Linked Lists

Adding to front:

```
void addToFront(string v) {
    Node *p;
    p = new Node; // #1
    p->value = v; // #2
    p->next = head; // #3
    head = p; // #4 }
```

Adding to the back:

```
void addToRear(string v) {
    if (head == nullptr)
        addToFront(v); // easy!!!
    else {
        Node *p;
        p = head; // #1
        while(p->next != nullptr) // #1
              p = p->next; // #1
        Node *n = new Node; // #2
        n->value = v; // #3
        p->next = n; // #4
        n->next = nullptr; // #5    } }
```

Adding to middle:

```
Node* temp = head;
for(int i=2; i < position; i++) {
  if(temp->next != NULL) {
    temp = temp->next;
  } }
newNode->next = temp->next;
temp->next = newNode;
```

Deleting:

```
void deleteItem(string v) {
    Node *p = head;
    while (p != nullptr) // #1 {
        p = p->next;
    }
    if (p->next && p->next->value == v) // #2
        break; // p pts to node above
    if (p != nullptr) // #3 {
        Node *killMe = p->next; // #4
    p->next = killMe->next; // #5
    delete killMe; // #6
    } }
```

Remove duplicates from linked list(assumes sorted)

```
ListNode* deleteDuplicates(ListNode* head) {
    if(!head) return head;
    ListNode *t = head, *p = head->next;
    int pre = head->val;
    while(p) {
        if(pre != p->val) {
            t->next = p;
            pre = p->val;
            t = t->next;
        }
        p = p->next;
    }
    t->next = NULL;
    return head; }
```

If linked list is a palindrome:

```
bool isPalindrome(ListNode* head) {
    vector<int> v;
    while(head) {
        v.push_back(head->val);
        head = head->next;
    }
    for(int i = 0; i < v.size()/2; ++i) {
        if(v[i] != v[v.size()-i-1]) return
false; }
    return true; }
```

Rotate list by k amount:

```
ListNode* rotateRight(ListNode* head, int k) {
    if(!head) return head;
    int len = 1;
    ListNode *p = head;
    while(p->next) { len++; p = p->next; }
    p->next = head;
    if(k %= len)
        for(int i = 0; i < len-k; ++i,
p=p->next) ;
    ListNode* newHead = p->next;
    p->next = NULL;
    return newHead; }
```

Add two numbers that are stored in linked lists:
Recursive version:

```
ListNode* addTwoNumbers(ListNode* l1, ListNode*
l2) {
    if(!l1 && !l2) return NULL;
    int c = (l1? l1->val:0) + (l2? l2->val:0);
    ListNode *newHead = new ListNode(c%10),
*next = l1? l1->next:NULL;
    c /= 10;
```

```
    if(next) next->val += c;
    else if(c) next = new ListNode(c);
    newHead->next = addTwoNumbers(l2?
l2->next:NULL, next);
    return newHead; }
```

Reversing linked list from place m to n:

```
ListNode* reverseBetween(ListNode* head, int m,
int n) {
    ListNode newHead(0);
    newHead.next = head;
    ListNode *pre = &newHead, *cur = head,
*next = NULL;
    int i = 1;
    while(i < n) {
        if(i++ < m) { pre = cur; cur =
cur->next; }
        else {
            next = cur->next;
            cur->next = cur->next->next;
            next->next = pre->next;
            pre->next = next;   } }
    return newHead.next; }
```

Find a cycle in the linked list:

```
ListNode *detectCycle(ListNode *head) {
    ListNode *slow = head, *fast = head;
    while(fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if(slow == fast) break;
    }
    if(slow != fast) return NULL;
    fast = head;
    while(fast && fast->next) {
        if(slow == fast) return slow;
        slow = slow->next;
        fast = fast->next;
    }
    return NULL;}
```

Merging linked lists:

```
ListNode* mergeSortedLists(ListNode* l1,
ListNode* l2) {
  // Base cases
  if(list1 == nullptr)
    return list2;
  if(list2 == nullptr)
    return list1;

  ListNode* head;
  if(list1->val < list2->val) {
    head = list1;
    head->next = mergeSortedLists(list1->next,
list2);
  }
  else{
    head = list2;
    head->next = mergeSortedLists(list1,
list2->next);
  }
  return head;
}
```

Difference in linked lists

```
bool checkNodeDifference(Node* head, int diff)
{
    if (head == nullptr)
        return true;
    Node* curr = head;
    Node* next = head->next;
    while (curr->next != nullptr) {
        if (next->val - curr->val != diff)
            return false;
        curr = next;
        next = next->next; }
    return true;}
```

Stacks

In order traversal using stack:

```
#include<iostream>
#include<stack>

using namespace std;

struct Node {
  Node* left;
  Node* right;
  int data;
};

void inorder(Node* p) {
  stack<Node*> MyStack;

  do {
    while (p != nullptr) {
      MyStack.push(p);
      p = p->left;
    }
```

```
    if (!MyStack.empty()) {
      p = MyStack.top();
      MyStack.pop();

      cout << p->data << endl;

      p = p->right;   }
 } while(!MyStack.empty() || p != nullptr);}
```

Balancing brackets

```
#include <stack>

bool isBalanced(string &s) {
  stack<char> stk;

  // consider the ith char of the string
  for (int i = 0; i < s.size(); i++) {
    // If the ith char of a string is an open
bracket, add it to stack.
    if (s[i] == '(' || s[i] == '{' || s[i] ==
'[') {
      stk.push(s[i]);
    }
    // else it's a close bracket
    // edge case: if it's a close bracket and
stack is empty, it's unbalanced
    else if (stk.empty())
      return false;
    else {
      // check whether the top is a matching
open bracket
      // if true, keep going; else, not
balanced, return false
      bool balanced = true;
      switch (s[i]) {
      case ')':
        if (stk.top() != '(') balanced = false;
break;

      case '}':
        if (stk.top() != '{') balanced = false;
break;

      case ']':
        if (stk.top() != '[') balanced = false;
break;
      }

      if (!balanced)
        return false;
```

```
      else
        stk.pop();   }}

  return stk.empty();}
```

Undirected graph traversal

```
int num_connected_components_stack(const
vector< vector<int> > &adj_graph) {
int n = adj_graph.size();
vector<bool> seen(n, false);

int count = 0;

  stack<int> s;

  for (int a = 0; a < n; a++) {
    if (!seen[a]) {
      s.push(a);

      while (!s.empty()) {
        int cur = s.top();
        s.pop();
        for (int i = 0; i <
adj_graph[cur].size(); i++) {
          int next = adj_graph[cur][i];
          if (!seen[next]) {
            s.push(next);
            seen[next] = true;
          }       } }
      count++;    }  }
    return count;}
```

Queue from stacks

```
template <class T>
class MyQueue {
private:
  stack<T> in;
  stack<T> out;
public:
  void pop() {
    if (out.empty()) {
      while (!in.empty()) {
        T& t = in.top();
        in.pop();
        out.push(t);
      }
    }
    out.pop();
  }
  void push(const T& val) {
```

```
    in.push(val);
  }
  T& front() const {
    if (out.empty()) {
      while (!in.empty()) {
        T& t = in.top();
        in.pop();
        out.push(t);
      }
    }
    return out.top(); }};
```

Shortest possible file path

```
vector<string> shortestFilePath(vector<string>
longFilePath){
  vector<string> answer;
  stack<string> filePath;
  for (vector<string>::iterator it =
longFilePath.begin();
                              it !=
longFilePath.end(); it++) {
    if (*it == ".." && !filePath.empty()) {
      filePath.pop();
    } else if (*it != ".") {
      filePath.push(*it); } }
  while (!filePath.empty()){
    answer.emplace(answer.begin(),
filePath.top());
    filePath.pop();}
  return answer;}
```

Postfix operations:

```
#include <stack>

int evalPostfix(char* c) {
  std::stack<int> operands;
  char* cur = c;
  while (*cur != '\0') {
    if (isdigit(*cur)) {
      operands.push(*cur-48);
    } else {
      int a = operands.top();
      operands.pop();
      int b = operands.top();
      operands.pop();
      switch (*cur) {
        case '+':
          operands.push(b+a);
          break;
        case '-':
          operands.push(b-a);
          break;
        case '*':
          operands.push(b*a);
          break;
        case '/':
          operands.push(b/a);
          break; } }
    cur++;}
  return operands.top();}
```

Reverse items in queue

```
queue<int> reverseQueue(queue<int> givenQueue)
{

int numberElements = givenQueue.size();

stack<int> reverser;

for(int i = 0; i < numberElements; i++) {
   int currentElement = givenQueue.front();
   reverser.push(currentElement);
   givenQueue.pop();
}

queue<int> reversedQueue;

for(int i = 0; i < numberElements; i++) {
    int currentElement = reverser.top();
    reversedQueue.push(currentElement);
    reverser.pop(); }
return reversedQueue;}
```

Reverse a stack:

```
void reverse(stack *s, queue *q) {
    if (s.empty()) return;
    while (!s.empty()) {
        q.push(s.top());
        s.pop();  }
    while (!q.empty()) {
        s.push(q.front());
        q.pop();}}
```

Queues

Solution: Stack from a Queue

```
bool Stack::empty() const {
   return q.empty();
}

int Stack::size() const {
   return q.size();
}

int Stack::top() {
   return q.back();
}
```

```
void Stack::push(int value) {
   q.push(value);
}

void Stack::pop() {
   int n = q.size() - 1;
   for (int i = 0; i < n; i++) {
      q.push(q.front());
      q.pop();
   }
   q.pop();
}
```

Also try implementing a queue by using one or more stacks!

Swapping the nodes of binary tree

```
#include <queue>

using namespace std;

struct Node {
    int m_data;
    Node* m_left;
    Node* m_right;
};

void reverseTreeRecur(Node* p) {
    // Base case
    if (p == nullptr) return;
    // Store the left pointer in a temp
variable
    Node* temp = p->m_left;

    // Swap the pointers at this level
    p->m_left = p->m_right;
    p->m_right = temp;

    // Recursively call this function for the
next level in the tree
    reverseTreeRecur(p->m_left);
    reverseTreeRecur(p->m_right);
}

void reverseTreeIter(Node* root) {
    // create a queue to keep track of nodes to
reverse
    queue<Node*> q;

    // attempt to push the root onto the queue
    if (root != nullptr) q.push(root);
    // keep swapping nodes while the queue is
not empty
    while (!q.empty()) {
        // Dequeue the first node
        Node* n = q.front();
        q.pop();

        // add the nodes children to the queue
        if (n->m_left != nullptr) {
            q.push(n->m_left);
        }

        if (n->m_right != nullptr) {
            q.push(n->m_right);
        }

        // Store the left pointer in a temp
variable
        Node* temp = n->m_left;

        // Swap the pointers at this level
        n->m_left = n->m_right;
        n->m_right = temp;
    }
}
```

Delete all nodes of binary tree with queue

```
#include <queue>

struct Node {
    Node(const int key, const int value) {
        k = key;
        v = value;
        left = right = nullptr;
    }

    int k, v;
    Node* left, right;};

void clear(Node* root) {
    Node* curr = root;

    // The tree is already empty
    if (curr == nullptr) return;
    // Queue for a level-order traversal
    std::queue<Node*> q;
    q.push(curr);

    while (!q.empty()) {
        Node* node = q.front();
        q.pop();

        if (node->left) q.push(node->left);
        if (node->right) q.push(node->right);
        delete node;   }
root = nullptr;}
```

Recursion

Reverse string recursively

```
string reverseRecursively(string s) {
  if (s.length() <= 1)
    return s;

  return s[s.length()-1] +
reverseRecursively(s.substr(0, s.length()-1));}
```

Undirected graph traversal

```
#include <stack>
#include <vector>
using namespace std;
void dfs(int u, const vector< vector<int> >
&adj_graph, vector<bool> &seen) {
  seen[u] = true;
  for (int i = 0; i < adj_graph[u].size(); i++)
{
    int next = adj_graph[u][i];
    if (!seen[next]) {
      dfs(next, adj_graph, seen);  } }}

int num_connected_components_recur(const
vector< vector<int> > &adj_graph) {
  int n = adj_graph.size();
  vector<bool> seen(n, false);
  int count = 0;
  for (int i = 0; i < n; i++) {
    if (!seen[i]) {
      dfs(i, adj_graph, seen);
      count++; }}
  return count;}
```

Number is even

```
bool is_even(int number) {
  if (number == 0) {
    return true;
  } else if (number == 1) {
    return false;
  } else if (number > 0) {
    return is_even(number - 2);
  } else {
    return is_even(number + 2);
  }
}
```

Any true recursion

```
bool any_true(const int a[], int n) {
  if (n <= 0)
    return false;

  if (givenCondition(a[0]))
```

```
    return true;
  else
    return any_true(a + 1, n - 1);
}
```

Create tree from heap

```
Node* helper_creator(int a[], int n, int level)
{
  if (n <= level)
       return nullptr;
  Node* head = new Node;
  head->val = a[level];
  if (n > 2 * level + 1)
       head->left = helper_creator(a, n, 2 *
level + 1);
  if (n > 2 * level + 2)
       head->right = helper_creator(a, n, 2 *
level + 2);
  return head;
}

Node* create_tree(int a[], int n) {
  return helper_creator(a, n, 0);
}
```

Multiplication

```
int multiply(int x, int y) {
    // 0  multiplied with anything should
return 0
    if (x == 0 || y == 0)
        return 0;

    // Add x with itself with a y number of
times
    // Recursively calling multiply(x, y-1)
    if (y > 0)
        return (x + multiply(x, y - 1));

    // If y is negative, we want to return the
negative of the result
    if (y < 0)
        return -multiply(x, -y);
}
```

Sum of array:

```
int arraySum(int a[], int n) {
    // Base case: no more elements to add
    if (n == 0)
        return 0;
```

```
    // Recursive case: add to current with array of one less element
    int sum = a[0] + arraySum(a+1, n-1);

    return sum;
}
```

Find cube root

```
double cbrt_helper(double d, double low, double high, double epsilon) {
  double approx = (low + high) / 2;
  double error = d - approx*approx*approx;
  if (error < 0) {
    error = -error;
    if (error < epsilon)
      return approx;
    else
      return cbrt_helper(d, low, approx, epsilon);
  } else {
    if (error < epsilon)
      return approx;
    else
      return cart_helper(d, approx, high, epsilon);
  }
}

double cbrt(double d, double epsilon) {
  return cbrt_helper(d, 0, d, epsilon);
}
```

How many ways to add with 1 and 2

```
int numCounts(int n) {
  if (n <= 0)
    return 0;
  if (n <= 2)
    return n;
  int p2 = 1;
  int p1 = 2;
  int total = 0;
  for (int i = 3; i <= n; i++) {
    total = p1 + p2;
    p2 = p1;
    p1 = total;
  }
  return total;
}
```

If element is in array

```
bool isInArray(int element, int array[], int n)
{
    if (n < 1) return false;
    if (array[n - 1] == element) return true;
    return isInArray(element, array, n - 1);
}
```

Calculate exponential

```
int FastExp(int p, int n) {
  if (n == 0) return 1;
  if (n == 1) return p;
  int result = FastExp(p, n/2) % N;
  if (n % 2 == 1) {
    return result * result * p % N;
  } else {
    return result * result % N;
  }
}
```

Number appears at least n times

```
bool atLeastN(int* arr, int len, int n, int target) {
  if (n == 0)
    return true;
  if (len <= 0)
    return false;
  if (arr[len-1] == target)
    return atLeastN(arr, len-1, n-1, target);
  else
    return atLeastN(arr, len-1, n, target);
}
```

Number of containers (division)

```
int containers_needed(int num_items, int box_size) {
    if (num_items <= box_size)
        return 1;

    int first_half = containers_needed(num_items/2, box_size);
    int second_half = containers_needed(num_items - (num_items/2), box_size);

    return first_half + second_half;
}
```

Count how many are positive

```
int countPos(int arr[], int size) {
```

```
    if (size == 0)
        return 0;
    if (arr[0] > 0)
        return 1 + countPos(arr + 1, size - 1);
    return countPos(arr + 1, size - 1);
}
```

Fibonacci sequence

```
int fibonacci(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n - 1) + fibonacci(n -
2);
    }
}
```

STL Maps

STL: Maps

```
map<string, int> m;
m["das"] = 33;     // "das" -> 33
m.erase("das");    // Removes element by key.
int size = s.size();
bool empty = s.empty();

// Returns m.end() if value is not found.
map<string, int>::iterator it = m.find("das");
if (it != m.end()) {
  m.erase(it);
}

// Takes two iterators, deleting the values
// between [begin, end).
m.erase(m.begin(), m.begin + 3);
```

Find the one duplicate in array

```
string getDup(string s[], int n) {
    unordered_map<string, bool> map;
    for (int i = 0; i < n; i ++) {
        if (map[s[i]] == true) return s[i];
        map[s[i]] = true;
    }
    return "-1";
}
```

Counting even more duplicates

```
int countDups(int ** arr, int height, int
width) {
    if (width < 1 || height < 1)
        return 0;

    // An unordered map allows constant lookup
(if small amount of collisions)
    unordered_map<int, int> vals;
    unordered_map<int, int>::const_iterator
checker;

    int curr = 0;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
        curr = arr[i][j];

        // Check if our unordered_map already
has the value
        checker = vals.find(curr);
        // If not, add it as a new value
        if (checker == vals.end())
            vals.insert({curr, 0});
        // Otherwise increment occurences of
that value
        else
            vals[curr]++;
        }
  }

  // Count all unique elements with more than
one occurence
  int count = 0;
  for (checker = vals.begin(); checker !=
vals.end(); checker++)
        if (checker->second > 0)
        count++;
  return count;
}
```

All unique pairs of integers that add up to given sum

```
void PrintNums(int sum, int arr[], int arrsize)
{
    if (arrsize < 0)
        cout << "invalid size of array" <<
endl;

    set<int> database;  // keeps track of what
elements we have seen so far
    map<int, int> frequency;  // keeps track of
how many times each element is seen
```

```
                               // each element
is the key and its frequency is the value

    for (int i = 0; i < arrsize; i++) {
        database.insert(arr[i]);

        map<int, int>::iterator it_freq =
frequency.find(arr[i]);

        if (it_freq != frequency.end()) {
            frequency[arr[i]]++;  // if already
exists, increment count
        } else {
            frequency.insert(pair<int,
int>(arr[i], 1));  // doesn't exist, add the

// element with count 1
        }
    }

    set<int>::iterator it_dat =
database.begin();

    while (it_dat != database.end()) {
        set<int>::iterator it_dat2 =
database.find(sum - *it_dat);

        if (it_dat2 != database.end() &&
((it_dat != it_dat2) || (frequency[*it_dat] >
1))) {
            cout << *it_dat << "," << *it_dat2
<< endl;
        }

        it_dat++;
    }

    return;
}
```

STL Data Structures

STL: Lists

```
list<int> l;
l.push_front(32); // Adds element to front.
l.pop_front();    // Deletes element at front.
l.push_back(31);  // Adds element to end.
l.pop_back();     // Deletes element at end.
int size = l.size();
bool empty = l.empty();

// Insert and erase, as before.
l.insert(l.begin(), 5, 33);
l.erase(l.begin(), l.begin() + 3);
```

STL: Vectors

```
vector<int> v;
v.push_back(31);  // Adds element to end.
int front = v[0]; // Square bracket to access.
v.pop_back();     // Deletes element at end.
int size = v.size();
bool empty = v.empty();

// Takes an iterator, number to copy, and
// value to copy. v = [33, 33, 33, 33, 33].
v.insert(v.begin(), 5, 33);

// Takes two iterators, deleting the values
// between [begin, end).
v.erase(v.begin(), v.begin() + 3);
```

STL: Sets

```
set<int> s;
s.insert(31);     // Adds element to set (unique).
s.erase(31);      // Remove element from set.
int size = s.size();
bool empty = s.empty();

// Returns s.end() if value is not found.
set<int>::iterator it = s.find(30);
if (it != s.end()) {
  s.erase(it);
}

// Takes two iterators, deleting the values
// between [begin, end).
s.erase(s.begin(), s.begin + 3);
```

Sorts

Mergesort: O (N * logN). Works the same on any kind of array, but could be slow because of space

Why is this so complicated

```
void merge(int *array, int l, int m, int r) {
   int i, j, k, nl, nr;
   //size of left and right sub-arrays
   nl = m-l+1; nr = r-m;
   int larr[nl], rarr[nr];
   //fill left and right sub-arrays
   for(i = 0; i<nl; i++)
      larr[i] = array[l+i];
   for(j = 0; j<nr; j++)
      rarr[j] = array[m+1+j];
   i = 0; j = 0; k = l;
   //marge temp arrays to real array
   while(i < nl && j<nr) {
      if(larr[i] <= rarr[j]) {
         array[k] = larr[i];
         i++;
      }else{
         array[k] = rarr[j];
         j++;
      }
      k++;
   }
   while(i<nl) {       //extra element in left
array
      array[k] = larr[i];
      i++; k++;
   }
   while(j<nr) {     //extra element in right
array
      array[k] = rarr[j];
      j++; k++;
   }
}
void mergeSort(int *array, int l, int r) {
   int m;
   if(l < r) {
      int m = l+(r-l)/2;
      // Sort first and second arrays
```

```
      mergeSort(array, l, m);
      mergeSort(array, m+1, r);
      merge(array, l, m, r);
   }
}
```

Quicksort: O(N * logN) wait: worst case O(N^2)
This is bad if the array is already sorted or mostly sorted
Why is this also so complicated

```
int partition(int array[], int low, int high) {

  // select the rightmost element as pivot
  int pivot = array[high];

  // pointer for greater element
  int i = (low - 1);

  // traverse each element of the array
  // compare them with the pivot
  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {

      // if element smaller than pivot is found
      // swap it with the greater element
pointed by i
      i++;

      // swap element at i with element at j
      swap(&array[i], &array[j]);
    }
  }

  // swap pivot with the greater element at i
  swap(&array[i + 1], &array[high]);

  // return the partition point
  return (i + 1);
}

void quickSort(int array[], int low, int high)
{
  if (low < high) {

    // find the pivot element such that
    // elements smaller than pivot are on left
of pivot
    // elements greater than pivot are on righ
of pivot
    int pi = partition(array, low, high);

    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);

    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
}
```

Selection sort: O(N^2)
Stable sort, same efficiency on unsorted vs mostly sorted arrays

```
void selectionSort(int A[], int n) {
    for (int i = 0; i < n; i++) {
        int minIndex = i;
        for (int j = i+1; j < n; j++) {
            if (A[j] < A[minIndex])
                minIndex = j;
        }
       swap(A[i], A[minIndex]);
       }
}
```

Insertion sort: more efficient on more sorted arrays O(N^2)

```
void insertionSort(int A[], int n) {
    for(int s = 2; s <= n; s++) {
        int sortMe = A[ s - 1 ];
        int i = s - 2;
        while (i >= 0 && sortMe < A[i]) {
            A[i+1] = A[i];
             --i;
        }
        A[i+1] = sortMe;
    }
}
```

Bubble sort: O(N^2)

```
void bubbleSort(int Arr[], int n) {
    bool atLeastOneSwap;
    do {
        atLeastOneSwap = false;
        for (int j = 0; j < (n-1); j++) {
            if (Arr[j] > Arr[j + 1]) {
                Swap(Arr[j],Arr[j+1]);
                atLeastOneSwap = true;
            }
        }
    }
    while (atLeastOneSwap == true);
}
```

In addition:
Bubble sort: largest always sinks to the bottom after 1 round
Insertion sort: swaps the first two after one round
Selection sort: after 1 round, smallest will be brought to front

Heapsort: O(N * logN)

```
// function to heapify the tree
void heapify(int arr[], int n, int root)
{
   int largest = root; // root is the largest element
   int l = 2*root + 1; // left = 2*root + 1
   int r = 2*root + 2; // right = 2*root + 2

   // If left child is larger than root
   if (l < n && arr[l] > arr[largest])
   largest = l;

   // If right child is larger than largest so far
   if (r < n && arr[r] > arr[largest])
   largest = r;

   // If largest is not root
   if (largest != root)
      {
      //swap root and largest
      swap(arr[root], arr[largest]);

      // Recursively heapify the sub-tree
      heapify(arr, n, largest);
      }
}

// implementing heap sort
void heapSort(int arr[], int n)
{
   // build heap
   for (int i = n / 2 - 1; i >= 0; i--)
   heapify(arr, n, i);

   // extracting elements from heap one by one
   for (int i=n-1; i>=0; i--)
   {
      // Move current root to end
      swap(arr[0], arr[i]);

      // again call max heapify on the reduced heap
      heapify(arr, i, 0);
   }
}
```

Trees: Preorder traversal:

```
void PreOrder(Node *cur) {
    if (cur == nullptr)
        return;
    cout << cur->value; // Process the current node.
    PreOrder(cur->left);
    PreOrder(cur-> right);
}
```

In order traversal:

```
void inOrder(Node* root) {
    if ( root ) {
        inOrder(root->left);
        cout << root->key << " ";
        inOrder(root->right);   }}
```

Postorder traversal:

```
void postOrder(Node* node) {
     if (node == NULL)  return;
     cout << node->data;
     postOrder(node->left);
     postOrder(node->right);}
```

Swap rightmost

```
#include <iostream>
using namespace std;

void swap_rightmost (Node* head)  {
  if (head == NULL || head -> right == NULL)
    return;
  Node* current = head;
  Node* previous = NULL;
  while (current -> right != NULL) {
    previous = current;
    current = current -> right;
  }
  int prevValue = current -> value;
  Node* prevLeft = current -> left;

  current -> right = head -> right;
  current -> left = head -> left;
```

```
  current -> value = head -> value;

  previous -> right = head;
  head -> value  = prevValue;
  head -> left = prevLeft;
  head = current;
}
```

Reverse binary tree:

```
void reverse_binary_tree(Node* root) {
  if (root == nullptr)
    return;

  Node* temp = root->left;
  root->left = root->right;
  root->right = temp;

  reverse_binary_tree(root->left);
  reverse_binary_tree(root->right);}
```

Max in a k-ary tree

```
int max_value(Node* root, int K) {
  if (root == NULL)
    return -1;
  int max = -1;
  if (root->value > max)
    max = root->value;
  for (int i = 0; i < K; i++) {
    int temp_max =
max_value(root->child_ptrs[i], K);
    if (temp_max > max)
      max = temp_max;
  }
  return max;}
```

Counting all nodes:

```
int nodeCount(Node *node) {
  int count = 1;
  for (int i = 0; i < node->children.size();
i++)
    count += nodeCount(node->children[i]);
  return count;}
```

Breadth first traversal:

```
void printTree(Node* root) {
    if (root != nullptr) {
        // first node
        std::cout << root->value << endl;
        // go left and right
        printTree(root->left);
        printTree(root->right);
    }
}
```

Height of binary tree:

```
int getHeight(Node* n) {
    if (n == nullptr) {
        return 0;
    } else {
        int left_height = getHeight(n->left);
        int right_height = getHeight(n->right);
        // return the max height between the
left and right subtrees
        if (left_height < right_height)
            return right_height + 1;
        else
            return left_height + 1;   }}
```

Invert tree:

```
void invertTree(Node* root) {
    if (root == NULL)
        return;
    Node* temp = root->right;
    root->right = root->left;
    root->left = temp;
    invertTree(root->right);
    invertTree(root->left);}
```

Balanced tree

```
bool isBalanced(TreeNode* node) {
    // An empty tree is balanced
    if (node == nullptr)
        return true;

    // Difference in height of left and right
subtrees must be within 1
    // Both subtrees must be balanced
    return abs(height(node->left) -
height(node->right)) <= 1
           && isBalanced(node->left)
           && isBalanced(node->right);
}

int height(TreeNode* root) {
    // Height of an empty tree is 0
    if (root == nullptr)
        return 0;
    // Height of a tree is 1 more than the
height of the longer
    // of the left and right subtrees
```

```
    return 1 + max(height(node->left),
height(node->right));
}
```

Node exists:

```
bool exists(Node *root, Node *dest) {
    if (!root || !dest) return false;
    if (root == dest)  return true;
    return exists(root->left, dest) ||
exists(root->right, dest);}
```

Depth of shallowest node w/ value:

```
int depth_of_x(node * root, int x) {
  if (root == nullptr)
    return 0;
  if (root->value == x)
    return 1;

  int left_depth = depth_of_x(root->left, x);
  int right_depth = depth_of_x(root->right, x);

  // If x is found in both right and left,
increment depth of shallower one
  if (left_depth != 0 && right_depth != 0)
    return std::min(left_depth, right_depth) +
1;

  // If x is found in left but not right,
increment depth of left
  if (left_depth != 0)
    return left_depth + 1;

  // If x is found in right but not left,
increment depth of right
  if (right_depth != 0)
    return right_depth + 1;

  // x was not found in either
  return 0;
}
```

Symmetric tree:

```
bool ismirror(TreeNode *p, TreeNode *q) {
    if (!p&&!q) return true;
    else if (!p||!q) return false;

    if (p->val == q->val) return
ismirror(p->left, q->right) &&
ismirror(p->right, q->left);
    else
        return false;
}

bool isSymmetric(TreeNode* root) {
    if (!root) return true;

    return ismirror(root->left, root->right);
}
```

Is BST?

```
bool isBSTHelper(Node* root, int min, int max)
{
    if ( root == nullptr) return true;
    if ((root->value <= min) || (root->value >
max)) {
        return false;
    }

    if (!isBSTHelper(root->left, min,
root->value) ||
        !isBSTHelper(root->right, root->value,
max)) {
        return false;
    }

    return true;
}

bool isBST(Node* root)  {
    return isBSTHelper(root, INT_MIN, INT_MAX);
}
```

Minimum depth:

```
int minDepth(Node* root) {
    if (root == nullptr) return 0;
    if (root->left == nullptr && root->right ==
nullptr) return 1;
    if (!root->left) return
minDepth(root->right) + 1;
    if (!root->right) return
minDepth(root->left) + 1;
    return min(minDepth(root->left),
minDepth(root->right)) + 1;}
```

Max sum of subtree

```
int max_subtree(const TreeNode* root) {
    // recursion ends
    if (root == nullptr) return 0;
    int sum = root -> item;
```

```
    if (root -> left != nullptr) {
        sum += root -> left -> item;
    }
    if (root -> right != nullptr) {
        sum += root -> right -> item;
    }
    int a = max_subtree(root->left);
    int b = max_subtree(root->right);

    int c = a > b ? a : b;
    return sum > c ? sum : c;
}
```

Same tree

```
bool isSameTree(Node* p, Node* q) {
  if (!p) return !q;
  if (!q) return !p;
  return (p->val == q->val) &&
isSameTree(p->left, q->left) &&
isSameTree(p->right, q->right);
}
```

Binary Search Trees

```
// Binary Search Tree operations in C++
#include <iostream>
using namespace std;

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

// Create a 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;
}

// Insert a node
struct node *insert(struct node *node, int key)
{
  // Return a new node if the tree is empty
  if (node == NULL) return newNode(key);

  // Traverse to the right place and insert the
node
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  return node;
}

// Deleting a node
struct node *deleteNode(struct node *root, int
key) {
  // Return if the tree is empty
  if (root == NULL) return root;

  // Find the node to be deleted
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    // If the node is 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;
    }

    // If the node has two children
    struct node *temp =
minValueNode(root->right);

    // Place the inorder successor in position
of the node to be deleted
    root->key = temp->key;

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

Search for a value:

```
bool Search(int V, Node *ptr) {
    if (ptr == nullptr)
        return(false); // nope
```

```
    else if (V == ptr->value)
        return(true); // found!!!
    else if (V < ptr->value)
        return(Search(V,ptr->left));
    else
        return(Search(V,ptr->right));}
```

Min of binary search tree: (max use right)

```
int GetMin(node *pRoot) {
    if (pRoot == NULL) return(-1); // empty
    while (pRoot->left != NULL)
         pRoot = pRoot->left;
    return(pRoot->value);}
```

Hash tables: First missing number

```
#include <vector>
#include <unordered_set>
#include <limits.h>
using namespace std;

unsigned firstMissing(vector<unsigned> boxNums)
{
  unordered_set<unsigned> foundNums;
  for (unsigned i = 0; i < boxNums.size(); i++)
{
    foundNums.insert(boxNums[i]);
  }
  for (unsigned i = 1; i < UINT_MAX; i++) {
    if (!foundNums.count(i)) return i;
  }
  return UINT_MAX;}
```

First unique character

```
int firstUniqueChar(std::string s) {
// Map character to the frequency of occurrence
unordered_map counter;
    for(int i = 0; i < s.size(); i++) {
        counter[s[i]]++;
    }
    for (int i = 0; i < s.size(); i++) {
        if (counter[s[i]] == 1) return i;
    }
    return -1;}
```

Two sum

```
bool twoSum(const int arr[], int n, int target)
{
unordered_set numsFound;
    for (int i = 0; i < n; i++) {
        int complement = target - arr[i];
        if (numsFound.find(complement) !=
numsFound.end()) {
                return true;    }
        else numsFound.insert(arr[i]);      }
     return false; }
```

Group anagrams

```
vector> groupAnagrams(vector strs){
    unordered_map> anagrams;
    for(int i = 0; i < strs.size(); i++) {
        int key = calculateHash(strs[i]);
        anagrams[key].push_back(strs[i]); }
    unordered_map>::iterator it =
anagrams.begin();
    vector> res; // Loop through Hash Table
    while(it != anagrams.end()) { // it->second
is the vector of strings (i.e. anagrams) //
corresponding to the one same key
        res.push_back(it->second);
        it++;
    }
    return res; }
```

Three sum

```
bool sum3(const int arr[], int n) { //create
hash table
    unordered_set hashedArr;
    for(int i = 0; i < n; i++){
        hashedArr.insert(arr[i]);
    } //search for opposite of every pair
for(int i = 0; i < n; i++) {
    for(int j = i+1; j < n; j++) {
        int oppSum = (arr[i] + arr[j])*-1;
        if(oppSum != arr[i] && oppSum != arr[j]
&& hashedArr.find(oppSum) != hashedArr.end()) {
           return true;
        }
     }
   } return false;
}
```