```cpp
List(const List<T> &other) {
    m_head = nullptr;
    Node<T>* curr = other.m_head;
    Node<T>* prev = nullptr; // node to save previous node
    while (curr) {
        Node<T>* newNode = new Node<T>(curr->element);
        if (!m_head) {
            m_head = newNode; // set up first node
        }
        else {
            prev->next = newNode; // link previous node to new node
        }
        prev = newNode; // saves memory address of previous node to link
        curr = curr->next; // shifts to next node in the linked list be
    }
    m_size = other.m_size; // copy "other" size
}
```

```cpp
void reverse() {
    if (!m_head) {
        throw std::out_of_range("Cannot reverse an empty container");
    }
    Node<T>* node = nullptr;
    while (m_head) {
        Node<T>* temp = m_head->next;
        m_head->next = node; // connect selected node to left
        node = m_head; // select next node
        m_head = temp; // select next node
    }
    m_head = node; // set head to node
}
```

```cpp
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode* slow = head;
        ListNode* fast = head;
        if(head == NULL || head->next == NULL){
            return false;
        }
        // checks if current node is NULL
        // checks if next node is NULL
        // ensures no null pointer dereferencing
        while(fast != NULL && fast->next != NULL){
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast){
                return true;
            }
        }
        return false;
    }
};
```

```cpp
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        ListNode* sentinel = new ListNode(NULL); // dummy node to create
        ListNode* temp1 = list1; // temp to iterate list1
        ListNode* temp2 = list2; // temp to iterate list2
        ListNode* tail = sentinel; // tail of dummy node

        while(temp1 != nullptr && temp2 != nullptr){ // check that both
            if(temp1->val > temp2->val){
                tail->next = temp2;
                temp2 = temp2->next;
            }
            else{
                tail->next = temp1;
                temp1 = temp1->next;
            }
            tail = tail->next;
        } // exit loop if one of them reaches the end

        if(temp1 != nullptr){ // append rest of list1
            tail->next = temp1;
        }
        if(temp2 != nullptr){ // append rest of list2
            tail->next = temp2;
        }
        return sentinel->next;
    }
};
```

```cpp
Node<T>* findSuccessor(Node<T>* node, T element) {
    if (!node) {
        return nullptr;
    }
    Node<T>* target = nullptr;
    Node<T>* temp = node;
    while (temp) {
        if (element < temp->element) {
            target = temp;
            temp = temp->left;
        }
        else if(element > temp->element){
            temp = temp->right;
        }
        else{
            if(temp->right){
                return findMinNode(temp->right);
            }
            break;
        }
    }
    return target;
}
```

```cpp
int select(int k){
  rank = m_left.weight + 1;
  if (k == rank){
    return v;
  }
  else if (k < rank) {
    return m_left.select(k);
  }
  else if (k > rank){
    return m_right.select(k - rank);
  }
}
```

```cpp
int rank(TreeNode* root, int key) {
    int rank = 0;
    while (root) {
        if (key < root->key) {
            // Move left, no change in rank
            root = root->left;
        } else if (key > root->key) {
            // Add left subtree size + 1 (including current node
            rank += (root->left ? root->left->weight + 1 : 1);
            root = root->right;
        } else {
            // Key found, final rank calculation
            rank += (root->left ? root->left->weight + 1 : 1);
            return rank;
        }
    }
    return -1; // Key not found
}
```

```cpp
int balanceFactor(TreeNode* node){
    if(!node){
        return 0;
    }
    return height(node->left) - height(node->right);
}
TreeNode* rightRotate(TreeNode* root){
    TreeNode* unbalancedNode = root->left;
    TreeNode* unwantedChild = unbalancedNode->right;
    unbalancedNode->right = root;
    root->left = unwantedChild;
    // set height of new root's new child first
    root->height = max(height(root->left), height(root->right)) + 1;
    // set height of new root
    unbalancedNode->height = max(height(unbalancedNode->left),
                                 height(unbalancedNode->right)) + 1;
    return unbalancedNode;
}
TreeNode* leftRotate(TreeNode* root){
    TreeNode* unbalancedNode = root->right;
    TreeNode* unwantedChild = unbalancedNode->left;
    unbalancedNode->left = root;
    root->right = unwantedChild;
    // set height of new root's new child first
    root->height = max(height(root->left), height(root->right)) + 1;
    // set height of new root
    unbalancedNode->height = max(height(unbalancedNode->left),
                                 height(unbalancedNode->right)) + 1;
    return unbalancedNode;
}
```

```cpp
TreeNode* deleteNode(TreeNode* root, int key) {
    if(!root){ // Case 1: Empty tree
        return root;
    }
    if(key < root->val){
        root->left = deleteNode(root->left, key);
    }
    else if(key > root->val){
        root->right = deleteNode(root->right, key);
    }
    else{
        if(!root->left && !root->right){ // Case 2: no children
            delete root;
            return nullptr;
        }
        else if(!root->right){ // Case 3: left child
            TreeNode* temp = root;
            root = root->left;
            delete temp;
            return root;
        }
        else if(!root->left){ // Case 3: right child
            TreeNode* temp = root;
            root = root->right;
            delete temp;
            return root;
        }
        else{ // Case 4: custody of both children
            TreeNode* temp = findMin(root->right);
            root->val = temp->val;
            root->right = deleteNode(root->right, temp->val);
        }
    }
    root->_height = max(height(root->_left),
                        height(root->_right)) + 1;
    int balance = calcBalanceFactor(root);
    // LL case : left child is left heavy
    if (balance > 1 && calcBalanceFactor(root->left) >= 0) {
      return _rightRotation(root);
    }
    // RR case : Right child is right heavy
    if (balance < -1 && calcBalanceFactor(root->right) <= 0) {
      return _leftRotation(root);
    }
    // LR case : Left child is right heavy
    if (balance > 1 && calcBalanceFactor(root->left) < 0) {
      current->_left = _leftRotation(current->_left);
      return _rightRotation(root);
    }
    // RL case : Right child is left heavy
    if (balance < -1 && calcBalanceFactor(root->right) > 0) {
      root->_right = _rightRotation(root->_right);
      return _leftRotation(root);
    }
    return root;
}
```

```cpp
void bubbleSort(vector<int>& arr){
    int n = arr.size();
    bool swapped;
    for(int i = 0; i < n - 1; i++){
        swapped = false;
        for(int j = 0; j < n - i - 1; j++){
            if(arr[j] > arr[j + 1]){
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        // no swap occurred
        if(!swapped)
            break;
    }
}
```

```cpp
TreeNode<T>* BinarySearchTree<T>::_insert(TreeNode<T>* current, T x)
  if (!current) {
    _size++;
    return new TreeNode<T>(x);
  }
  if (x < current->_item) {
    current->_left = _insert(current->_left, x);
  }
  else if (x > current->_item) {
    current->_right = _insert(current->_right, x);
  }
  else { // node already exists
    return current;
  }
  current->_weight++;
  current->_height = max(height(current->_left),
                         height(current->_right)) + 1;
  int balance = calcBalanceFactor(current);
  // LL case : left child is left heavy
  if (balance > 1 && x < current->_left->_item) {
    return _rightRotation(current);
  }
  // RR case : Right child is right heavy
  if (balance < -1 && x > current->_right->_item) {
    return _leftRotation(current);
  }
  // LR case : Left child is right heavy
  if (balance > 1 && x > current->_left->_item) {
    current->_left = _leftRotation(current->_left);
    return _rightRotation(current);
  }
  // RL case : Right child is left heavy
  if (balance < -1 && x < current->_right->_item) {
    current->_right = _rightRotation(current->_right);
    return _leftRotation(current);
  }
  return current;
}
```

```cpp
class dsu{
    vector<int> parent, size;

public:
    dsu(int n) : parent(n), size(n, 1) {
        for(int i = 0 ; i < n; i++){
            parent[i] = i;
        }
    }

    int findRoot(int p) {
        int root = p;

        // search for parent
        while (parent[root] != root) {
            root = parent[root];
        }

        // path compression
        while (parent[p] != p) {
            int temp = parent[p]; // save previous parent
            parent[p] = root; // change parent to root
            p = temp; // set previous parent to be target next loop
        }
        return root;
    }

    bool unite(int p, int q) {
        p = findRoot(p);
        q = findRoot(q);
        if(p == q){
            return false;
        }
        // weighted union: attach smaller tree to larger tree
        if (size[p] > size[q]) {
            parent[q] = p;  // Link q to p
            size[p] = size[p] + size[q];  // Update the size of the new
        }
        else {
            parent[p] = q;  // Link p to q
            size[q] = size[p] + size[q];  // Update the size of the new
        }
        return true;
    }
};
```

C++

```cpp
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        ListNode* slist = new ListNode(0, nullptr);
        ListNode* blist = new ListNode(0, nullptr);
        ListNode* small = slist;
        ListNode* big = blist;
        while (head != nullptr) {
            if (head->val < x) {
                small->next = head;
                small = small->next;
            } else {
                big->next = head;
                big = big->next;
            }
            head = head->next;
        }
        small->next = blist->next;
        big->next = nullptr;
        ListNode* result = slist->next;
        delete slist;
        delete blist;
        return result;
    }
};
```

```cpp
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head){
            return head;
        }
        int length=1;
        ListNode* tail=head;
        ListNode* cur=head;
        while (tail->next){
            tail=tail->next;
            length++;
        }
        k=k%length;
        if (k==0){
            return head;
        }
        for(int i=0;i<length-k-1;i++){
            cur=cur->next;
        }
        ListNode* newh=cur->next;
        cur->next=nullptr;
        tail->next=head;
        return newh;
    }
ListNode* insertionSortList(ListNode* head) {
        ListNode* dummy = new ListNode(0);
        ListNode* current = head;
        while (current != nullptr) {
            ListNode* prev = dummy;
            ListNode* nextNode = current->next;
            // Find the correct spot to insert current node
            while (prev->next != nullptr &&
                   prev->next->val < current->val) {
                prev = prev->next;
            }
            // Insert current node between prev and prev->next
            current->next = prev->next;
            prev->next = current;
            // Move to the next node in the original list
            current = nextNode;
        }
        return dummy->next;
    }
```

```cpp
void Heap<T>::heapify(int index) {
    int left = find_left(index);
    int right = find_right(index);
    int biggest = index;
    // check if left child is bigger than root
    if (left < _size && _heap[left] > _heap[index]) {
        biggest = left;
    }
    // check if right child is bigger than existing biggest
    // biggest as of now could be either root or left child
    if (right < _size && _heap[right] > _heap[biggest]) {
        biggest = right;
    }
    // biggest is either left or right
    if (biggest != index) {
        swap(&_heap[index], &_heap[biggest]);
        // recursively heapify the node that just got bubbl
        heapify(biggest);
    }
    // root is bigger than both left and right child, no he

}
template <class T>
void Heap<T>::insert(const T& item) {
  // TODO: implement this
    if (_size == DEFAULTHEAPSIZE) {
        throw out_of_range("Heap is full. Cannot add any more entr
    }
    _size++;
    int index = _size - 1;
    _heap[index] = item;
    while (index != 0 && _heap[index] > _heap[parent(index)]) {
        swap(&_heap[index], &_heap[parent(index)]); // bubble up
        index = parent(index);
    }
}
template <class T>
T Heap<T>::extractMax() {
  // TODO: implement this
    if (_size == 0) {
        throw out_of_range("Cannot extract from empty heap");
    }
    if (_size == 1) {
        _size--;
        return _heap[0];
    }
    T root = _heap[0];
    _heap[0] = _heap[_size - 1];
    _size--;
    heapify(0); // bubble up or down
    return root;
}
```

```cpp
template <class T>
void Heap<T>::increaseKey(int index, const T& to) {
    _heap[index] = to;
    while (index != 0 && _heap[index] > _heap[parent(index)]) {
        swap(&_heap[index], &_heap[parent(index)]);
        index = parent(index);
    }
}
template <class T>
void Heap<T>::decreaseKey(int index, const T& to) {
    _heap[index] = to;
    heapify(index);
}
template <class T>
void Heap<T>::changeKey(const T& from, const T& to) {
  // TODO: implement this
    if (from == to) {
        return;
    }
    int index = findKey(from);
    if (from > to) {
        decreaseKey(index, to);
    }
    else if (from < to) {
        increaseKey(index, to);
    }
}
```

```cpp
void HashTable::insert(int n) {
  cout << INSERT_STR << n << endl;
  //find the first available spot
  int idx = h(n) % _size;
  while (_ht[idx] > 0) {
    cout << COL_STR << idx << endl;
    idx = (idx + 1) % _size;
  }
  _ht[idx] = n;
  _nItem++;
  if (_nItem > _size / 2 && _size * 2 <= 100) {
    _resize(_size * 2);
  }
}

bool HashTable::exist(int n) {
  int idx = h(n) % _size;
  int nCollision = 0;
  while (_ht[idx] != n && _ht[idx] != 0) {
    idx = (idx + 1) % _size;
  }
  return _ht[idx] == n;
}
```

```cpp
class Graph {
    int numNodes;
    vector<vector<int>> adjList;

public:
    Graph(int nodes) {
        numNodes = nodes;
        adjList.resize(nodes);
    }

    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u); // For an undirected grap
    }

    void BFS() {
        vector<bool> visited(numNodes, false);
        vector<int> parent(numNodes, -1); // Initialize al

        for (int start = 0; start < numNodes; start++) {
            if (!visited[start]) {
                queue<int> frontier;
                frontier.push(start);
                visited[start] = true;

                while (!frontier.empty()) {
                    int v = frontier.front();
                    frontier.pop();

                    for (int w : adjList[v]) {
                        if (!visited[w]) {
                            visited[w] = true;
                            parent[w] = v;
                            frontier.push(w);
                        }
                    }
                }
            }
```

```cpp
//n does not necasarrily exist in the hash table
void HashTable::remove(int n) {
  cout << REMOVE_STR << n << endl;

  if (!exist(n))
  {
    cout << "Fail to remove " << n << endl;
    return;
  }

  int idx = h(n) % _size;
  while (_ht[idx] != n) {
    idx++;
  }
  _ht[idx] = -1;
  _nItem--;
  if (_nItem < _size / 4 && _size / 2 >= HT_MIN_SIZE) {
    _resize(_size / 2);
  }
```

```cpp
void HashTable::_resize(int newSize) {
    cout << RESIZE_STR << newSize << endl;
    int* new_table = new int[newSize];
    for (int i = 0; i < newSize; i++) {
      new_table[i] = 0;
    }

    for (int i = 0; i < _size; i++) {
      if (_ht[i] > 0) {
        cout << INSERT_STR << _ht[i] << endl;
        int idx = h(_ht[i]) % newSize;
        while (new_table[idx] > 0) {
          cout << COL_STR << idx << endl;
          idx = (idx + 1) % newSize;
        }
        new_table[idx] = _ht[i];
      }
    }
    _ht = new_table;
    _size = newSize;
}
```

```cpp
void dfs(int at, vector<bool>& visited, queue<int>& visitedNodes, const
    visited[at] = true;
    forward_list<GraphEdge> edges = g.edges_from(at);
    for(GraphEdge edge : edges){
        if(!visited[edge.dest()]){
            dfs(edge.dest(), visited, visitedNodes, g);
        }
    }
    visitedNodes.push(at);
}

vector<int> topologicalSort(const Graph& g){
    int N = g.num_vertices();
    vector<bool> visited(N, false);
    vector<int> ordering(N, 0);
    int i = N - 1;

    for(int at = 0; at < N; at++){
        if(!visited[at]){
            queue<int> visitedNodes;
            dfs(at, visited, visitedNodes, g);
            while(!visitedNodes.empty()){
                ordering[i] = visitedNodes.front();
                visitedNodes.pop();
                i--;
            }
        }
    }
    return ordering;
}
```

```cpp
void CircularList::insertHead(int n)
{
  ListNode* newNode = new ListNode(n);
  // modify this
  if (!_head) {
    _head = newNode;
    newNode->_next = _head;
  }
  else if(!_head->_next){
    ListNode* tail = _head->_next;
    newNode->_next = _head;
    _head->_next = newNode;
  }
  else {
    ListNode* temp = _head->_next; // sec
    _head->_next = newNode; // make head
    newNode->_next = temp; // new node po
    swap(_head->_item, newNode->_item);
  }
};
```

```cpp
void CircularList::removeHead()
{
  // modify this
  if (!_head) {
    return;
  }
  if (_head == _head->_next) { // one node
    delete _head;
    _head = nullptr;
    return;
  }
  ListNode* current = _head->_next; // cur
  ListNode* thirdNode = current->_next;
  swap(_head->_item, current->_item); // s
  _head->_next = thirdNode;
  delete current;
}
```