Leaf nodes contain pointers to data records
  - Data records are in sorted order by the search key
  - All leaves are at the same depth

## Definition of a B+Tree

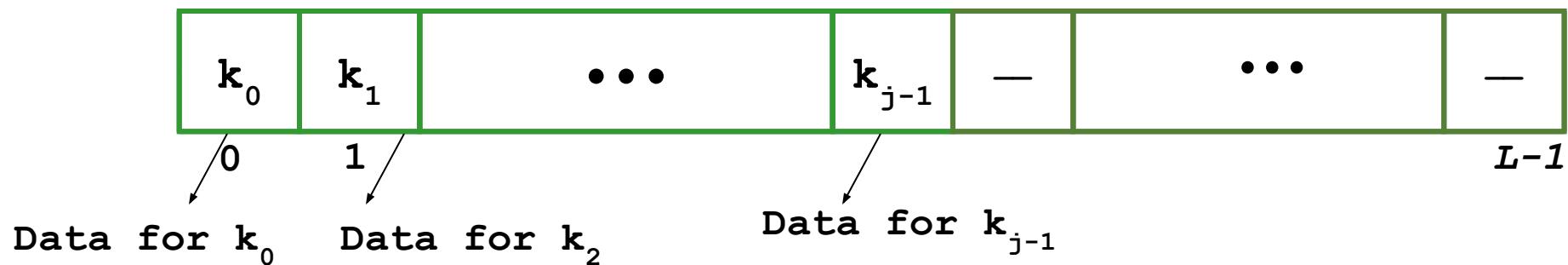
A B+ tree is a balanced tree in which every path from the root of the tree to a leaf is of the same length, and each non-leaf node of the tree has between  $[M/2]$  and  $[M]$  children, where  $n$  is fixed for a particular tree.

# B+ Tree Nodes

- Internal node
  - Pointer (Key, NodePointer)\*M-1 in each node
  - First  $i$  keys are currently in use

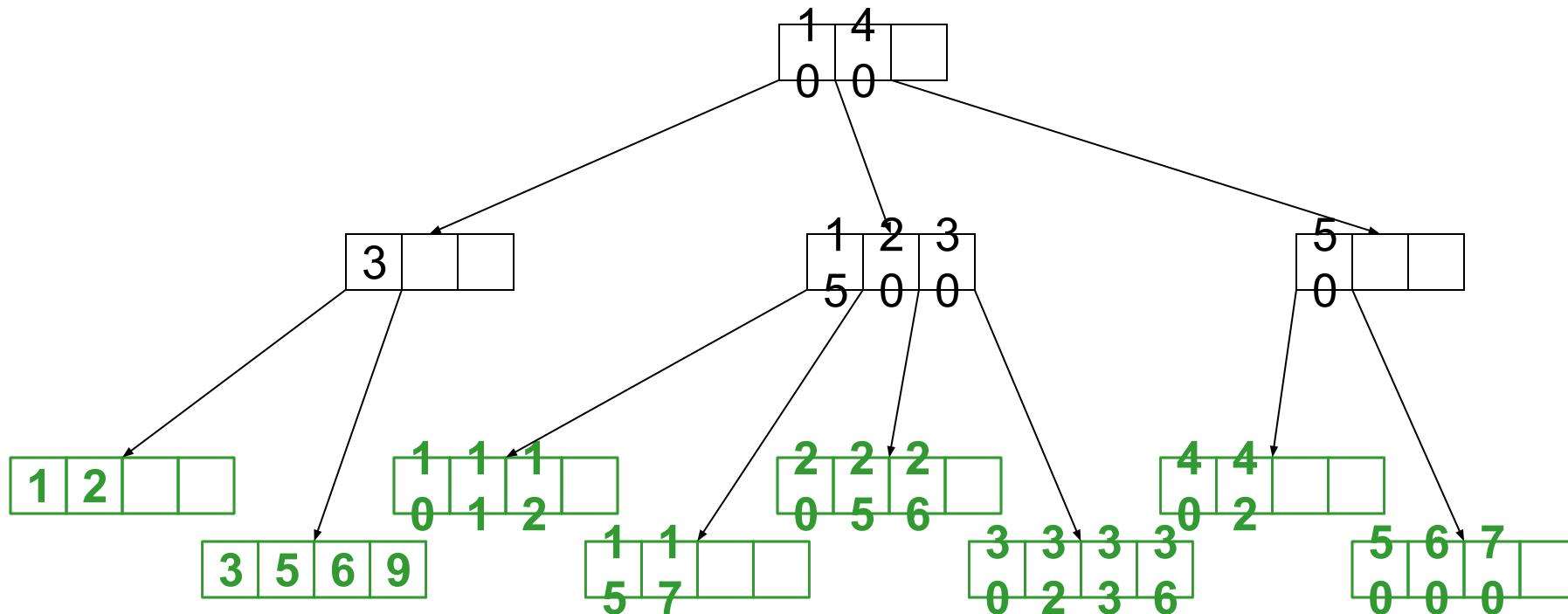


- Leaf
  - (Key, DataPointer)\* L in each node
  - first  $j$  Keys currently in use



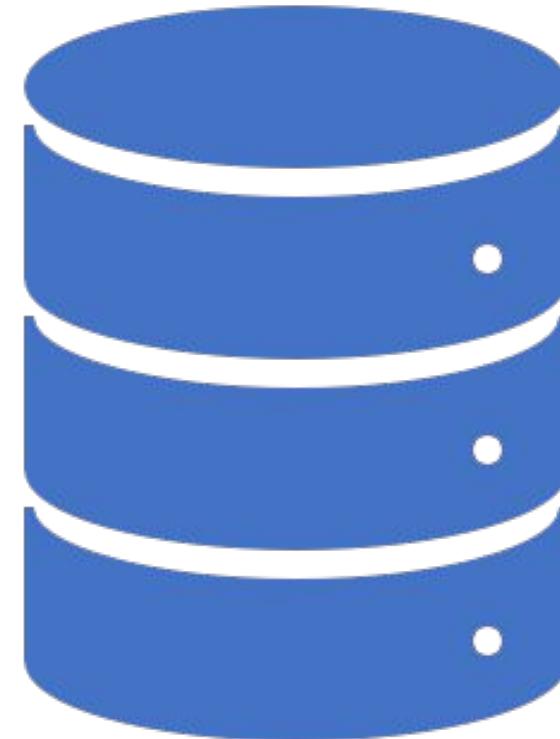
Example:

B+ Tree with  $M = 4$ . Often, leaf nodes linked together



# Advantages of B+ tree usage for databases

- keeps keys in sorted order for sequential traversing
- uses a hierarchical index to minimize the number of disk reads
- uses partially full blocks to speed insertions and deletions
- keeps the index balanced with a recursive algorithm
- In addition, a B+ tree minimizes waste by making sure the interior nodes are at least half full. A B+ tree can handle an arbitrary number of insertions and deletions.

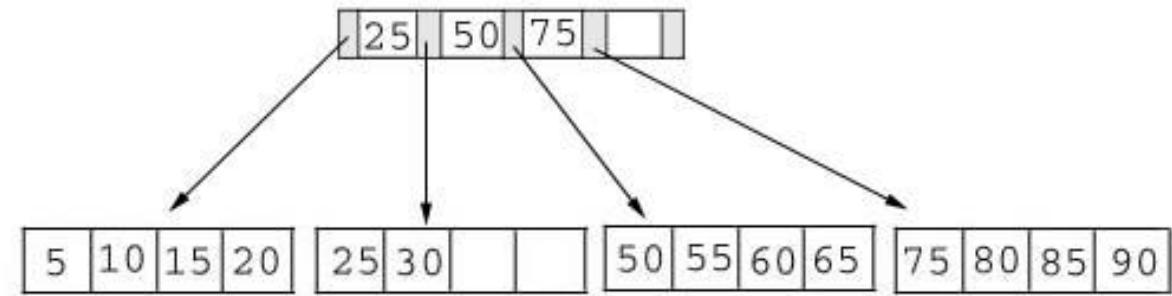


# Searching

- Just compare the key value with the data in the tree, then return the result.

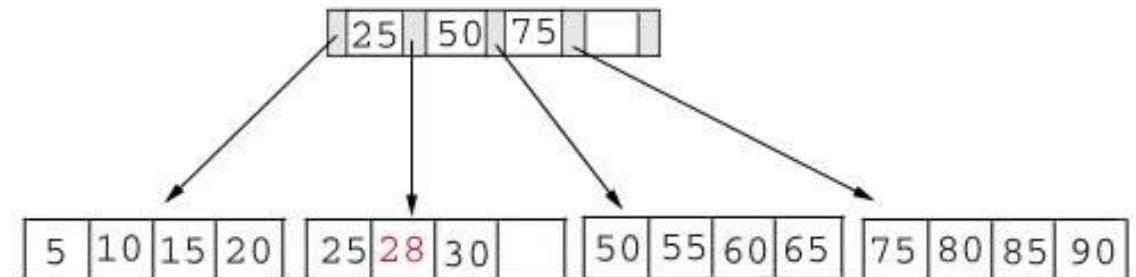
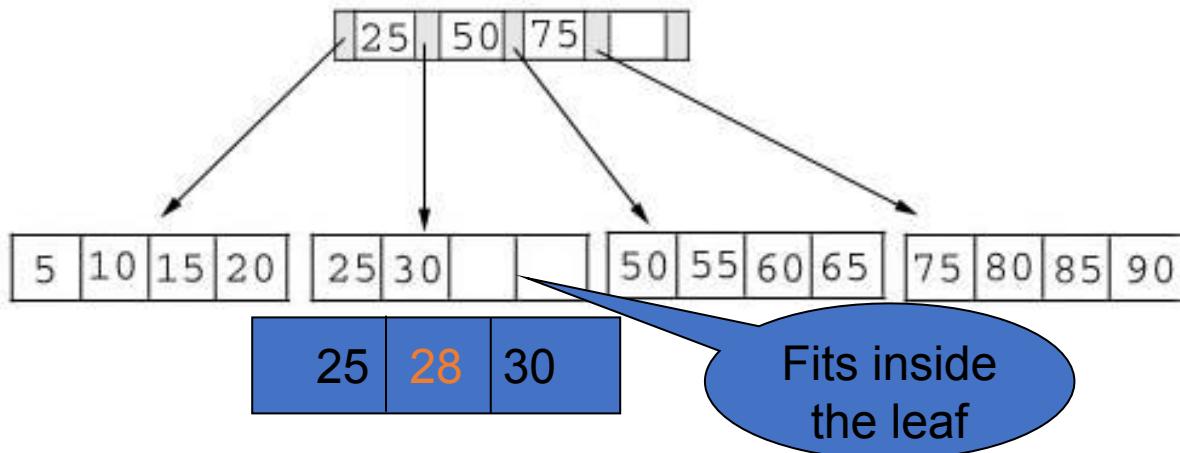
For example: find the value 45, and 15 in below tree.

- Result:
  1. For the value of 45, not found.
  2. For the value of 15, return the position where the pointer located.



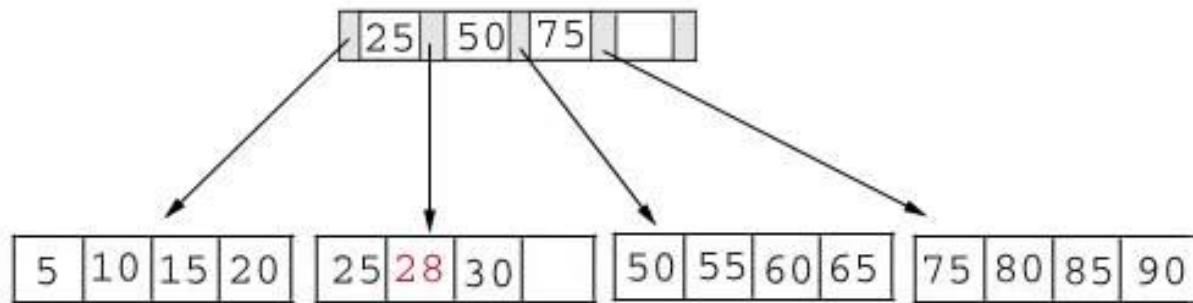
# Insertion

- inserting a value into a B+ tree may imbalance the tree, so rearrange the tree if needed.
- Example #1: insert 28 into the below tree.



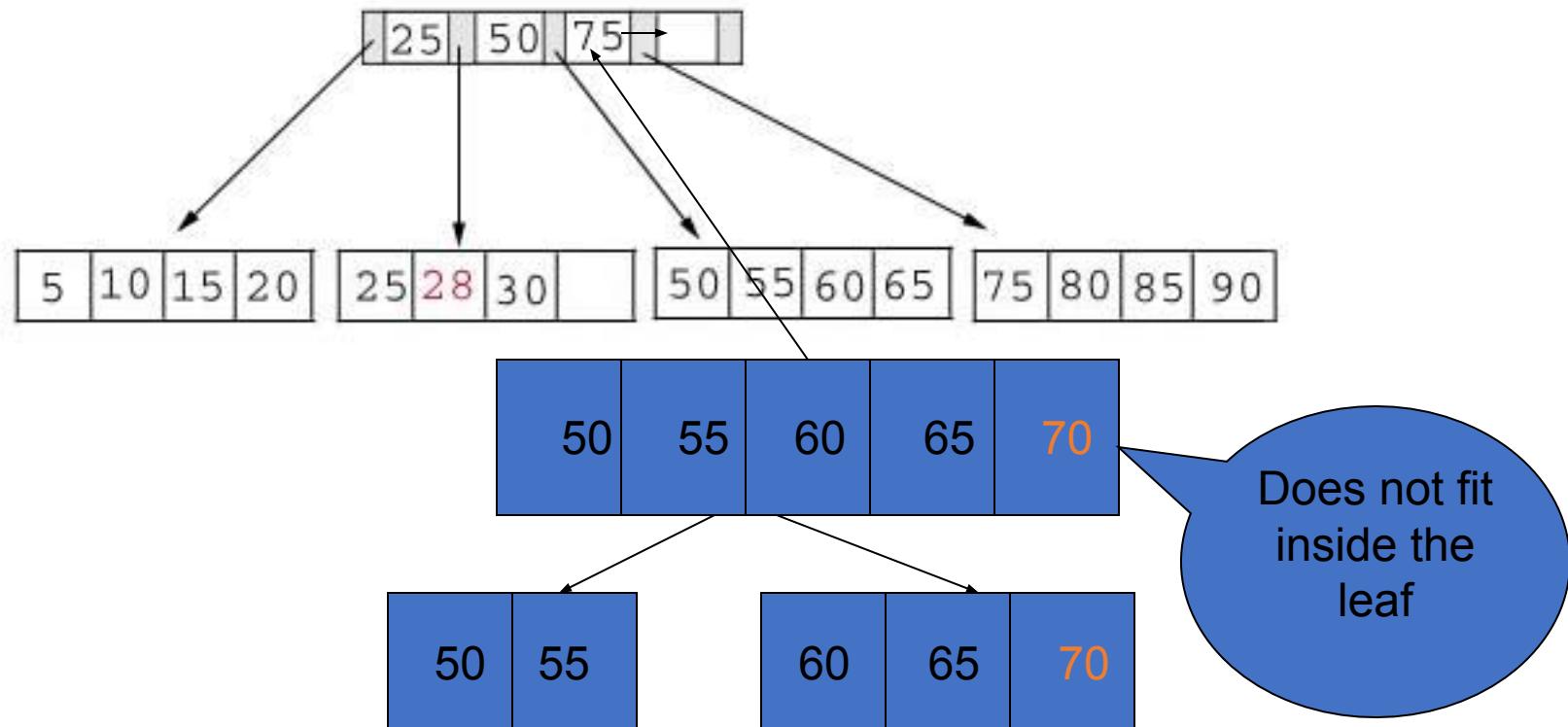
# Insertion

- Example #2: insert 70 into below tree



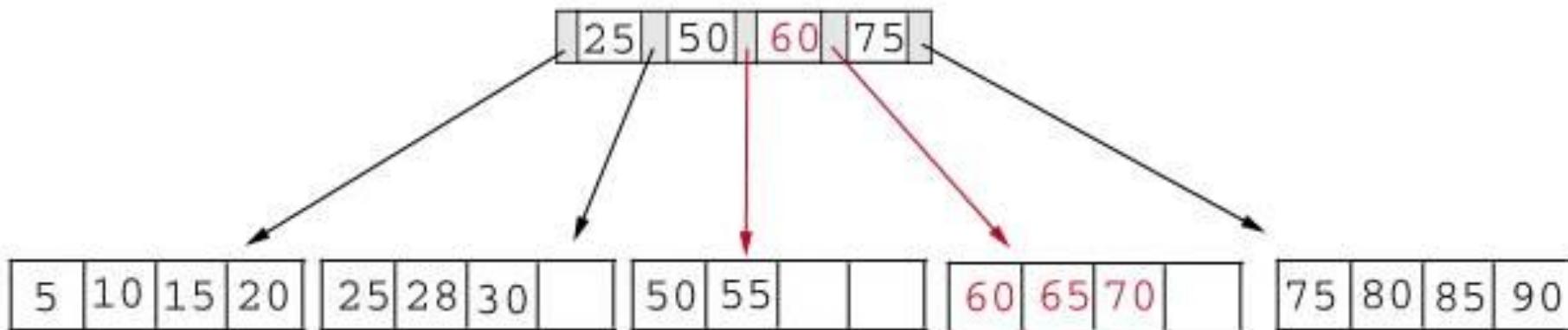
# Insertion

- Process: split the leaf and propagate middle key up the tree



# Insertion

- Result: choose the middle key 60 and place it in the index page between 50 and 75.



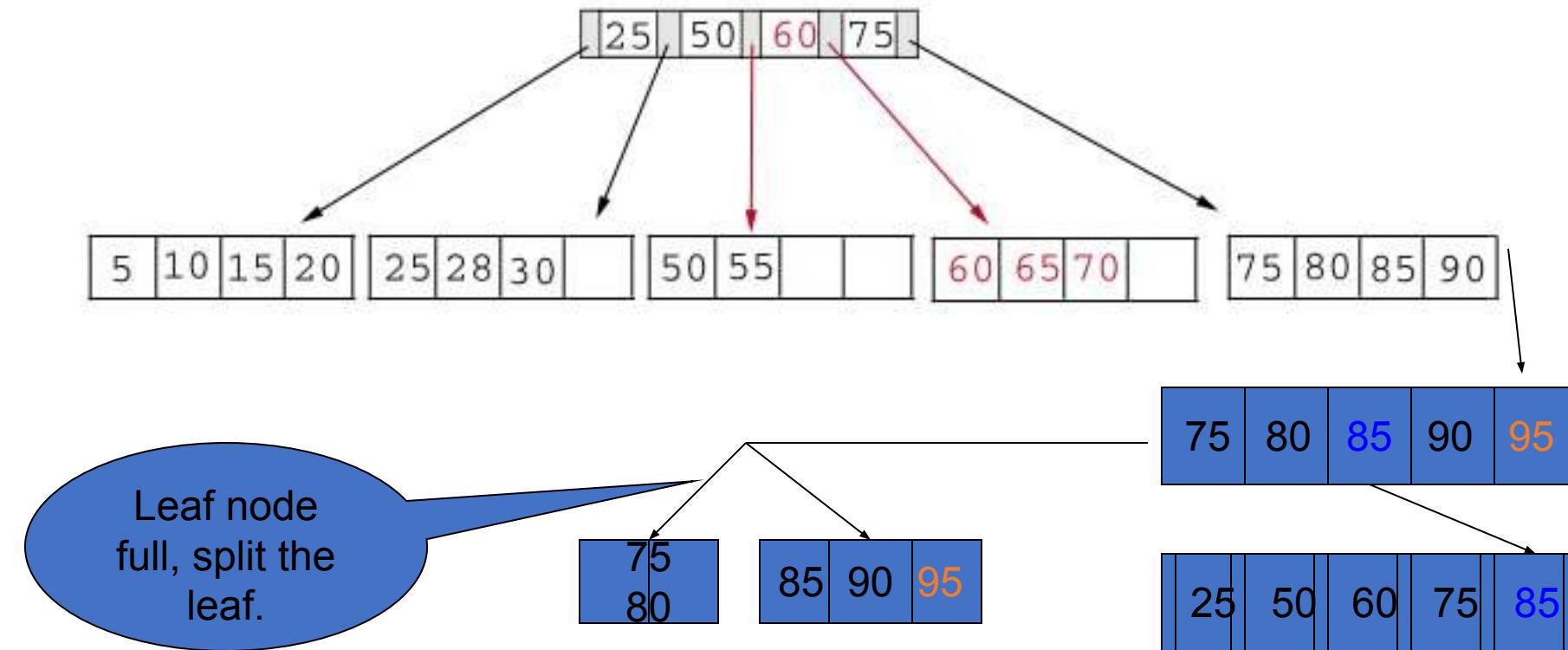
# Insertion

The insert algorithm for B+ Tree

| Leaf Node Full | Index Node Full | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|----------------|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NO             | NO              | Place the record in sorted position in the appropriate leaf page                                                                                                                                                                                                                                                                                                                                                                                                                           |
| YES            | NO              | <ol style="list-style-type: none"><li>1. Split the leaf node</li><li>2. Place Middle Key in the index node in sorted order.</li><li>3. Left leaf node contains records with keys below the middle key.</li><li>4. Right leaf node contains records with keys equal to or greater than the middle key.</li></ol>                                                                                                                                                                            |
| YES            | YES             | <ol style="list-style-type: none"><li>1. Split the leaf node.</li><li>2. Records with keys &lt; middle key go to the left leaf node.</li><li>3. Records with keys <math>\geq</math> middle key go to the right leaf node.</li><li>4. Split the index node.</li><li>5. Keys &lt; middle key go to the left index node.</li><li>6. Keys <math>&gt;</math> middle key go to the right index node.</li><li>IF the next level index node is full, continue splitting the index nodes.</li></ol> |

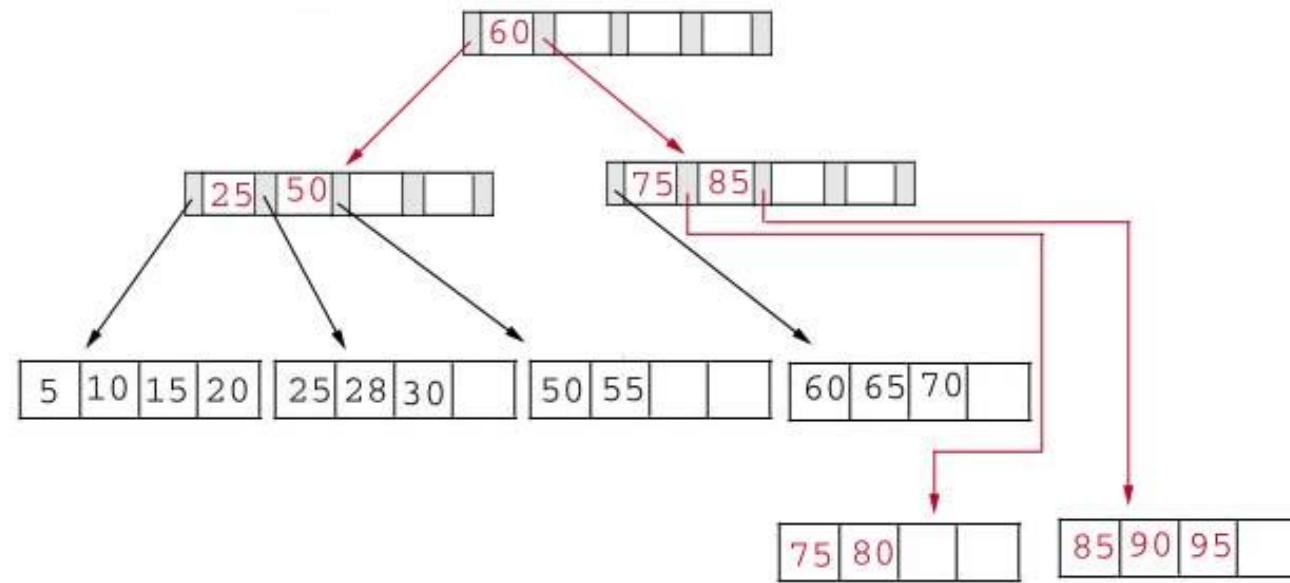
# Insertion

- Example: add a key value 95 to the below tree.



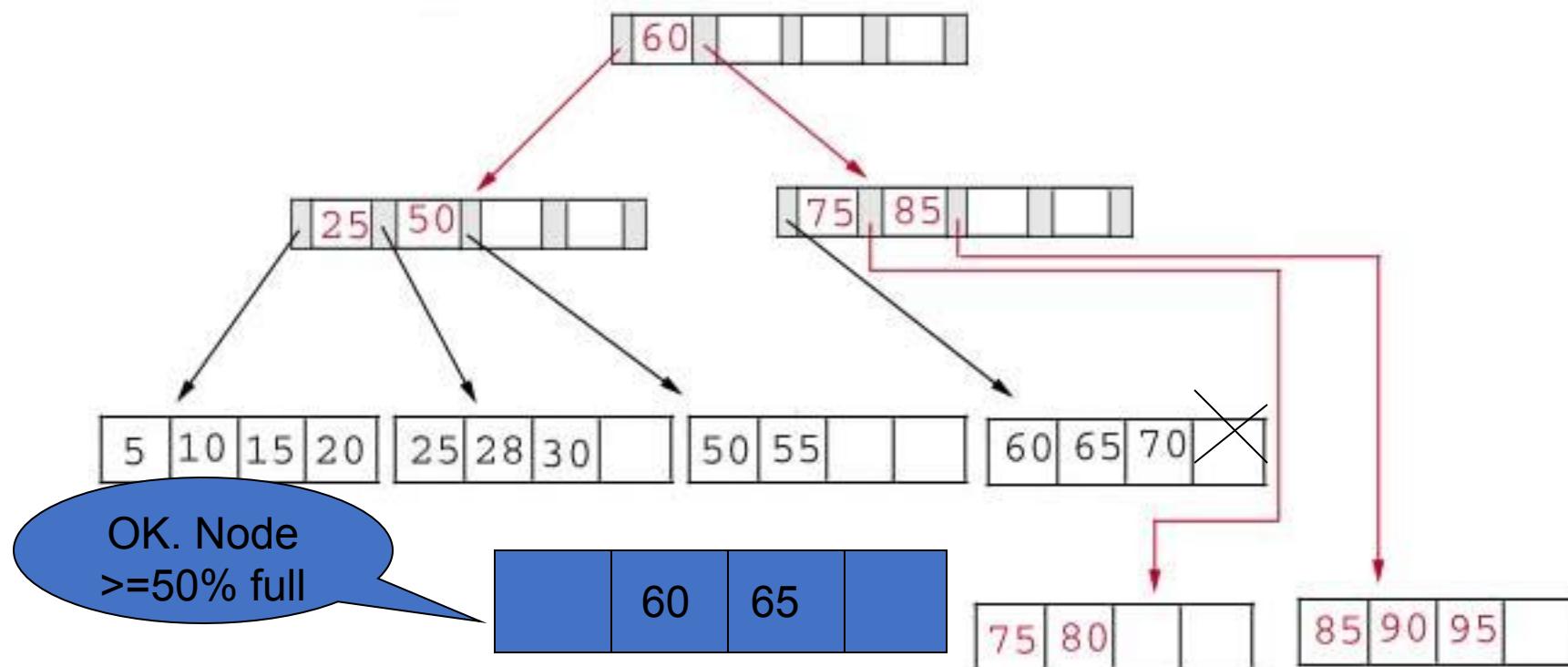
# Insertion

- Result: Again, put the middle key 60 to the index page and rearrange the tree.



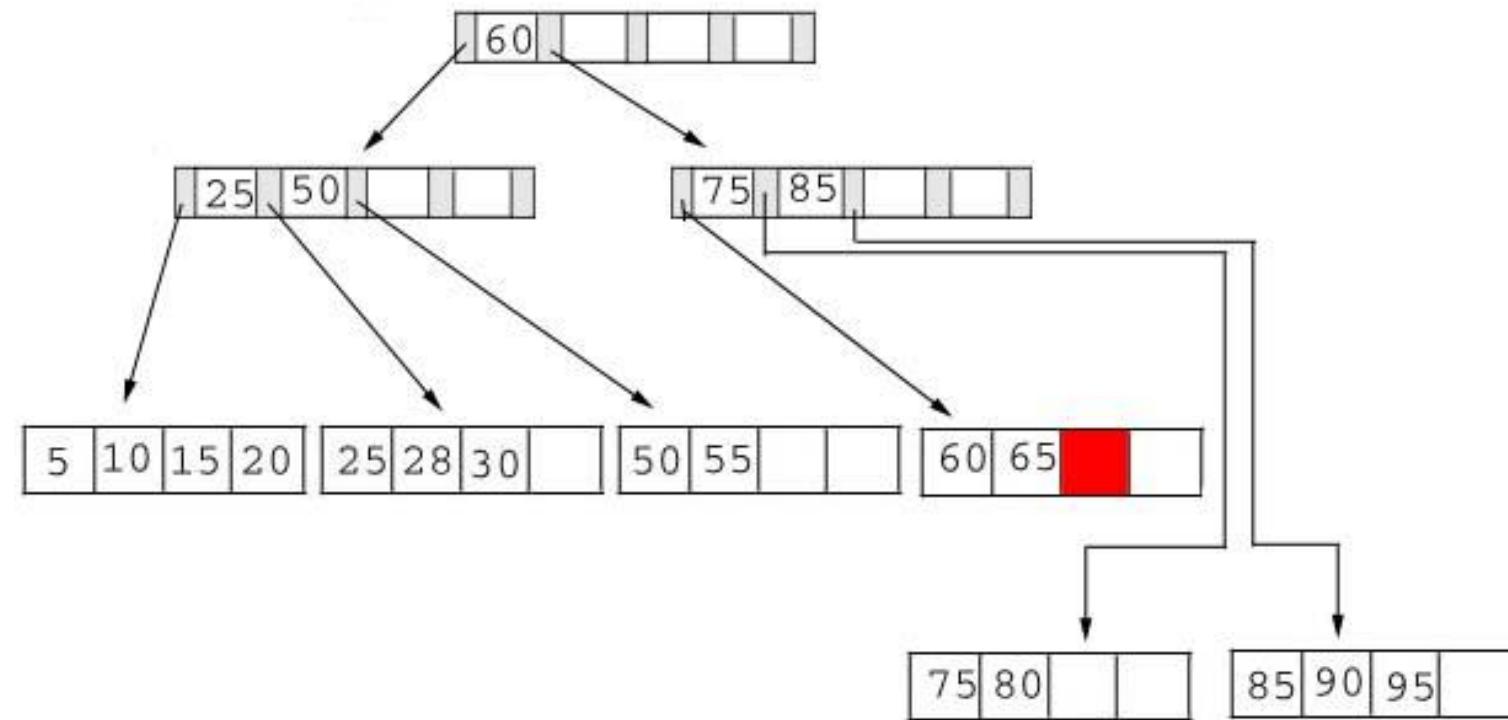
# Deletion

- Same as insertion, the tree must be rebuilt if the deletion result violate the rule of B+ tree.
- Example #1: delete 70 from the tree



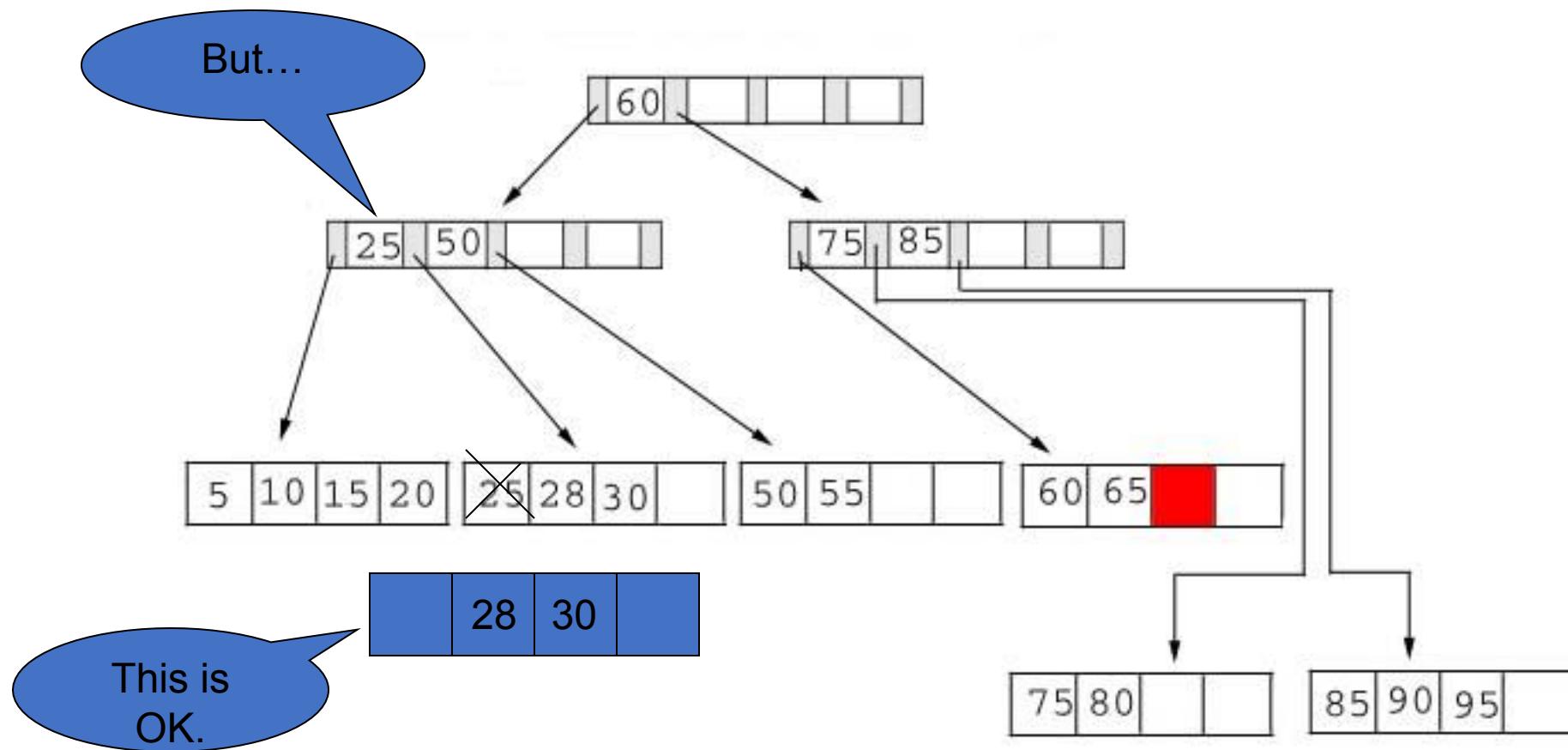
# Deletion

- Result:



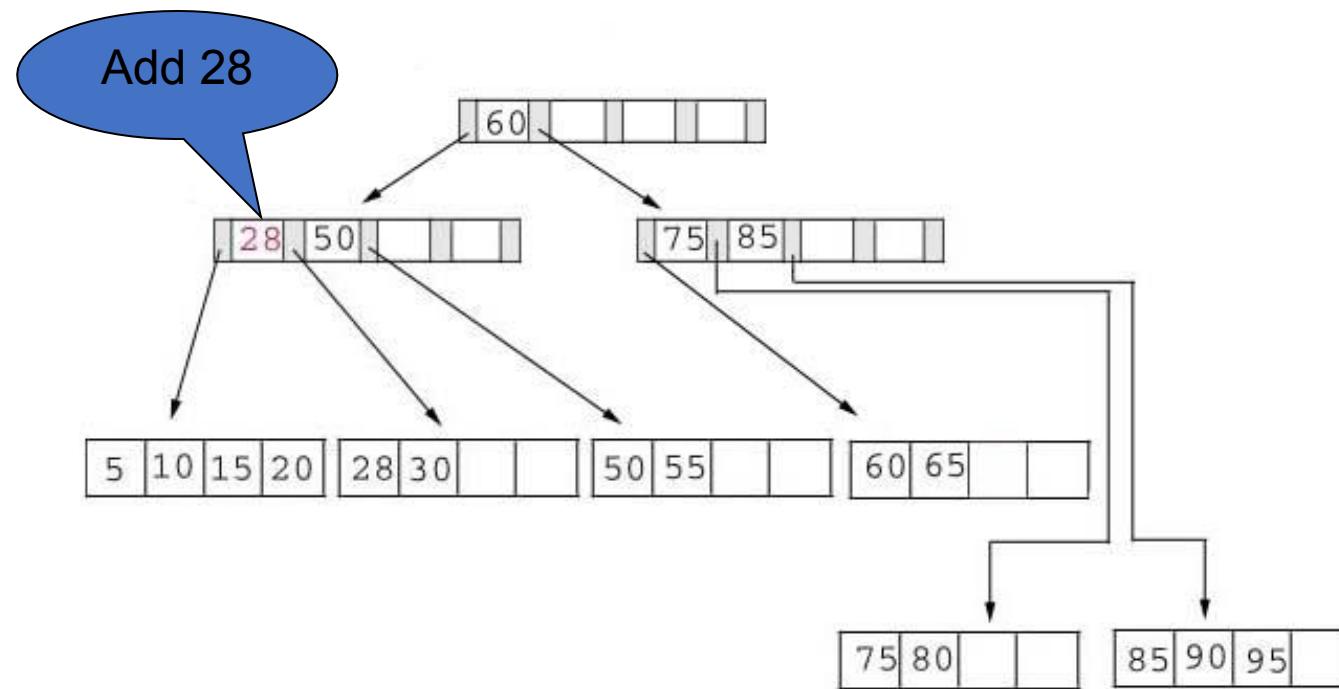
# Deletion

- Example #2: delete 25 from below tree, but 25 appears in the index page.



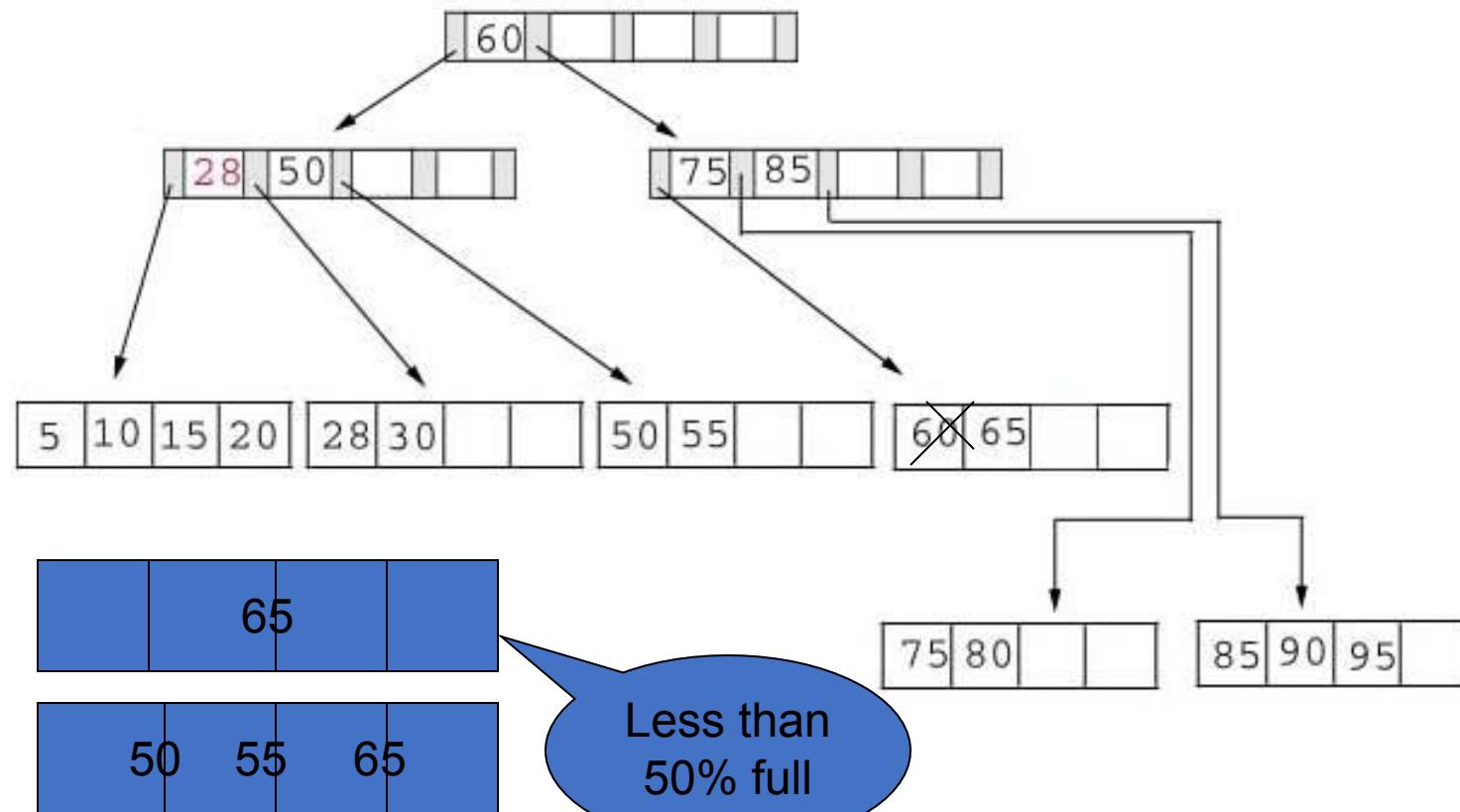
# Deletion

- Result: replace 28 in the index page.



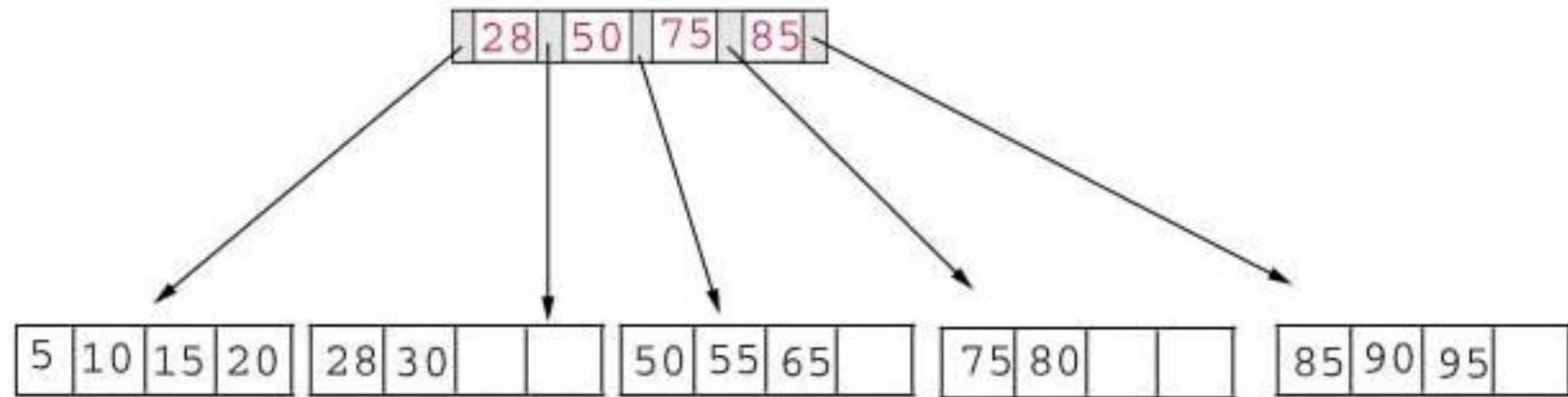
# Deletion

- Example #3: delete 60 from the below tree



# Deletion

- Result: delete 60 from the index page and combine the rest of index pages.



# Deletion

- Delete algorithm for B+ trees

| Data Page Below Fill Factor | Index Page Below Fill Factor | Action                                                                                                                                                                                                                                                                                                                      |
|-----------------------------|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NO                          | NO                           | Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.                                                                                                                                             |
| YES                         | NO                           | Combine the leaf page and its sibling. Change the index page to reflect the change.                                                                                                                                                                                                                                         |
| YES                         | YES                          | <ol style="list-style-type: none"><li>1. Combine the leaf page and its sibling.</li><li>2. Adjust the index page to reflect the change.</li><li>3. Combine the index page with its sibling.</li></ol> <p>Continue combining index pages until you reach a page with the correct fill factor or you reach the root page.</p> |

# Conclusion

- It is “easy” to maintain its balance
- Insert/Deletion complexity  $O(\log_{M/2})$
- The searching time is shorter than most of other types of trees because branching factor is high