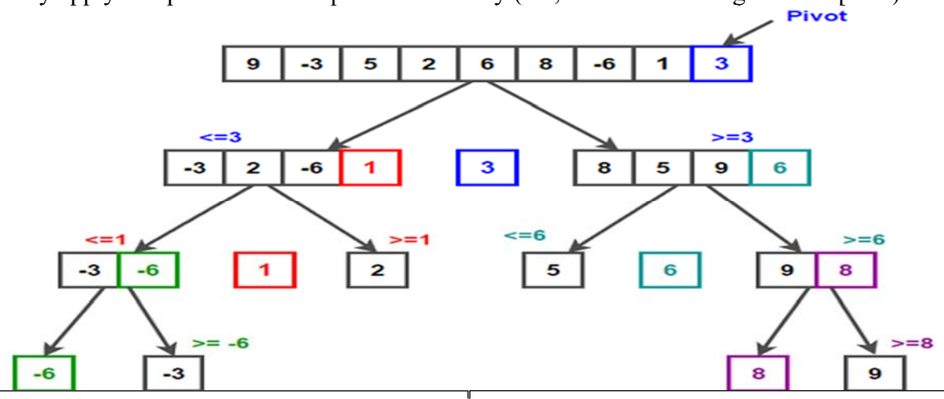
Exercise 2 : Sorting algorithms: (8p)

QuickSort is a widely used sorting algorithm known for its efficiency and simplicity. It belongs to the divide-and-conquer family of algorithms and works by **recursively partitioning** the input array or list, **sorting** the partitions, and **combining** them to achieve the final sorted result. Here are the key steps of the QuickSort algorithm:

1. Choose an element from the array to serve as the pivot.
2. Perform value exchanges in the array so that:
   - Values lower than the pivot are placed to its left.
   - Values higher than the pivot are placed to its right.
3. Recursively apply this process to both parts of the array (i.e., to the left and right of the pivot).



3. **Implement Partition Function: (4 p)**

- **Description:** Rearrange elements around a pivot in the linked list.
- **Logic:** Use two pointers to swap elements less than the pivot to its left and elements greater to its right.

```
struct Node* partition (struct Node* low, struct Node* high) {
    int pivot = high->data;
   struct Node* i = low->prev;

   for (struct Node* j = low; j != high; j = j->next) {
      if (j->data < pivot) {
         if (i == NULL) {
            i = low;
         } else {
            i = i->next;
         }
         // Swap data values
         int temp = i->data;
         i->data = j->data;
         j->data = temp;
      }
   }
   if (i == NULL) {
      i = low;
   } else {
      i = i->next;
   }
   // Swap data values
   int temp = i->data;
   i->data = high->data;
   high->data = temp;

   return i;
}
```

1. **Implement QuickSortRec Function (2 p)**

- **Description:** Recursively apply QuickSort to sublists on the left and right of the pivot.

```
void quickSortRec(struct Node* low, struct Node* high)
{
if (high != nullptr && low != high && low != high->next) {
      Node* pivot = partition(low, high);
      quickSortRec(low, pivot->prev);
      quickSortRec(pivot->next, high);
   }
}
```

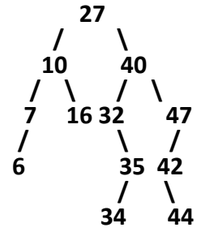
2. **Implement QuickSort Function: (2 p)**

- **Description**: Initialize QuickSort by finding the end of the list and calling QuickSortRec**.**

```
void quickSort(struct Node** head) {
    Node* end = head;
    while (end != nullptr && end->next != nullptr) {
       end = end->next;
    }
    quickSortRec(head, end);
}
```

Exercise 3: trees & graphs (9p):

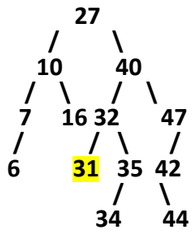
1/Given the following Tree:



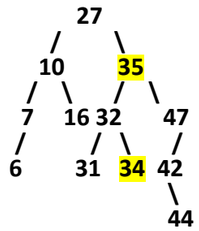
1/ display In_order traversal (1p):

2/ display In_order traversal (1p):

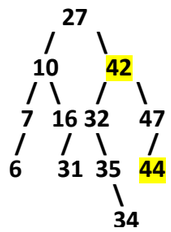
1.1/Insert new Node 31: (1p)



1.2/ Delete the Node 40:(1p)



Or



4/A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.

```
struct Node { int key ;
              struct Node *left, *right;
  };
bool isFullBinaryTree(struct Node *root) {
  // Checking tree emptiness
  if (root == NULL)
    return true;
  // Checking the presence of children
  if (root->left == NULL && root->right == NULL)
    return true;
  if ((root->left) && (root->right))
    return (isFullBinaryTree(root->left) && isFullBinaryTree(root->right));

  return false;
}}
```

University Kasdi Merbah Ouargla
Faculty of New Technologies of Information and Communication
Computer Science & information technology Department
SEMESTER 3 EXAMINATION

Module: Algorithm and Data Structure	***Last name:***
Level: 2nd year	***First name:***
Date: 08/03/2012	***Group:***
Duration: 1h30	***Registration number:***

Exercise 1 : Complexity (3p)

Calculate the complexity of the following algorithms: "Write only the result."

```
1. int sum_of_pairs_recursive(const int arr[], int i, int j, int total_sum, int n) {
2.     if (i < n) {
3.         if (j < n) {
4.             total_sum += arr[i] + arr[j];
5.             return sum_of_pairs_recursive(arr, i, j + 1, total_sum, n);
6.         } else {
7.             // Increment i correctly here
8.             return sum_of_pairs_recursive(arr, i + 1, i + 2, total_sum, n);
9.         }
10.    } else {
11.        return total_sum;
12.    }
13. }
```

Complexity is: O(n^2)

```
1. x = n;
2. while (x > 0) {
3.     y = n; j = 1;
4.     while (j < n) {
5.         if (y > 1)
6.             y = y / 2
7.         else {
8.             y = x
9.             j++
10.        }
11.    }
12.    x = x / 4
13. }
```

Complexity is: O(n log^2 n)