COMP 6481: Programming and Problem Solving

Tutorial 10:

Binary search trees, AVL tree, Quick and merge sort

Outline

- Binary Search Trees
  - Definitions
  - Search and Update Algorithms
  - Performance
- AVL Trees
  - Definitions
  - Update Operations / Rotations
- Sorting Algorithms
  - Divide-and-Conquer Method
  - Merge Sort
  - Quick Sort
- Problem Solving

Binary Search Trees - Definitions (1)

- A ***binary search tree*** is a data structure for storing the entries of a dictionary.
- A ***binary search tree*** is a binary tree T such that each internal node v of T stores an entry (k,x) such that:
  - Keys stored at nodes in the left subtree of v are less than or equal to k.
  - Keys stored at nodes in the right subtree of v are greater than or equal to k.

Binary Search Trees - Definitions (2)

- Entries in a ***binary search tree*** are stored in internal nodes; empty external nodes are added to form a **proper** binary tree.

- An **inorder traversal** of nodes in a ***binary search tree*** lists the keys in increasing order.

Binary Search Trees - Searching Algorithm

```
Algorithm TreeSearch(p, k):
    if p is external then
        return p                                   {unsuccessful search}
    else if k == key(p) then
        return p                                   {successful search}
    else if k < key(p) then
        return TreeSearch(left(p), k)              {recur on left subtree}
    else {we know that k > key(p)}
        return TreeSearch(right(p), k)             {recur on right subtree}
```

- This algorithm has *O(h)* complexity, where *h* is the height of the binary search tree.
- In the worst case, it is linear because the height will be equal to *n*.

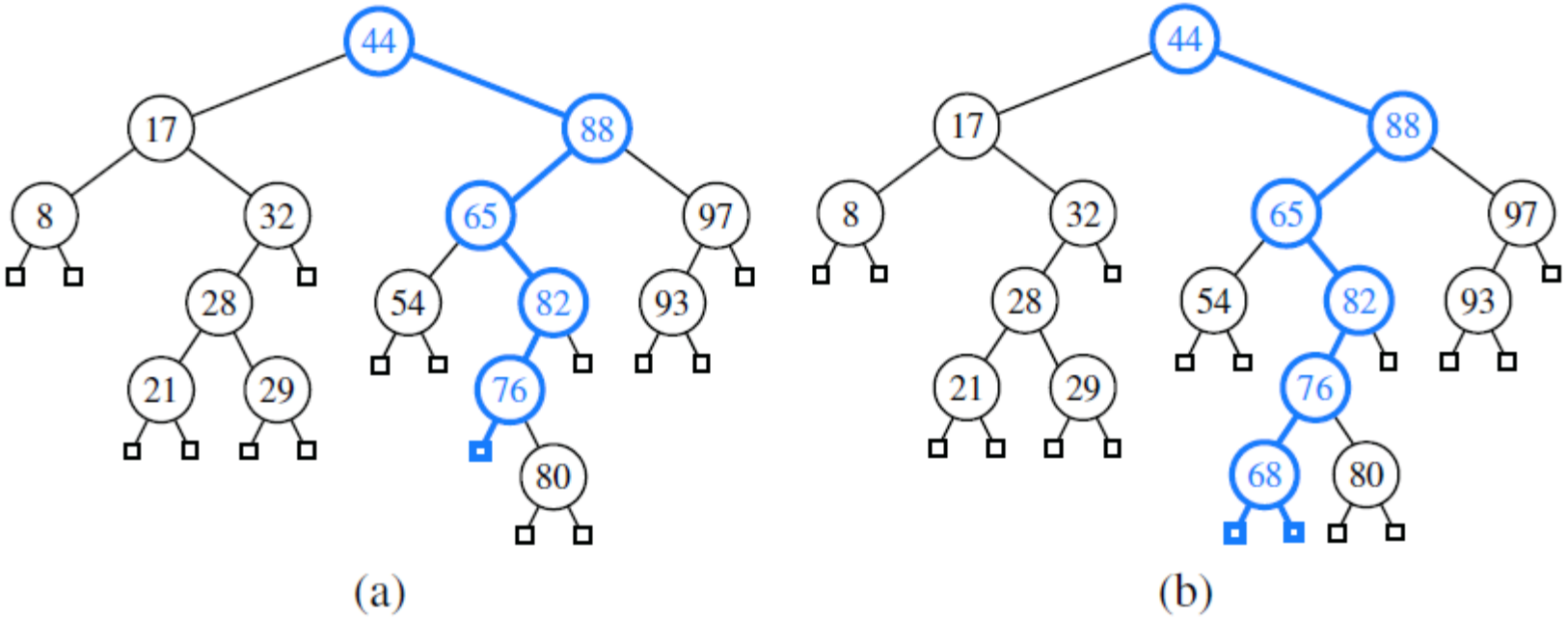
Binary Search Trees - Update Operations (1)

- *insertAtExternal(v,e):* Insert the element *e* at the external node *v*, and expand *v* to be internal, having new (empty) external node children; an error occurs if *v* is an internal node.
- Recursive Algorithm for insertion:

```
Algorithm TreeInsert(k, v):
    Input: A search key k to be associated with value v
    p = TreeSearch(root(), k)
    if k == key(p) then
        Change p's value to (v)
    else
        expandExternal(p, (k, v))
```

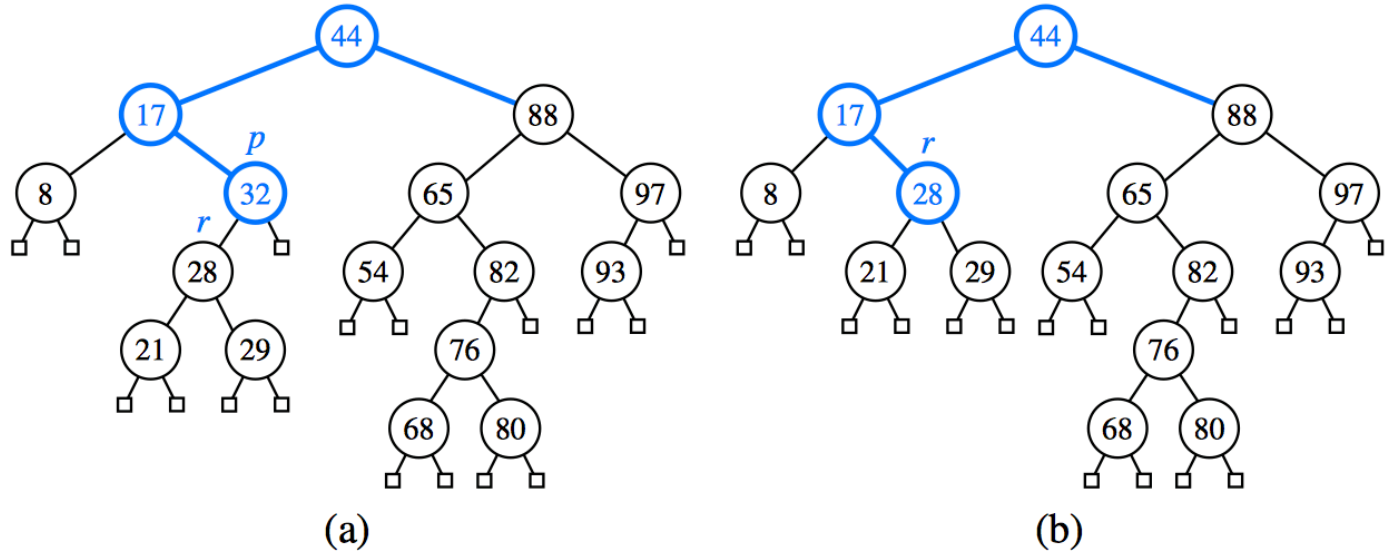
Binary Search Trees - Update Operations (2)

- Insertion of an entry with key 68:



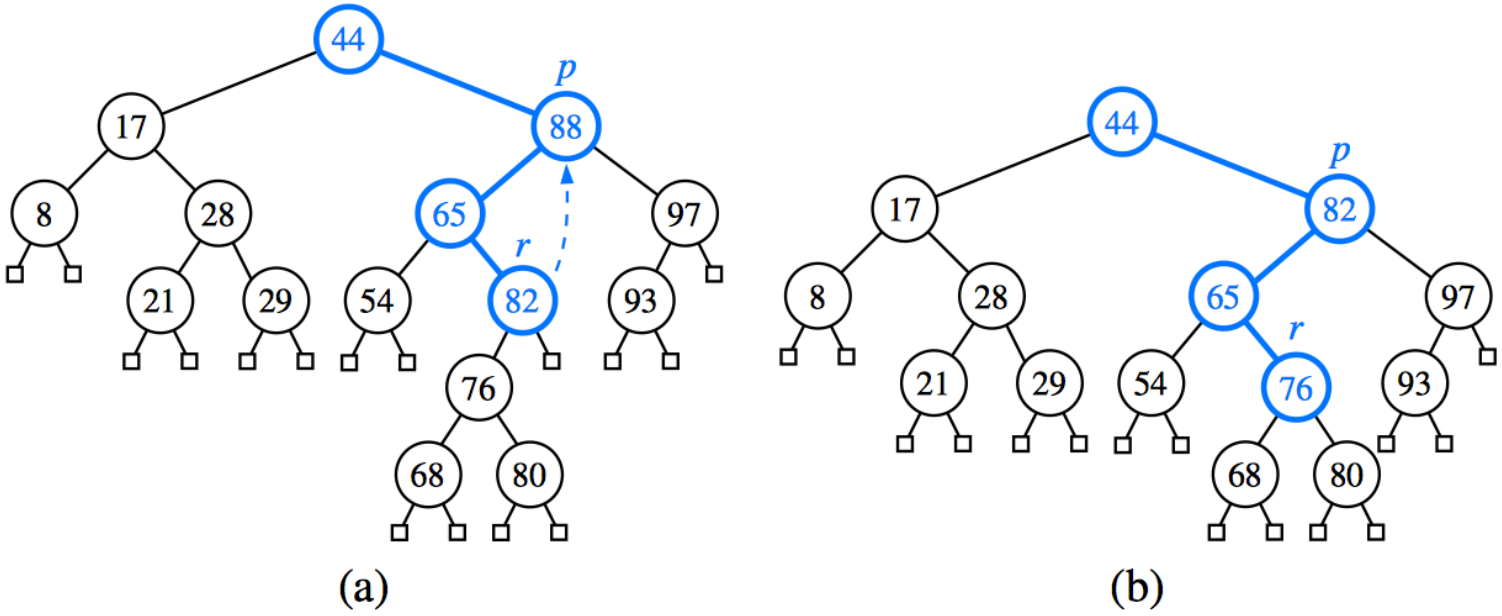
Binary Search Trees - Update Operations (3)

- *removeExternal(v):* Remove an external node *v* and its parent, replacing *v*'s parent with *v*'s sibling; an error occurs if *v* is not external.
- Removal with entry to be removed having an external child: *remove(32)*



Binary Search Trees - Update Operations (4)

Removal with entry to be removed having both its children internal: *remove(88)*



Binary Search Trees - Performance

- The find, insert, and remove methods run in $O(h)$ time, where h is the height of T.
- A binary search tree T is an efficient implementation of a dictionary with n entries only if the height of T is small
- In the worst case, T has height n

AVL Trees - Definitions

- An ***AVL Tree*** presents a more efficient way to implement a dictionary.

- It maintains a logarithmic-time performance, due to the ***Height-Balance Property***:
  - For every internal node *v* of the tree, the heights of the children of *v* differ by at most *1*.

- If this property is violated after insertion/removal from an AVL tree, a re-structure is needed.

AVL Trees - Rotations (1)

- Rotation Algorithm:
  - z is the first unbalanced node encountered while going upwards, y is the child of z with a higher height, and x the child of y with a higher height

Algorithm restructure(x):

Input: A position x of a binary search tree T that has both a parent y and a grandparent z

Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving positions x, y, and z

1: Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y, and z, and let (T_1, T_2, T_3, T_4) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.

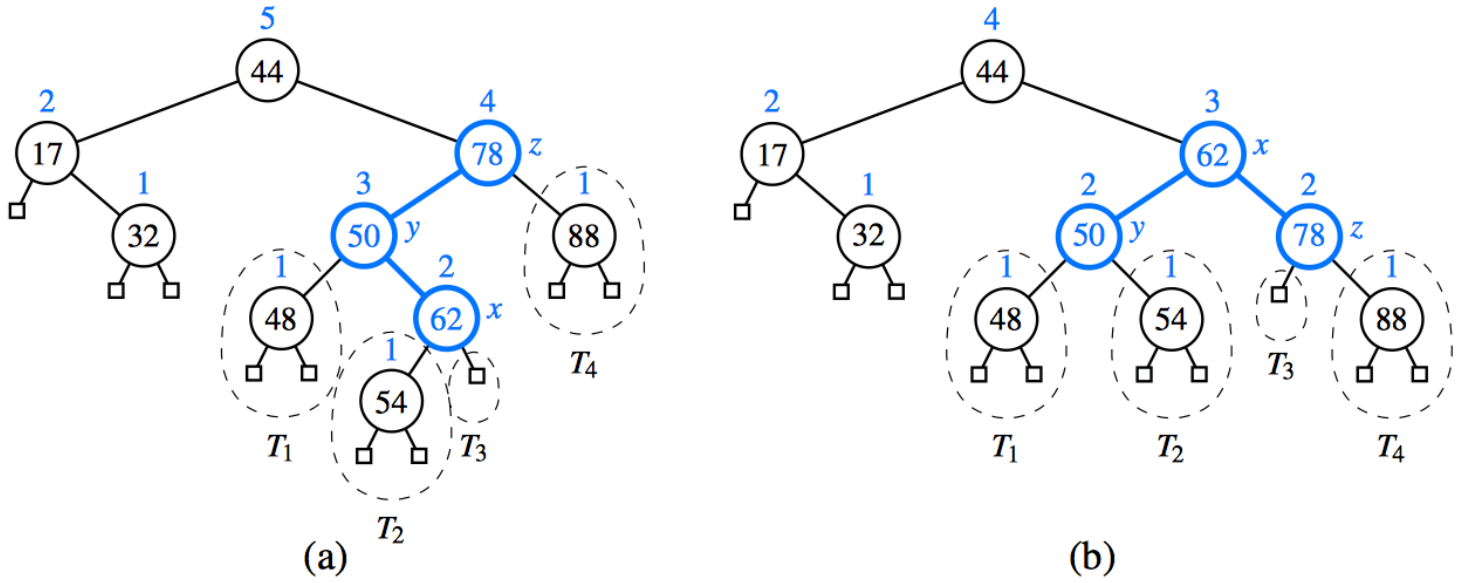
2: Replace the subtree rooted at z with a new subtree rooted at b.

3: Let a be the left child of b and let T_1 and T_2 be the left and right subtrees of a, respectively.

4: Let c be the right child of b and let T_3 and T_4 be the left and right subtrees of c, respectively.

AVL Trees – Rotations (2)

- Representation of a tree before and after a rotation:

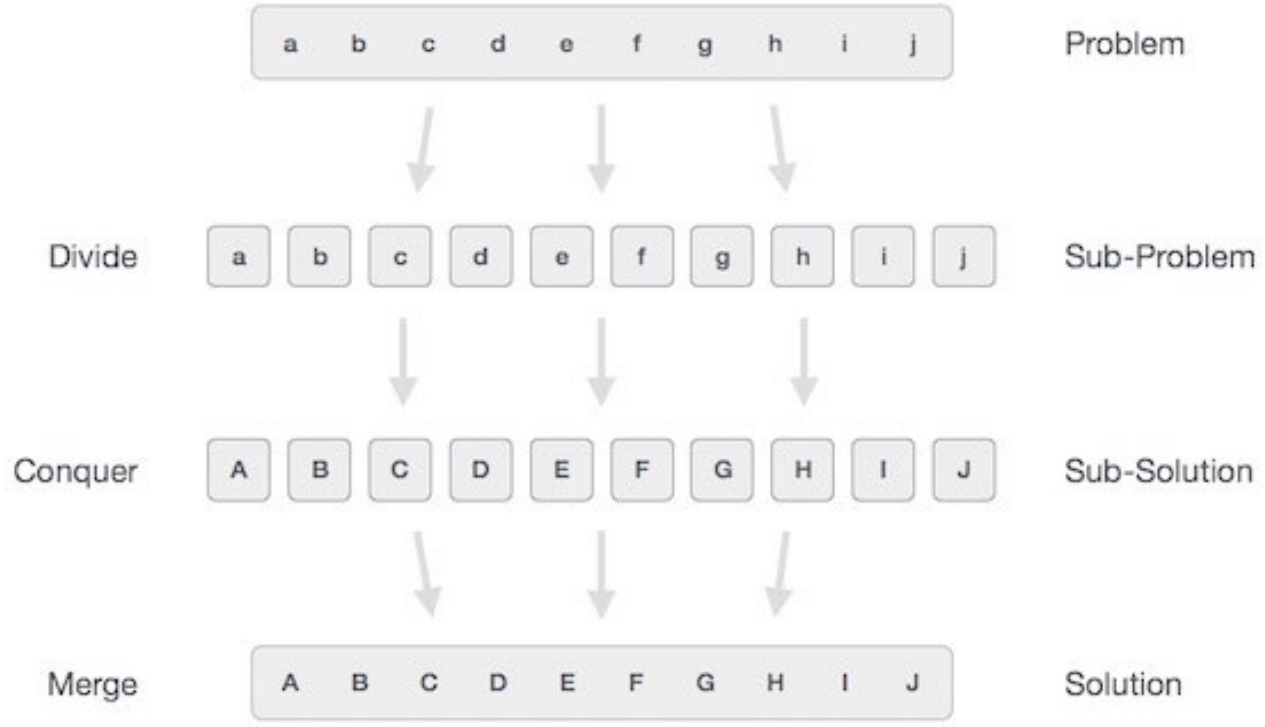


- If b= y, the trinode restructuring method is called a single rotation.
- If b= x, the trinode restructuring method is called a double rotation.

Divide-and-Conquer

- ***Divide and Conquer*** is a method of algorithm design.
- This method has three distinct steps:
  - **Divide/Break:** Divide the data into two or more disjoint subsets.
  - **Conquer/Solve:** Use divide and conquer to solve the subproblems associated with the subsets.
  - **Merge/Combine:** combine the subproblem solutions into a solution for the original problem.

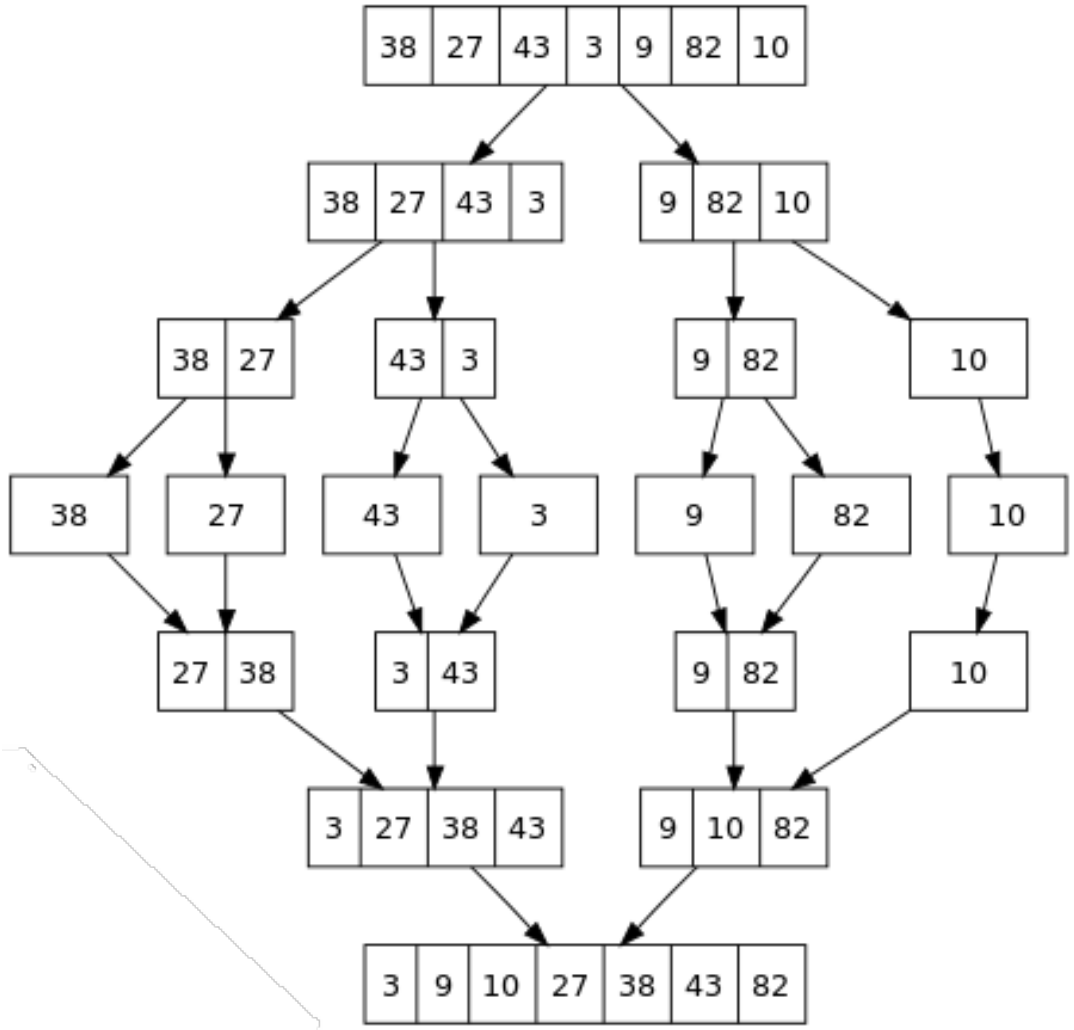
Divide-and-Conquer



Merge-Sort Algorithm

- Merge sort is a sorting technique based on divide and conquer technique.
- **Divide**: If S has at least two elements (nothing needs to be done if S has zero or one elements), remove all the elements from S and put them into two sequences, S_1 and S_2, each containing about half of the elements of S. (i.e. S_1 contains the first $\lceil n/2 \rceil$ elements and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements.
- **Recur**: Recursive sort sequences S_1 and S_2.
- **Conquer**: Put back the elements into S by merging the sorted sequences S_1 and S_2 into a unique sorted sequence.

Merge-Sort Example



Running Time of Merge-Sort

- At each level in the binary tree created for Merge-Sort O(n) time is spent
- The height of the tree is O(log n)

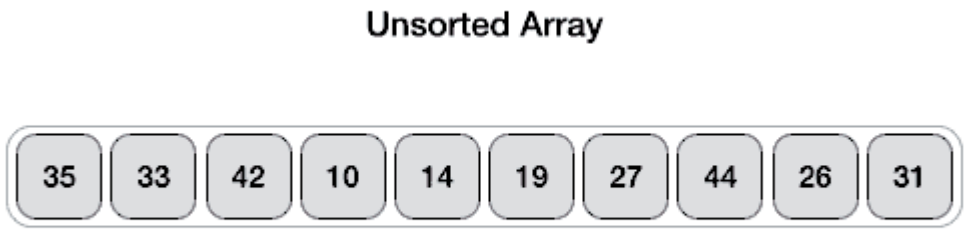
- Therefore, the time complexity is ***O(nlog n)***

Quick-Sort

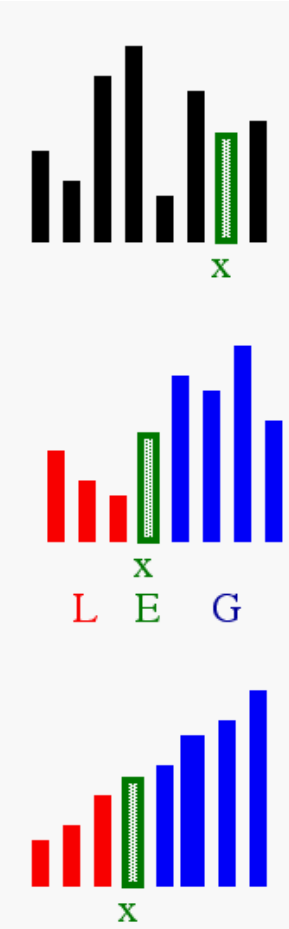
- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- **Divide** : If the sequence S has 2 or more elements, select an element x from S to be your **pivot**. Any arbitrary element, like the last, will do. Remove all the elements of S and divide them into 3 sequences:
  - L, holds S's elements less than x
  - E, holds S's elements equal to x
  - G, holds S's elements greater than x
- **Recurse**: Recursively sort L and G
- **Conquer**: Finally, to put elements back into S in order, first inserts the elements of L, then those of E, and those of G.

How quick Sort Works?

- The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.



Idea of Quick Sort

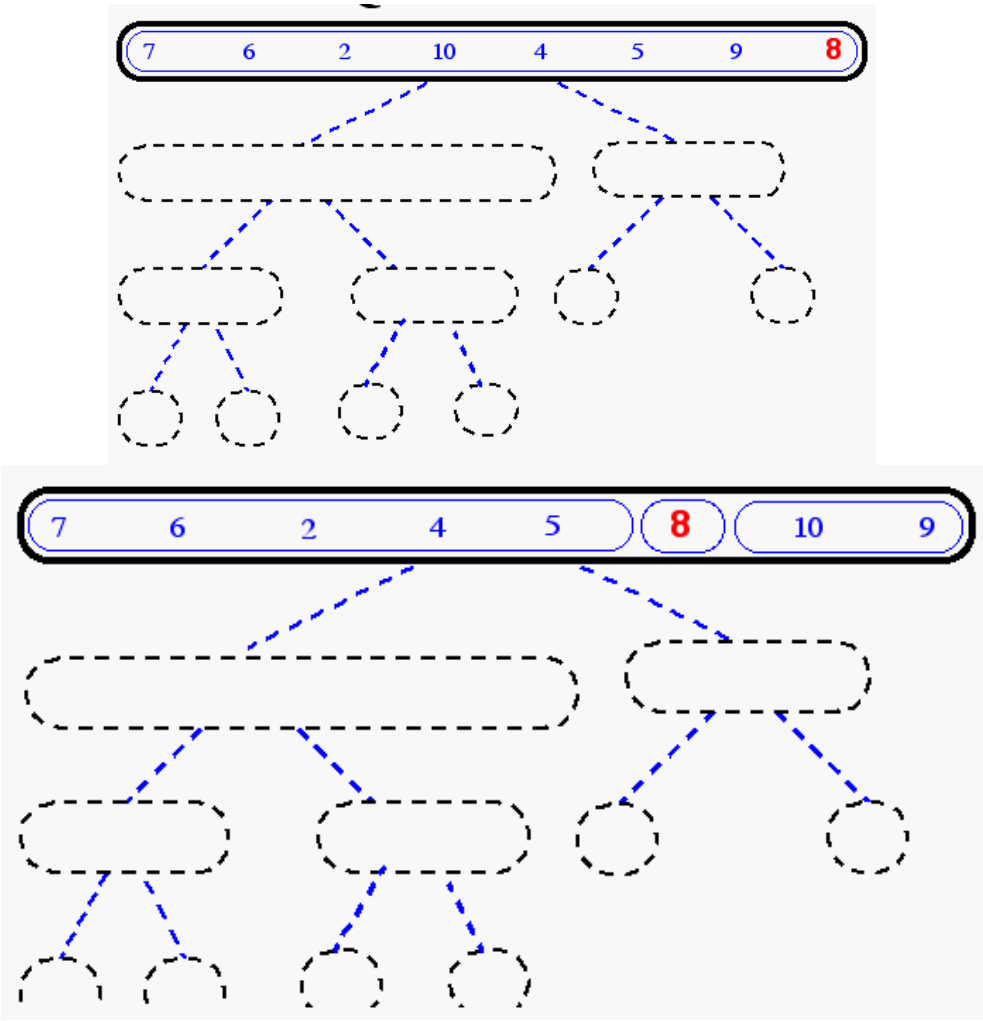


1) **Select**: pick an element

2) **Divide**: rearrange elements so that x goes to its final position E

3) **Recur and Conquer**: recursively sort

Quick-Sort Tree



Quick Sort Running Time

- Worst case: when the pivot does not divide the sequence in two
- At each step, the length of the sequence is only reduced by 1
- Total running time

$$\sum_{i=n}^{1} length(S_i) = O(n^2)$$

- General case:
  - Time spent at level i in the tree is O(n)
  - Running time: O(n) * O(height)
- Average case:
  - O(n log n)

Problem Solving
Question 1

- We defined a BST so that keys equal to a node's key can be in either the left or right subtree of the node. Suppose we change the definition so that we restrict equal keys to the right subtree.
- What must a subtree of a binary search tree containing only equal keys look like in this case?

Problem Solving
Question 2

- Dr. Amongus claims that the order in which a fixed set of entries is inserted into a binary search tree does not matter— the same tree results every time. Give a small example that proves him wrong.

Problem Solving
Question 3

- How many different BST's can store the keys *{1,2,3}*?

Problem Solving - Question 4

- Draw a Binary Search Tree that initially is empty and shows the result of the tree after inserting the following keys (from left to right):
  - key ={ 30, 40, 24, 58, 48, 26, 11, 13 }

Problem Solving - Question 5

- Build an AVL tree with the following keys:

  {3, 2, 1, 4 ,5 ,6 ,7 ,16 ,15}

Problem Solving - Question 6

- Using the AVL tree obtained in Question 5, delete nodes with values 1 and 3, and draw the final orientation of the tree.

Problem Solving - Question 7

A company wants to offer a free CD to its customers. However, it wishes to offer 1 CD per household. Customers of the same household are defined as those that share the same address. Customers are kept in an unsorted linked list, where each element contains a name and an address.

Describe an efficient algorithm, in pseudocode or in English, to prune the company's customers list. Make sure that you are doing better than the naïve, inefficient O(n^2) method that compares every possible pair of records.

Problem Solving - Question 8

- Given an array A[1...n] of integers, give O(*nlogn*) algorithm to find the most commonly occurring element. If there is more than one such element, you should return the larger element. For example, if the array is [1,9,5,9,4,7,9], then you should return 9 since it is the only element that appears twice. If the array is [9,5,6,5,3,9,15], both 5 and 9 appear twice, but since 9>5, you should return 9. You must give a pseudocode for your algorithm as well as explain why it is O(*nlogn*).

Problem Solving - Question 9

- A lazy programmer thinks that implementing the procedure Merge (of the merge sort) is too complicated. To simplify merge sort he eliminates the merge all together, and only recursively “sorts" the two halves of the array. Here is his pseudo code.

```
1: procedure LazySort(A[1 .. n])
2: if n > 1 then
3: m =  n / 2
4: LazySort(A[1 .. m])
5: LazySort(A[m + 1 .. n])
```

Help the lazy programmer to understand his algorithm by answering the following questions.

- What does LazySort do?
- What is the running time of LazySort?