Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	θ(nk)	O(nk)

Type	Explanation	Format Specifier
char	Smallest addressable unit of the machine that can contain basic character set. It is an integer type. Actual type can be either signed or unsigned. It contains `CHAR_BIT` bits.[3]	%c
signed char	Of the same size as `char`, but guaranteed to be signed. Capable of containing **at least** the [−127, +127] range;[3][4]	%c (or %hhi for numerical output)

unsigned char	Of the same size as `char`, but guaranteed to be unsigned. Contains **at least** the [0, 255] range.[5]	%c (or %hhu for numerical output)
short short int signed short signed short int	*Short* signed integer type. Capable of containing **at least** the [−32,767, +32,767] range;[3][4] thus, it is at least 16 bits in size. The negative value is −32767 (not −32768) due to the one's-complement and sign-magnitude representations allowed by the standard, though the two's-complement representation is much more common.[6]	%hi
unsigned short unsigned short int	*Short* unsigned integer type. Contains **at least** the [0, 65535] range;[3][4]	%hu
int signed signed int	Basic signed integer type. Capable of containing **at least** the [−32,767, +32,767] range;[3][4] thus, it is at least 16 bits in size.	%i or %d
unsigned unsigned int	Basic unsigned integer type. Contains **at least** the [0, 65535] range;[3][4]	%u
long long int signed long signed long int	*Long* signed integer type. Capable of containing **at least** the [−2,147,483,647, +2,147,483,647] range;[3][4] thus, it is at least 32 bits in size.	%li
unsigned long unsigned long int	*Long* unsigned integer type. Capable of containing **at least** the [0, 4,294,967,295] range;[3][4]	%lu
long long long long int signed long long signed long long int	*Long long* signed integer type. Capable of containing **at least** the [−9,223,372,036,854,775,807, +9,223,372,036,854,775,807] range;[3][4] thus, it is at least 64 bits in size. Specified since the C99 version of the standard.	%lli
unsigned long long unsigned long long int	*Long long* unsigned integer type. Contains **at least** the [0, +18,446,744,073,709,551,615] range;[3][4] Specified since the C99 version of the standard.	%llu

float	Real floating-point type, usually referred to as a single-precision floating-point type. Actual properties unspecified (except minimum limits), however on most systems this is the IEEE 754 single-precision binary floating-point format. This format is required by the optional Annex F "IEC 60559 floating-point arithmetic".	for formatted input: %f %F for digital notation, or %g %G, or %e %E %a %A for scientific notation[7]
double	Real floating-point type, usually referred to as a double-precision floating-point type. Actual properties unspecified (except minimum limits), however on most systems this is the IEEE 754 double-precision binary floating-point format. This format is required by the optional Annex F "IEC 60559 floating-point arithmetic".	%lf %lF %lg %lG %le %lE %la %lA;[7] for formatted output, the length modifier l is optional.
long double	Real floating-point type, usually mapped to an extended precision floating-point number format. Actual properties unspecified. Unlike types float and double, it can be either 80-bit floating point format, the non-IEEE "double-double" or IEEE 754 quadruple-precision floating-point format if a higher precision format is provided, otherwise it is the same as double. See the article on long double for details.	%Lf %LF %Lg %LG %Le %LE %La %LA[7]

Rabin-Karp Complexity

• If a sufficiently large prime number is used for the hash function, the hashed values of two different patterns will usually be distinct.

• If this is the case, searching takes O(N) time, where N is the number of characters in the larger body of text.

• It is always possible to construct a scenario with a worst case complexity of O(MN). This, however, is likely to happen only if the prime number used for hashing is small

KMP

• define k = i - j

• In every iteration through the while loop, one of three things happens.

1) if T[i] = P[j], then i increases by 1, as does j
k remains the same.

2) if T[i] != P[j] and j > 0, then i does not change and k increases by at least 1, since k changes from i - j to i - f(j-1)

3) if T[i] != P[j] and j = 0, then i increases by 1 and k increases by 1 since j remains the same.

• Thus, each time through the loop, either i or k increases by at least 1, so the greatest possible number of loops is 2n

• This of course assumes that f has already been computed.

• However, f is computed in much the same manner as KMPMatch so the time complexity argument is analogous. KMPFailureFunction is O(m)

• Total Time Complexity: O(n + m)

*Stack

```
struct Stack
{
	SNode *head;
};

void init(Stack &s)
{
	s.head = NULL;
}

bool isEmpty(Stack s)
{
	return s.head == NULL;
}

bool isFull(Stack s)
{
	return false;
}

bool push(Stack &s, string x)
{
	SNode *p = getNode(x);

	if (isEmpty(s))
	{
		s.head = p;
	}
	else
	{
		p->next = s.head;
		s.head = p;
	}
	return true;
}

bool pop(Stack &s, string &x)
{
	if (isEmpty(s)) return false;
	SNode *p;
	p = s.head;
	x = p->data;
	s.head = s.head->next;
	delete p;
	return true;
}

string peek(Stack s)
{
	if (isEmpty(s)) return "";
	return s.head->data;
}
```

```
void input(Stack &s)
{
	init(s);
	int n;
	cout << "Enter n: ";
	cin >> n;
	if (n <= 0)
	{
		cout << "N is invalid..." << endl;
		return;
	}
	for (int i = 1; i <= n; i++)
	{
		string x;
		cout << "Enter element " << i << ": ";
		cin >> x;
		push(s, x);
	}
}

void output(Stack s)
{
	while (!isEmpty(s))
	{
		string x;
		pop(s, x);
		cout << x << "   ";
	}
}
```

*Queue

```
struct SNode
{
	string data;
	SNode *next;
};

struct Queue
{
	SNode *head;
	SNode *tail;
};

void init(Queue &q)
{
	q.head = q.tail = NULL;
}

SNode* getNode(string x)
{
	SNode*p = new SNode;
	if (p == NULL)
	{
```

```
		cout << "Not enough memory ! " << endl;
		exit(0);
	}
	p->data = x;
	p->next = NULL;
	return p;
}

bool isEmpty(Queue q)
{
	return q.head == NULL;
}

bool isFull(Queue q)
{
	return false;
}

bool push(Queue &q, string x)
{
	SNode *p = getNode(x);

	if (isEmpty(q))
	{
		q.head = q.tail = p;
	}
	else
	{
		q.tail->next = p;
		q.tail = p;
	}
	return true;
}

bool pop(Queue &q, string &x)
{
	if (isEmpty(q)) return false;

	SNode *p;
	p = q.head;
	x = p->data;
	q.head = q.head->next;
	delete p;
	return true;
}

bool peek(Queue q, string &x)
{
	if (isEmpty(q)) return false;
	x = q.head->data;
	return true;
}

void input(Queue &q)
```

```
{
	init(q);
	int n;
	cout << "Enter n: ";
	cin >> n;

	if (n <= 0)
	{
		cout << "N is invalid..." << endl;
		return;
	}

	for (int i = 1; i <= n; i++)
	{
		string x;
		cout << "Enter element " << i << ": ";
		cin >> x;
		push(q, x);
	}
}

void output(Queue q)
{
	while (!isEmpty(q))
	{
		string x;
		pop(q, x);
		cout << x << "    ";
	}
}
```

*Linked list

```
void reverseDispList()
{
    struct node *prevNode, *curNode;

    if(stnode != NULL)
    {
        prevNode = stnode;
        curNode = stnode->nextptr;
        stnode = stnode->nextptr;

        prevNode->nextptr = NULL; //convert the first node as last

        while(stnode != NULL)
        {
            stnode = stnode->nextptr;
            curNode->nextptr = prevNode;
```

```
                prevNode = curNode;
                curNode = stnode;
            }
            stnode = prevNode; //convert the last node as head
        }
    }
```

*Priority Queue

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int priority;
    char *data;
} node_t;

typedef struct {
    node_t *nodes;
    int len;
    int size;
} heap_t;

void push (heap_t *h, int priority, char *data) {
    if (h->len + 1 >= h->size) {
        h->size = h->size ? h->size * 2 : 4;
        h->nodes = (node_t *)realloc(h->nodes, h->size * sizeof (node_t));
    }
    int i = h->len + 1;
    int j = i / 2;
    while (i > 1 && h->nodes[j].priority > priority) {
        h->nodes[i] = h->nodes[j];
        i = j;
        j = j / 2;
    }
    h->nodes[i].priority = priority;
    h->nodes[i].data = data;
    h->len++;
}

char *pop (heap_t *h) {
    int i, j, k;
    if (!h->len) {
        return NULL;
    }
    char *data = h->nodes[1].data;

    h->nodes[1] = h->nodes[h->len];
    int priority = h->nodes[1].priority;

    h->len--;

    i = 1;
    while (1) {
```

```
        k = i;
        j = 2 * i;

        if (j <= h->len && h->nodes[j].priority < priority) {
            k = j;
        }
        if (j + 1 <= h->len && h->nodes[j + 1].priority < h->nodes[k].priority) {
            k = j + 1;
        }
        if (k == i) {
            break;
        }
        h->nodes[i] = h->nodes[k];
        i = k;
    }
    h->nodes[i] = h->nodes[h->len + 1];
    return data;
}

int main () {
    heap_t *h = (heap_t *)calloc(1, sizeof (heap_t));
    push(h, 3, "Clear drains");
    push(h, 4, "Feed cat");
    push(h, 5, "Make tea");
    push(h, 1, "Solve RC tasks");
    push(h, 2, "Tax return");
    int i;
    for (i = 0; i < 5; i++) {
        printf("%s\n", pop(h));
    }
    return 0;
}
```

*Binary search tree

```
// C program to demonstrate insert operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
```

```c
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d \n", root->key);
        inorder(root->right);
    }
}
```

```c
/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```

```c
/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}
```

```c
/* Given a binary search tree and a key, this function deletes the key
   and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);
```

```
    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // node with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

**Construct BST from preorder

```
// A recursive function to construct Full from pre[]. preIndex is used
// to keep track of index in pre[].
struct node* constructTreeUtil (int pre[], int* preIndex,
                                int low, int high, int size)
{
    // Base case
    if (*preIndex >= size || low > high)
        return NULL;

    // The first node in preorder traversal is root. So take the node at
    // preIndex from pre[] and make it root, and increment preIndex
    struct node* root = newNode ( pre[*preIndex] );
    *preIndex = *preIndex + 1;

    // If the current subarry has only one element, no need to recur
```

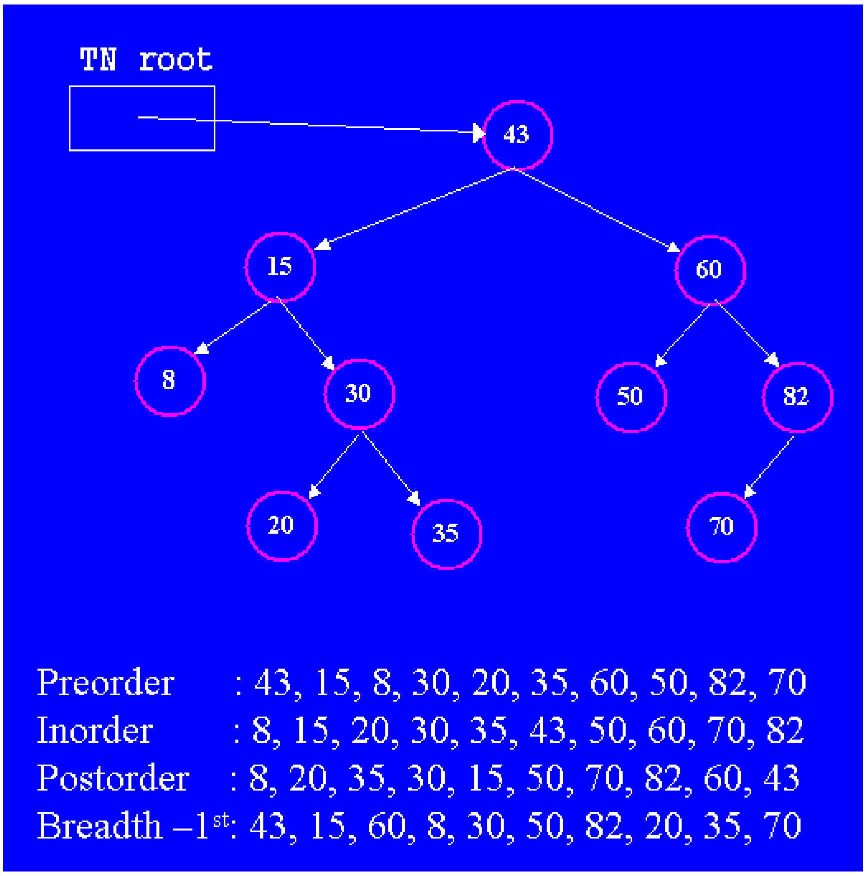
```
    if (low == high)
        return root;

    // Search for the first element greater than root
    int i;
    for ( i = low; i <= high; ++i )
        if ( pre[ i ] > root->data )
            break;

    // Use the index of element found in preorder to divide preorder array in
    // two parts. Left subtree and right subtree
    root->left = constructTreeUtil ( pre, preIndex, *preIndex, i - 1, size );
    root->right = constructTreeUtil ( pre, preIndex, i, high, size );

    return root;
}

// The main function to construct BST from given preorder traversal.
// This function mainly uses constructTreeUtil()
struct node *constructTree (int pre[], int size)
{
    int preIndex = 0;
    return constructTreeUtil (pre, &preIndex, 0, size - 1, size);
}
```

*USING STACK construct BST

```
/* A binary tree node has data, pointer to left child
   and a pointer to right child */
typedef struct Node
{
    int data;
    struct Node *left, *right;
} Node;

// A Stack has array of Nodes, capacity, and top
typedef struct Stack
{
    int top;
    int capacity;
    Node* *array;
} Stack;

// A utility function to create a new tree node
Node* newNode( int data )
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to create a stack of given capacity
Stack* createStack( int capacity )
{
    Stack* stack = (Stack *)malloc( sizeof( Stack ) );
    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (Node **)malloc( stack->capacity * sizeof( Node* ) );
```

```
    return stack;
}

// A utility function to check if stack is full
int isFull( Stack* stack )
{
    return stack->top == stack->capacity - 1;
}

// A utility function to check if stack is empty
int isEmpty( Stack* stack )
{
    return stack->top == -1;
}

// A utility function to push an item to stack
void push( Stack* stack, Node* item )
{
    if( isFull( stack ) )
        return;
    stack->array[ ++stack->top ] = item;
}

// A utility function to remove an item from stack
Node* pop( Stack* stack )
{
    if( isEmpty( stack ) )
        return NULL;
    return stack->array[ stack->top-- ];
}

// A utility function to get top node of stack
Node* peek( Stack* stack )
{
    return stack->array[ stack->top ];
}

// The main function that constructs BST from pre[]
Node* constructTree ( int pre[], int size )
{
    // Create a stack of capacity equal to size
    Stack* stack = createStack( size );

    // The first element of pre[] is always root
    Node* root = newNode( pre[0] );

    // Push root
    push( stack, root );

    int i;
    Node* temp;

    // Iterate through rest of the size-1 items of given preorder array
    for ( i = 1; i < size; ++i )
    {
        temp = NULL;

        /* Keep on popping while the next value is greater than
           stack's top value. */
        while ( !isEmpty( stack ) && pre[i] > peek( stack )->data )
            temp = pop( stack );

        // Make this greater value as the right child and push it to the stack
        if ( temp != NULL)
        {
            temp->right = newNode( pre[i] );
```

```
            push( stack, temp->right );
        }

        // If the next value is less than the stack's top value, make this value
        // as the left child of the stack's top node. Push the new node to stack
        else
        {
            peek( stack )->left = newNode( pre[i] );
            push( stack, peek( stack )->left );
        }
    }

    return root;
}
```




```
// A recursive function that traverses the given BST in reverse inorder and
// for every key, adds all greater keys to it
void addGreaterUtil(struct node *root, int *sum_ptr)
{
    // Base Case
    if (root == NULL)
        return;

    // Recur for right subtree first so that sum of all greater
    // nodes is stored at sum_ptr
    addGreaterUtil(root->right, sum_ptr);

    // Update the value at sum_ptr
    *sum_ptr = *sum_ptr + root->key;

    // Update key of this node
```

```
    root->key = *sum_ptr;

    // Recur for left subtree so that the updated sum is added
    // to smaller nodes
    addGreaterUtil(root->left, sum_ptr);
}

// A wrapper over addGreaterUtil().  It initializes sum and calls
// addGreaterUtil() to recursivel upodate and use value of sum
void addGreater(struct node *root)
{
    int sum = 0;
    addGreaterUtil(root, &sum);
}
```
==

**Linked list to BST

```
/* This function counts the number of nodes in Linked List and then calls
   sortedListToBSTRecur() to construct BST */
struct TNode* sortedListToBST(struct LNode *head)
{
    /*Count the number of nodes in Linked List */
    int n = countLNodes(head);

    /* Construct BST */
    return sortedListToBSTRecur(&head, n);
}

/* The main function that constructs balanced BST and returns root of it.
       head_ref -->  Pointer to pointer to head node of linked list
       n  --> No. of nodes in Linked List */
struct TNode* sortedListToBSTRecur(struct LNode **head_ref, int n)
{
    /* Base Case */
    if (n <= 0)
        return NULL;

    /* Recursively construct the left subtree */
    struct TNode *left = sortedListToBSTRecur(head_ref, n/2);

    /* Allocate memory for root, and link the above constructed left
       subtree with root */
    struct TNode *root = newNode((*head_ref)->data);
    root->left = left;

    /* Change head pointer of Linked List for parent recursive calls */
    *head_ref = (*head_ref)->next;

    /* Recursively construct the right subtree and link it with root
      The number of nodes in right subtree  is total nodes - nodes in
      left subtree - 1 (for root) which is n-n/2-1*/
    root->right = sortedListToBSTRecur(head_ref, n-n/2-1);

    return root;
}
```

===

Transform a BST to greater sum tree

```
// Recursive function to transform a BST to sum tree.
// This function traverses the tree in reverse inorder so
// that we have visited all greater key nodes of the currently
// visited node
void transformTreeUtil(struct Node *root, int *sum)
{
   // Base case
   if (root == NULL)  return;

   // Recur for right subtree
   transformTreeUtil(root->right, sum);

   // Update sum
   *sum = *sum + root->data;

   // Store old sum in current node
   root->data = *sum - root->data;

   // Recur for left subtree
   transformTreeUtil(root->left, sum);
}

// A wrapper over transformTreeUtil()
void transformTree(struct Node *root)
{
    int sum = 0; // Initialize sum
    transformTreeUtil(root, &sum);
}
```

==

**BST to linked list

```
// A simple recursive function to convert a given
// Binary Search tree to Sorted Linked List
// root     --> Root of Binary Search Tree
// head_ref --> Pointer to head node of created
//              linked list
void BSTToSortedLL(Node* root, Node** head_ref)
{
    // Base cases
    if(root == NULL)
        return;

    // Recursively convert right subtree
    BSTToSortedLL(root->right, head_ref);

    // insert root into linked list
    root->right = *head_ref;

    // Change left pointer of previous head
    // to point to NULL
    if (*head_ref != NULL)
        (*head_ref)->left = NULL;

    // Change head of linked list
    *head_ref = root;
```

```
    // Recursively convert left subtree
    BSTToSortedLL(root->left, head_ref);
}
```
==

**BST to minheap

```
void SortedLLToMinHeap(Node* &root, Node* head)
{
    // Base Case
    if (head == NULL)
        return;

    // queue to store the parent nodes
    queue<Node *> q;

    // The first node is always the root node
    root = head;

    // advance the pointer to the next node
    head = head->right;

    // set right child to NULL
    root->right = NULL;

    // add first node to the queue
    q.push(root);

    // run until the end of linked list is reached
    while (head)
    {
        // Take the parent node from the q and remove it from q
        Node* parent = q.front();
        q.pop();

        // Take next two nodes from the linked list and
        // Add them as children of the current parent node
        // Also in push them into the queue so that
        // they will be parents to the future nodes
        Node *leftChild = head;
        head = head->right;        // advance linked list to next node
        leftChild->right = NULL; // set its right child to NULL
        q.push(leftChild);

        // Assign the left child of parent
        parent->left = leftChild;

        if (head)
        {
            Node *rightChild = head;
            head = head->right; // advance linked list to next node
            rightChild->right = NULL; // set its right child to NULL
            q.push(rightChild);

            // Assign the right child of parent
            parent->right = rightChild;
        }
    }
}
```

```
// Function to convert BST into a Min-Heap
// without using any extra space
Node* BSTToMinHeap(Node* &root)
{
    // head of Linked List
    Node *head = NULL;

    // Convert a given BST to Sorted Linked List
    BSTToSortedLL(root, &head);

    // set root as NULL
    root = NULL;

    // Convert Sorted Linked List to Min-Heap
    SortedLLToMinHeap(root, head);
}
```

===

**Check BST ???

```
int isBST(struct node* node)
{
  if (node == NULL)
    return 1;

  /* false if left is > than node */
  if (node->left != NULL && node->left->data > node->data)
    return 0;

  /* false if right is < than node */
  if (node->right != NULL && node->right->data < node->data)
    return 0;

  /* false if, recursively, the left or right is not a BST */
  if (!isBST(node->left) || !isBST(node->right))
    return 0;

  /* passing all that, it's a BST */
  return 1;
}
```

===

K’th Largest Element in BST when modification to BST is not allowed

```
// A function to find k'th largest element in a given tree.
void kthLargestUtil(Node *root, int k, int &c)
{
    // Base cases, the second condition is important to
    // avoid unnecessary recursive calls
    if (root == NULL || c >= k)
        return;

    // Follow reverse inorder traversal so that the
    // largest element is visited first
    kthLargestUtil(root->right, k, c);
```

```
    // Increment count of visited nodes
    c++;

    // If c becomes k now, then this is the k'th largest
    if (c == k)
    {
        cout << "K'th largest element is "
             << root->key << endl;
        return;
    }

    // Recur for left subtree
    kthLargestUtil(root->left, k, c);
}

// Function to find k'th largest element
void kthLargest(Node *root, int k)
{
    // Initialize count of nodes visited as 0
    int c = 0;

    // Note that c is passed by reference
    kthLargestUtil(root, k, c);
}
```

===

K’th smallest element in BST using O(1) Extra Space morris

```
// A function to find
int KSmallestUsingMorris(Node *root, int k)
{
    // Count to iterate over elements till we
    // get the kth smallest number
    int count = 0;

    int ksmall = INT_MIN; // store the Kth smallest
    Node *curr = root; // to store the current node

    while (curr != NULL)
    {
        // Like Morris traversal if current does
        // not have left child rather than printing
        // as we did in inorder, we will just
        // increment the count as the number will
        // be in an increasing order
        if (curr->left == NULL)
        {
            count++;

            // if count is equal to K then we found the
            // kth smallest, so store it in ksmall
            if (count==k)
                ksmall = curr->key;

            // go to current's right child
            curr = curr->right;
```

```
        }
        else
        {
            // we create links to Inorder Successor and
            // count using these links
            Node *pre = curr->left;
            while (pre->right != NULL && pre->right != curr)
                pre = pre->right;

            // building links
            if (pre->right==NULL)
            {
                //link made to Inorder Successor
                pre->right = curr;
                curr = curr->left;
            }

            // While breaking the links in so made temporary
            // threaded tree we will check for the K smallest
            // condition
            else
            {
                // Revert the changes made in if part (break link
                // from the Inorder Successor)
                pre->right = NULL;

                count++;

                // If count is equal to K then we found
                // the kth smallest and so store it in ksmall
                if (count==k)
                    ksmall = curr->key;

                curr = curr->right;
            }
        }
    }
    return ksmall; //return the found value
}
```
==

Largest number in BST which is less than or equal to N

```
// function to find max value less then N
int findMaxforN(Node* root, int N)
{
    /* If leaf node reached and is greater than N*/
    if (root->left == NULL && root->right == NULL &&
        root->key > N)
        return -1;

    /* If node's value is less than N and right value
       is NULL or grater than then return the node
       value*/
    if ((root->key <= N && root->right == NULL) ||
        (root->key <= N && root->right->key > N))
        return root->key;
```

```
    // if node value is grater than N search in the
    // left subtree
    if (root->key >= N)
        return findMaxforN(root->left, N);

    // if node value is less than N search in the
    // right subtree
    else
        return findMaxforN(root->right, N);
}
```

==

Merge Two Balanced Binary Search Trees

```
/* This function merges two balanced BSTs with roots as root1 and root2.
   m and n are the sizes of the trees respectively */
struct node* mergeTrees(struct node *root1, struct node *root2, int m, int n)
{
    // Store inorder traversal of first tree in an array arr1[]
    int *arr1 = new int[m];
    int i = 0;
    storeInorder(root1, arr1, &i);

    // Store inorder traversal of second tree in another array arr2[]
    int *arr2 = new int[n];
    int j = 0;
    storeInorder(root2, arr2, &j);

    // Merge the two sorted array into one
    int *mergedArr = merge(arr1, arr2, m, n);

    // Construct a tree from the merged array and return root of the tree
    return sortedArrayToBST (mergedArr, 0, m+n-1);
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

// A utility function to print inorder traversal of a given binary tree
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    printf("%d ", node->data);
```

```
    /* now recur on right child */
    printInorder(node->right);
}

// A utility unction to merge two sorted arrays into one
int *merge(int arr1[], int arr2[], int m, int n)
{
    // mergedArr[] is going to contain result
    int *mergedArr = new int[m + n];
    int i = 0, j = 0, k = 0;

    // Traverse through both arrays
    while (i < m && j < n)
    {
        // Pick the smaler element and put it in mergedArr
        if (arr1[i] < arr2[j])
        {
            mergedArr[k] = arr1[i];
            i++;
        }
        else
        {
            mergedArr[k] = arr2[j];
            j++;
        }
        k++;
    }

    // If there are more elements in first array
    while (i < m)
    {
        mergedArr[k] = arr1[i];
        i++; k++;
    }

    // If there are more elements in second array
    while (j < n)
    {
        mergedArr[k] = arr2[j];
        j++; k++;
    }

    return mergedArr;
}

// A helper function that stores inorder traversal of a tree rooted with node
void storeInorder(struct node* node, int inorder[], int *index_ptr)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    storeInorder(node->left, inorder, index_ptr);

    inorder[*index_ptr] = node->data;
    (*index_ptr)++;  // increase index for next entry

    /* now recur on right child */
    storeInorder(node->right, inorder, index_ptr);
```

```
}

/* A function that constructs Balanced Binary Search Tree from a sorted array
   See https://www.geeksforgeeks.org/archives/17138 */
struct node* sortedArrayToBST(int arr[], int start, int end)
{
    /* Base Case */
    if (start > end)
      return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    struct node *root = newNode(arr[mid]);

    /* Recursively construct the left subtree and make it
       left child of root */
    root->left =  sortedArrayToBST(arr, start, mid-1);

    /* Recursively construct the right subtree and make it
       right child of root */
    root->right = sortedArrayToBST(arr, mid+1, end);

    return root;
}
```

==

Two nodes of a BST are swapped, correct the BST

```
// This function does inorder traversal to find out the two swapped nodes.
// It sets three pointers, first, middle and last.  If the swapped nodes are
// adjacent to each other, then first and middle contain the resultant nodes
// Else, first and last contain the resultant nodes
void correctBSTUtil( struct node* root, struct node** first,
                     struct node** middle, struct node** last,
                     struct node** prev )
{
    if( root )
    {
        // Recur for the left subtree
        correctBSTUtil( root->left, first, middle, last, prev );

        // If this node is smaller than the previous node, it's violating
        // the BST rule.
        if (*prev && root->data < (*prev)->data)
        {
            // If this is first violation, mark these two nodes as
            // 'first' and 'middle'
            if ( !*first )
            {
                *first = *prev;
                *middle = root;
            }

            // If this is second violation, mark this node as last
            else
                *last = root;
```

```
        }

        // Mark this node as previous
        *prev = root;

        // Recur for the right subtree
        correctBSTUtil( root->right, first, middle, last, prev );
    }
}

// A function to fix a given BST where two nodes are swapped.  This
// function uses correctBSTUtil() to find out two nodes and swaps the
// nodes to fix the BST
void correctBST( struct node* root )
{
    // Initialize pointers needed for correctBSTUtil()
    struct node *first, *middle, *last, *prev;
    first = middle = last = prev = NULL;

    // Set the poiters to find out two nodes
    correctBSTUtil( root, &first, &middle, &last, &prev );

    // Fix (or correct) the tree
    if( first && last )
        swap( &(first->data), &(last->data) );
    else if( first && middle ) // Adjacent nodes swapped
        swap( &(first->data), &(middle->data) );

    // else nodes have not been swapped, passed tree is really BST.
}
```
==

How to handle duplicates in Binary Search Tree?

```
// C program to implement basic operations (search, insert and delete)
// on a BST that handles duplicates by storing count with every node
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    int count;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    temp->count = 1;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
```

```
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d(%d) ", root->key, root->count);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    // If key already exists in BST, icnrement count and return
    if (key == node->key)
    {
       (node->count)++;
        return node;
    }

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the node with
   minimum key value found in that tree. Note that the entire
   tree does not need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

/* Given a binary search tree and a key, this function
   deletes a given key and returns root of modified tree */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the
    // root's key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
```

```
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key
    else
    {
        // If key is present more than once, simply decrement
        // count and return
        if (root->count > 1)
        {
           (root->count)--;
           return root;
        }

        // ElSE, delete the node

        // node with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

==

Inorder predecessor and successor for a given key in BST

```
// This function finds predecessor and successor of key in BST.
// It sets pre and suc as predecessor and successor respectively
void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{
    // Base case
    if (root == NULL)  return ;

    // If key is present at root
```

```
    if (root->key == key)
    {
        // the maximum value in left subtree is predecessor
        if (root->left != NULL)
        {
            Node* tmp = root->left;
            while (tmp->right)
                tmp = tmp->right;
            pre = tmp ;
        }

        // the minimum value in right subtree is successor
        if (root->right != NULL)
        {
            Node* tmp = root->right ;
            while (tmp->left)
                tmp = tmp->left ;
            suc = tmp ;
        }
        return ;
    }

    // If key is smaller than root's key, go to left subtree
    if (root->key > key)
    {
        suc = root ;
        findPreSuc(root->left, pre, suc, key) ;
    }
    else // go to right subtree
    {
        pre = root ;
        findPreSuc(root->right, pre, suc, key) ;
    }
}
```

===

Find the closest element in Binary Search Tree

```
// Function to find node with minimum absolute
// difference with given K
// min_diff   --> minimum difference till now
// min_diff_key  --> node having minimum absolute
//                    difference with K
void maxDiffUtil(struct Node *ptr, int k, int &min_diff,
                                      int &min_diff_key)
{
    if (ptr == NULL)
        return ;

    // If k itself is present
    if (ptr->key == k)
    {
        min_diff_key = k;
        return;
    }
```

```
    // update min_diff and min_diff_key by checking
    // current node value
    if (min_diff > abs(ptr->key - k))
    {
        min_diff = abs(ptr->key - k);
        min_diff_key = ptr->key;
    }

    // if k is less than ptr->key then move in
    // left subtree else in right subtree
    if (k < ptr->key)
        maxDiffUtil(ptr->left, k, min_diff, min_diff_key);
    else
        maxDiffUtil(ptr->right, k, min_diff, min_diff_key);
}

// Wrapper over maxDiffUtil()
int maxDiff(Node *root, int k)
{
    // Initialize minimum difference
    int min_diff = INT_MAX, min_diff_key = -1;

    // Find value of min_diff_key (Closest key
    // in tree with k)
    maxDiffUtil(root, k, min_diff, min_diff_key);

    return min_diff_key;
}
=================================================================================
```

Sum of k smallest elements in BST

```
// function return sum of all element smaller than
// and equal to Kth smallest element
int ksmallestElementSumRec(Node *root, int k, int &count)
{
    // Base cases
    if (root == NULL)
        return 0;
    if (count > k)
        return 0;

    // Compute sum of elements in left subtree
    int res = ksmallestElementSumRec(root->left, k, count);
    if (count >= k)
        return res;

    // Add root's data
    res += root->data;

    // Add current Node
    count++;
    if (count >= k)
      return res;

    // If count is less than k, return right subtree Nodes
    return res + ksmallestElementSumRec(root->right, k, count);
```

```
}

// Wrapper over ksmallestElementSumRec()
int ksmallestElementSum(struct Node *root, int k)
{
   int count = 0;
   ksmallestElementSumRec(root, k, count);
}
```
==

Print Common Nodes in Two Binary Search Trees

```
// Function two print common elements in given two trees
void printCommon(Node *root1, Node *root2)
{
    // Create two stacks for two inorder traversals
    stack<Node *> stack1, s1, s2;

    while (1)
    {
        // push the Nodes of first tree in stack s1
        if (root1)
        {
            s1.push(root1);
            root1 = root1->left;
        }

        // push the Nodes of second tree in stack s2
        else if (root2)
        {
            s2.push(root2);
            root2 = root2->left;
        }

        // Both root1 and root2 are NULL here
        else if (!s1.empty() && !s2.empty())
        {
            root1 = s1.top();
            root2 = s2.top();

            // If current keys in two trees are same
            if (root1->key == root2->key)
            {
                cout << root1->key << " ";
                s1.pop();
                s2.pop();

                // move to the inorder successor
                root1 = root1->right;
                root2 = root2->right;
            }

            else if (root1->key < root2->key)
```

```
            {
                // If Node of first tree is smaller, than that of
                // second tree, then its obvious that the inorder
                // successors of current Node can have same value
                // as that of the second tree Node. Thus, we pop
                // from s2
                s1.pop();
                root1 = root1->right;

                // root2 is set to NULL, because we need
                // new Nodes of tree 1
                root2 = NULL;
            }
            else if (root1->key > root2->key)
            {
                s2.pop();
                root2 = root2->right;
                root1 = NULL;
            }
        }

        // Both roots and both stacks are empty
        else  break;
    }
}
```

===

Count BST nodes that lie in a given range

```
// Returns count of nodes in BST in range [low, high]
int getCount(node *root, int low, int high)
{
    // Base case
    if (!root) return 0;

    // Special Optional case for improving efficiency
    if (root->data == high && root->data == low)
        return 1;

    // If current node is in range, then include it in count and
    // recur for left and right children of it
    if (root->data <= high && root->data >= low)
         return 1 + getCount(root->left, low, high) +
                    getCount(root->right, low, high);

    // If current node is smaller than low, then recur for right
    // child
    else if (root->data < low)
         return getCount(root->right, low, high);

    // Else recur for left child
    else return getCount(root->left, low, high);
}
```

===

```
// Delete leaf nodes from binary search tree.
struct Node* leafDelete(struct Node* root)
```

```
{
    if (root->left == NULL && root->right == NULL) {
        free(root);
        return NULL;
    }

    // Else recursively delete in left and right
    // subtrees.
    root->left = leafDelete(root->left);
    root->right = leafDelete(root->right);

    return root;
}
```
===

Shortest distance between two nodes in BST

```
// This function returns distance of x from
// root. This function assumes that x exists
// in BST and BST is not NULL.
int distanceFromRoot(struct Node* root, int x)
{
    if (root->key == x)
        return 0;
    else if (root->key > x)
        return 1 + distanceFromRoot(root->left, x);
    return 1 + distanceFromRoot(root->right, x);
}

// Returns minimum distance beween a and b.
// This function assumes that a and b exist
// in BST.
int distanceBetween2(struct Node* root, int a, int b)
{
    if (!root)
        return 0;

    // Both keys lie in left
    if (root->key > a && root->key > b)
        return distanceBetween2(root->left, a, b);

    // Both keys lie in right
    if (root->key < a && root->key < b) // same path
        return distanceBetween2(root->right, a, b);

    // Lie in opposite directions (Root is
    // LCA of two nodes)
    if (root->key >= a && root->key <= b)
        return distanceFromRoot(root, a) +
               distanceFromRoot(root, b);
}

// This function make sure that a is smaller
// than b before making a call to findDistWrapper()
int findDistWrapper(Node *root, int a, int b)
{
```

```
   if (a > b)
     swap(a, b);
   return distanceBetween2(root, a, b);
}
```

===

LINKED LIST

===

```
ref getNode(PHANSO k)
{
	ref p;
	p = (ref)malloc(sizeof(Node));
	if (p == NULL)
	{
		printf("Khong du bo nho!\n");
		exit(0);
	}
	p->key = k;
	p->next = NULL;
	return p;
}

void addFirst(ref &head, ref &tail, PHANSO k)
{
	ref p = getNode(k);
	if (head == NULL)
	{
		head = tail = p;
	}
	else
	{
		p->next = head;
		head = p;
	}
}
```

```
void PrintList(ref head)
{
	ref p;
	if (head == NULL)
	{
		printf("\nDanh sach rong!!!");
	}
	else
	{
		for (p = head; p; p = p->next)
		{
			printf("%d/%d -> ", p->key.tuso, p->key.mauso);
		}
		printf("NULL\n");
	}
}

void addLast(ref &head, ref &tail, PHANSO k)
{
	ref p = getNode(k);
	if (head == NULL)
	{
		head = tail = p;
	}
	else
	{
		tail->next = p;
		tail = p;
	}
}

int length(ref head)
{
	int count = 0;
	ref p;
	for (p = head; p; p = p->next)
	{
		count++;
	}
	return count;
}

void insertBefore(ref q, PHANSO k)
{
	ref p;
	p = (ref)malloc(sizeof(Node));
	if (p == NULL)
	{
		printf("Loi khong du bo nho\n");
		exit(0);
	}
	else
	{
		*p = *q;
		q->next = p;
```

```
            q->key = k;
        }
}

void insertAfter(ref q, PHANSO k)
{
        ref p;
        p = getNode(k);
        p->next = q->next;
        q->next = p;
}

void insertAt(ref &head, ref &tail, int pos, PHANSO k)
{
        int n, i;
        ref q;
        n = length(head);
        if (pos<0 || pos>n)
        {
            printf("Vi tri chen khong phu hop!\n");
            return;
        }

        if (pos == 0) addFirst(head, tail, k);
        else if (pos == n) addLast(head, tail, k);
        else
        {
            for (i = 0, q = head; i < pos; i++, q = q->next);
            insertBefore(q, k);
            if (tail->next)
                  tail = tail->next;
        }
}

void deleteBegin(ref &head, ref &tail)
{
        ref q;
        if (head == tail)
        {
            free(head);
            head = tail = NULL;
        }
        else
        {
            q = head;
            head = head->next;
            free(q);
        }
}

void deleteEnd(ref &head, ref &tail)
{
        ref q;
        if (head == tail)
        {
```

```
		free(head);
		head = tail = NULL;
	}
	else
	{
		for (q = head; q->next != tail; q = q->next);
		free(tail);
		tail = q;
		q->next = NULL;
	}
}

void deleteMiddle(ref q)
{
	ref r;
	r = q->next;
	*q = *r;
	free(r);
}

void deleteAt(ref &head, ref &tail, int pos)
{
	int n, i;
	ref q;
	n = length(head);
	if (pos < 0 || pos >= n)
	{
		printf("Vi tri xoa khong phu hop!\n");
		return;
	}

	if (pos == 0) deleteBegin(head, tail);
	else if (pos == n - 1) deleteEnd(head, tail);
	else
	{
		for (i = 0, q = head; i < pos; i++, q = q->next);
		if (q->next == tail) tail = q;
		deleteMiddle(q);
	}
}

void destroyList(ref &head)
{
	ref p;
	while (head)
	{
		p = head;
		head = head->next;
		free(p);
	}
}
```
===

```
void deleteNode(struct Node **head_ref, int key)
{
    // Store head node
    struct Node* temp = *head_ref, *prev;

    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next;   // Changed head
        free(temp);               // free old head
        return;
    }

    // Search for the key to be deleted, keep track of the
    // previous node as we need to change 'prev->next'
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in linked list
    if (temp == NULL) return;

    // Unlink the node from linked list
    prev->next = temp->next;

    free(temp);  // Free memory
}
==================================================================================
```

Swap nodes in a linked list without swapping data

```
/* Function to swap nodes x and y in linked list by
   changing links */
void swapNodes(struct Node **head_ref, int x, int y)
{
   // Nothing to do if x and y are same
   if (x == y) return;

   // Search for x (keep track of prevX and CurrX
   struct Node *prevX = NULL, *currX = *head_ref;
   while (currX && currX->data != x)
   {
       prevX = currX;
       currX = currX->next;
   }

   // Search for y (keep track of prevY and CurrY
   struct Node *prevY = NULL, *currY = *head_ref;
   while (currY && currY->data != y)
   {
       prevY = currY;
```

```
        currY = currY->next;
    }

    // If either x or y is not present, nothing to do
    if (currX == NULL || currY == NULL)
        return;

    // If x is not head of linked list
    if (prevX != NULL)
        prevX->next = currY;
    else // Else make y as new head
        *head_ref = currY;

    // If y is not head of linked list
    if (prevY != NULL)
        prevY->next = currX;
    else  // Else make x as new head
        *head_ref = currX;

    // Swap next pointers
    struct Node *temp = currY->next;
    currY->next = currX->next;
    currX->next  = temp;
}
```

===

Reverse a linked list

```
static void reverse(struct Node** head_ref)
{
    struct Node* prev   = NULL;
    struct Node* current = *head_ref;
    struct Node* next;
    while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

/* Function to reverse the linked list */
void printReverse(struct Node* head)
{
    // Base case
    if (head == NULL)
       return;

    // print the list after head node
    printReverse(head->next);

    // After everything else is printed, print head
    printf("%d  ", head->data);
}
```

===

Remove duplicates from a sorted linked list

```
/* The function removes duplicates from a sorted list */
void removeDuplicates(struct Node* head)
{
    /* Pointer to traverse the linked list */
    struct Node* current = head;

    /* Pointer to store the next pointer of a node to be deleted*/
    struct Node* next_next;

    /* do nothing if the list is empty */
    if (current == NULL)
       return;

    /* Traverse the list till last node */
    while (current->next != NULL)
    {
       /* Compare current node with next node */
       if (current->data == current->next->data)
       {
           /* The sequence of steps is important*/
           next_next = current->next->next;
           free(current->next);
           current->next = next_next;
       }
       else /* This is tricky: only advance if no deletion */
       {
          current = current->next;
       }
    }
}
==================================================================================
```

Remove duplicates from an unsorted linked list

```
/* Function to remove duplicates from a
   unsorted linked list */
void removeDuplicates(struct Node *start)
{
    struct Node *ptr1, *ptr2, *dup;
    ptr1 = start;

    /* Pick elements one by one */
    while (ptr1 != NULL && ptr1->next != NULL)
    {
        ptr2 = ptr1;

        /* Compare the picked element with rest
           of the elements */
        while (ptr2->next != NULL)
        {
```

```
            /* If duplicate then delete it */
            if (ptr1->data == ptr2->next->data)
            {
                /* sequence of steps is important here */
                dup = ptr2->next;
                ptr2->next = ptr2->next->next;
                delete(dup);
            }
            else /* This is tricky */
                ptr2 = ptr2->next;
        }
        ptr1 = ptr1->next;
    }
}
```

===

Insertion Sort for Singly Linked List

```
// function to sort a singly linked list using insertion sort
void insertionSort(struct Node **head_ref)
{
    // Initialize sorted linked list
    struct Node *sorted = NULL;

    // Traverse the given linked list and insert every
    // node to sorted
    struct Node *current = *head_ref;
    while (current != NULL)
    {
        // Store next for next iteration
        struct Node *next = current->next;

        // insert current in sorted linked list
        sortedInsert(&sorted, current);

        // Update current
        current = next;
    }

    // Update head_ref to point to sorted linked list
    *head_ref = sorted;
}

/* function to insert a new_node in a list. Note that this
  function expects a pointer to head_ref as this can modify the
  head of the input linked list (similar to push())*/
void sortedInsert(struct Node** head_ref, struct Node* new_node)
{
    struct Node* current;
    /* Special case for the head end */
    if (*head_ref == NULL || (*head_ref)->data >= new_node->data)
    {
        new_node->next = *head_ref;
        *head_ref = new_node;
    }
```

```
    else
    {
        /* Locate the node before the point of insertion */
        current = *head_ref;
        while (current->next!=NULL && current->next->data < new_node->data)
        {
            current = current->next;
        }
        new_node->next = current->next;
        current->next = new_node;
    }
}
```

===

We have discussed different solutions of this problem (here and here). In this post a simple circular linked list based solution is discussed.

1) Create a circular linked list of size n.

2) Traverse through linked list and one by one delete every m-th node until there is one node left.

3) Return value of the only left node.

```
/* Function to find the only person left
   after one in every m-th node is killed
   in a circle of n nodes */
void getJosephusPosition(int m, int n)
{
    // Create a circular linked list of
    // size N.
    Node *head = newNode(1);
    Node *prev = head;
    for (int i = 2; i <= n; i++)
    {
        prev->next = newNode(i);
        prev = prev->next;
    }
    prev->next = head; // Connect last
                       // node to first

    /* while only one node is left in the
    linked list*/
    Node *ptr1 = head, *ptr2 = head;
    while (ptr1->next != ptr1)
    {
        // Find m-th node
        int count = 1;
        while (count != m)
        {
            ptr2 = ptr1;
            ptr1 = ptr1->next;
            count++;
        }

        /* Remove the m-th node */
        ptr2->next = ptr1->next;
        ptr1 = ptr2->next;
```

```
    }

    printf ("Last person left standing "
            "(Josephus Position) is %d\n ",
            ptr1->data);
}
```

==

Reverse a Doubly Linked List

```
/* a node of the doubly linked list */
struct Node
{
  int data;
  struct Node *next;
  struct Node *prev;
};

/* Function to reverse a Doubly Linked List */
void reverse(struct Node **head_ref)
{
     struct Node *temp = NULL;
     struct Node *current = *head_ref;

     /* swap next and prev for all nodes of
       doubly linked list */
     while (current !=  NULL)
     {
       temp = current->prev;
       current->prev = current->next;
       current->next = temp;
       current = current->prev;
     }

     /* Before changing head, check for the cases like empty
        list and list with only one node */
     if(temp != NULL )
        *head_ref = temp->prev;
}
```

==

Convert a given Binary Tree to Doubly Linked List

```
// A simple recursive function to convert a given
// Binary tree to Doubly Linked List
// root     --> Root of Binary Tree
// head_ref --> Pointer to head node of created
//              doubly linked list
void BToDLL(Node* root, Node** head_ref)
{
    // Base cases
    if (root == NULL)
        return;

    // Recursively convert right subtree
    BToDLL(root->right, head_ref);
```

```
    // insert root into DLL
    root->right = *head_ref;

    // Change left pointer of previous head
    if (*head_ref != NULL)
        (*head_ref)->left = root;

    // Change head of Doubly linked list
    *head_ref = root;

    // Recursively convert left subtree
    BToDLL(root->left, head_ref);
}
```

===

=== Remove duplicates from a sorted doubly linked list

```
/* Function to delete a node in a Doubly Linked List.
   head_ref --> pointer to head node pointer.
   del  -->  pointer to node to be deleted. */
void deleteNode(struct Node** head_ref, struct Node* del)
{
    /* base case */
    if (*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if (*head_ref == del)
        *head_ref = del->next;

    /* Change next only if node to be deleted
        is NOT the last node */
    if (del->next != NULL)
        del->next->prev = del->prev;

    /* Change prev only if node to be deleted
       is NOT the first node */
    if (del->prev != NULL)
        del->prev->next = del->next;

    /* Finally, free the memory occupied by del*/
    free(del);
}

/* function to remove duplicates from a
   sorted doubly linked list */
void removeDuplicates(struct Node** head_ref)
{
    /* if list is empty */
    if ((*head_ref) == NULL)
        return;

    struct Node* current = *head_ref;
    struct Node* next;

    /* traverse the list till the last node */
    while (current->next != NULL) {
```

```
        /* Compare current node with next node */
        if (current->data == current->next->data)

            /* delete the node pointed to by
              'current->next' */
            deleteNode(head_ref, current->next);

        /* else simply move to the next node */
        else
            current = current->next;
    }
}
```

==
==

Doubly Linked List | Set 1 (Introduction and Insertion)

```
/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. Make next of new node as head and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    /* 4. change prev of head node to new node */
    if((*head_ref) !=  NULL)
      (*head_ref)->prev = new_node ;

    /* 5. move the head to point to the new node */
    (*head_ref)    = new_node;
}
```

==

Remove duplicates from an unsorted doubly linked list

```
/ Function to delete a node in a Doubly Linked List.
// head_ref --> pointer to head node pointer.
// del  -->  pointer to node to be deleted.
void deleteNode(struct Node** head_ref, struct Node* del)
{
    // base case
    if (*head_ref == NULL || del == NULL)
        return;

    // If node to be deleted is head node
    if (*head_ref == del)
        *head_ref = del->next;

    // Change next only if node to be deleted
```

```
    // is NOT the last node
    if (del->next != NULL)
        del->next->prev = del->prev;

    // Change prev only if node to be deleted
    // is NOT the first node
    if (del->prev != NULL)
        del->prev->next = del->next;

    // Finally, free the memory occupied by del
    free(del);
}

// function to remove duplicates from
// an unsorted doubly linked list
void removeDuplicates(struct Node** head_ref)
{
    // if DLL is empty or if it contains only
    // a single node
    if ((*head_ref) == NULL ||
        (*head_ref)->next == NULL)
        return;

    struct Node* ptr1, *ptr2;

    // pick elements one by one
    for (ptr1 = *head_ref; ptr1 != NULL; ptr1 = ptr1->next) {
        ptr2 = ptr1->next;

        // Compare the picked element with the
        // rest of the elements
        while (ptr2 != NULL) {

            // if duplicate, then delete it
            if (ptr1->data == ptr2->data) {

                // store pointer to the node next to 'ptr2'
                struct Node* next = ptr2->next;

                // delete node pointed to by 'ptr2'
                deleteNode(head_ref, ptr2);

                // update 'ptr2'
                ptr2 = next;
            }

            // else simply move to the next node
            else
                ptr2 = ptr2->next;
        }
    }
}
```

Performance [edit]

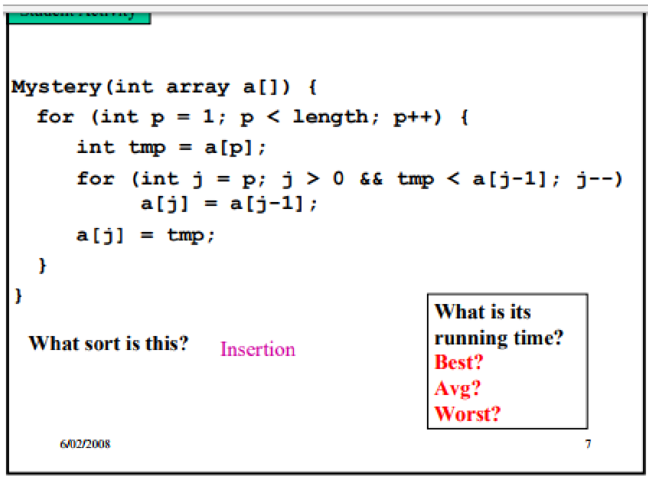
The only extra memory requirements are the auxiliary vector L for storing class bounds and the constant number of other variables used.

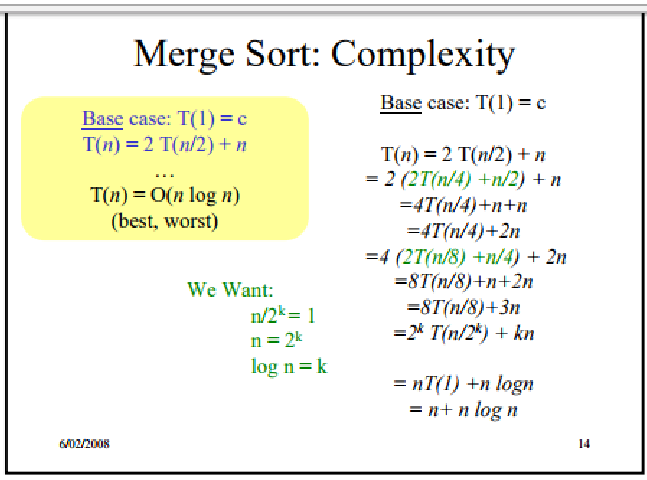
In the ideal case of a balanced data set, each class will be approximately the same size, and sorting an individual class by itself has complexity $O(1)$. If the number m of classes is proportional to the input set size n, the running time of the final insertion sort is $m \cdot O(1) = O(m) = O(n)$. In the worst-case scenarios where almost all the elements are in a few or one class, the complexity of the algorithm as a whole is limited by the performance of the final-step sorting method. For insertion sort, this is $O(n^2)$. Variations of the algorithm improve worst-case performance by using better-performing sorts such as quicksort or recursive flashsort on classes that exceed a certain size limit.[2][3]

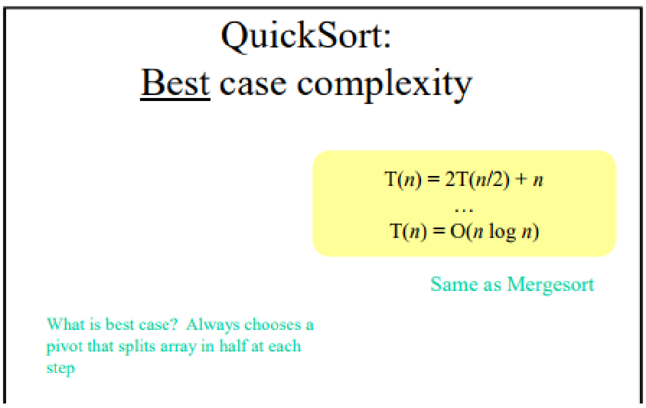
Choosing a value for m, the number of classes, trades off time spent classifying elements (high m) and time spent in the final insertion sort step (low m). Based on his research, Neubert found $m = 0.42n$ to be optimal.

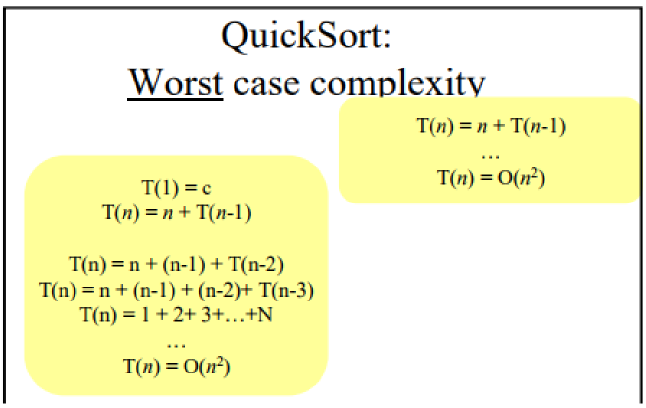
Memory-wise, flashsort avoids the overhead needed to store classes in the very similar bucketsort. For $m = 0.1n$ with uniform random data, flashsort is faster than heapsort for all n and faster than quicksort for $n > 80$. It becomes about as twice as fast as quicksort at $n = 10000$.[1]

Due to the *in situ* permutation that flashsort performs in its classification process, flashsort is not stable. If stability is required, it is possible to use a second, temporary, array so elements can be classified sequentially. However, in this case, the algorithm will require $O(n)$ space.









The time complexity

- It takes $\Theta(1)$ time to fix up the relationships among the elements A[*i*], A[LEFT(*i*)], and A[RIGHT(*i*)].
- Also, we need to run MAX-HEAPIFY on a subtree rooted at one of the children of node *i*.
- The children's subtrees each have size at most $2n/3$
  - worst case occurs when the last row of the tree is exactly half full
- The running time of MAX-HEAPIFY is

$$T(n) = T(2n/3) + \Theta(1)$$
$$= O(\lg n)$$

  - solve it by case 2 of the master theorem
- Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$.

Time complexity$_{1/2}$

- **Analysis 1**:
  - Each call to MAX-HEAPIFY costs $O(\lg n)$, and there are $O(n)$ such calls.
  - Thus, the running time is $O(n \lg n)$. This upper bound, through correct, is **not asymptotically tight**.
- **Analysis 2**:
  - For an n-element heap, height is $\lfloor \lg n \rfloor$ and at most $\lceil n / 2^{h+1} \rceil$ nodes of any height h.
  - The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$.
  - The total cost is $\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$.