```cpp
bool CircularList::exist(int n) {
    // modify this
    ListNode* cur = _head;
    for (int i = 0; i < _size; i++) {
        if (cur->_item == n) {
            return true;
        }
        cur = cur->_next;
    }
    return false;
}

int CircularList::headItem()
{
    // modify this
    if (_size > 1) {
        return _head->_next->_item;
    }
    else {
        return _head->_item;
    }
}

void CircularList::advanceHead()
{
    // modify this
    if (_size == 0 || _size == 1) {
        return;
    }
    else {
        _head = _head->_next;
    }
}

void CircularList::insertHead(int n)
{
    // modify this
    if (_size == 0) {
        _head = new ListNode(n);
        _head->_next = _head;
        _size++;
    }
    else if (_size == 1) {
        _head->_next = new ListNode(n);
        _head->_next->_next = _head;
        _size++;
    }
    else {
        ListNode* temp = _head->_next;
        _head->_next = new ListNode(n);
        _head->_next->_next = temp;
        _size++;
    }
};
void CircularList::removeHead()
{
    // modify this
    if (_size == 0) {
        return;
    }
    else if (_size == 1) {
        delete(_head);
        _head = NULL;
        _size--;
    }
    else {
        ListNode* temp = _head->_next;
        _head->_next = _head->_next->_next;
        delete(temp);
        _size--;
    }
}
```

```cpp
bool HashTable::exist(int n) {
    int idx = h(n) % _size;
    int nCollision = 0;
    while (_ht[idx] != n) {
        idx++;
        idx %= _size;
        if (_ht[idx] == 0) {
            return false;
        }
        nCollision++;
        if (nCollision == _size) {
            return false;
        }
    }
    return true;
}
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++;
        idx %= _size;
    }
    _ht[idx] = -1;
    _nItem--;
    if (_nItem < _size / 4 && _size > 8) {
        int Size = _size;
        _resize(_size / 2);
        int* new_ht = new int[_size];
        for (int i = 0; i < _size; i++) {
            new_ht[i] = 0;
        }
        _nItem = 0;
        int* temp = _ht;
        _ht = new_ht;
        for (int i = 0; i < Size; i++) {
            if (temp[i] > 0) {
                insert(temp[i]);
            }
        }
        delete(temp);
    }
}
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 %= _size;
    }
    _ht[idx] = n;
    _nItem++;
    if (_nItem > _size / 2) {
        _resize(_size * 2);
        int* new_ht = new int[_size];
        for (int i = 0; i < _size; i++) {
            new_ht[i] = 0;
        }
        _nItem = 0;
        int* temp = _ht;
        _ht = new_ht;
        for (int i = 0; i < _size / 2; i++) {
            if (temp[i] > 0) {
                insert(temp[i]);
            }
        }
        delete(temp);
    }
```

```cpp
void List::insertTail(int n)
{
    if (!_size) {
        _head = new ListNode(n);
        _tail = _head;
        _mid = _head;
    }
    else {
        _tail->_next = new ListNode(n);
        _tail->_next->_prev = _tail;
        _tail = _tail->_next;
        if (_size % 2 == 0) {
            _mid = _mid->_next;
        }
    }
    _size++;
};

void List::removeHead()
{
    if (_size > 0) {
        ListNode* temp = _head;
        _head = _head->_next;
        if (_head) {
            _head->_prev = NULL;
        }
        if (_size % 2 == 0) {
            _mid = _mid->_next;
        }
        delete(temp);
        _size--;
    }
}
```

```cpp
template <class T>
TreeNode<T>* BinarySearchTree<T>::_leftRotation(TreeNode<T>* node)
{
    if (!node || !node->_right) {
        return node;
    }
    TreeNode<T>* temp = node->_right;
    node->_right = temp->_left;
    temp->_left = node;
    updateNode(node);
    updateNode(temp);
    return temp;
}

template <class T>
TreeNode<T>* BinarySearchTree<T>::_rightRotation(TreeNode<T>* node)
{
    if (!node || !node->_left) {
        return node;
    }
    TreeNode<T>* temp = node->_left;
    node->_left = temp->_right;
    temp->_right = node;
    updateNode(node);
    updateNode(temp);
    return temp;
}
```

```cpp
int height(TreeNode<T>* node) {
    return node ? node->_height : -1;
}

int weight(TreeNode<T>* node) {
    return node ? node->_weight : 0;
}

int getBalance(TreeNode<T>* node) {
    return node ? height(node->_left) - height(node->_right) : 0;
}

void updateNode(TreeNode<T>* node) {
    if (node) {
        node->_height = 1 + std::max(height(node->_left), height(node->_right));
        node->_weight = 1 + weight(node->_left) + weight(node->_right);
    }
}
```

```cpp
template <class T>
TreeNode<T>* BinarySearchTree<T>::_insert(TreeNode<T>* current, T x) {


    if (current->_item > x)
    {
        if (current->_left)
            current->_left = _insert(current->_left, x);
        else
        {
            current->_left = new TreeNode<T>(x);
            _size++;
        }
    }
    else if (x > current->_item) {
        if (current->_right)
            current->_right = _insert(current->_right, x);
        else
        {
            current->_right = new TreeNode<T>(x);
            _size++;
        }
    }
    else
        return current; // When the node already existed in the tree

    updateNode(current);
    int balance = getBalance(current);
    if (balance > 1 && x <= current->_left->_item) {
        return _rightRotation(current);
    }
    else if (balance < -1 && x >= current->_right->_item) {
        return _leftRotation(current);
    }
    else if (balance > 1 && x > current->_left->_item) {
        current->_left = _leftRotation(current->_left);
        return _rightRotation(current);
    }
    else if (balance < -1 && x < current->_right->_item) {
        current->_right = _rightRotation(current->_right);
        return _leftRotation(current);
    }
    else {
        return current;
    }
}
```

```cpp
void List::split(List* otherList) {
    // TODO: Implement splitting of the
    //       The first half of the list
    //       The second half of the list
    //       It is guaranteed that the o
    //
    //       No need to print within spl

    // ENTER YOUR ANSWER BELOW.
    int Size = _size / 2;
    ListNode* cur = _head;
    for (int i = 0; i < Size - 1; i++) {
        cur = cur->_next;
    }

    ListNode* start = cur->_next;
    cur->_next = NULL;
    otherList->_head = start;
    otherList->_size = _size - Size;
    _size = Size;
   // ENTER YOUR ANSWER ABOVE.
}
```

```cpp
void List::merge(List* otherList) {
    // TODO: Implement merging of current list and otherList.
    //       The otherList should be empty after merging.
    //       It is guaranteed that both lists are non-empty.
    //
    //       No need to print within merge()
    // ENTER YOUR ANSWER BELOW.
    List result;
    ListNode* cur1 = _head;
    ListNode* cur2 = otherList->_head;
    while (_head && otherList->_head) {
        if (cur1->_item > cur2->_item) {
            result.insertHead(cur2->_item);
            otherList->removeHead();
            cur2 = otherList->_head;
        }
        else {
            result.insertHead(cur1->_item);
            removeHead();
            cur1 = _head;
        }
    }
    if (!_head && !otherList->_head) {
        while (result._head) {
            insertHead(result._head->_item);
            result.removeHead();
        }
    }
    else if (!_head && otherList->_head) {
        while (otherList->_head) {
            insertHead(otherList->_head->_item);
            otherList->removeHead();
        }
        while (result._head) {
            insertHead(result._head->_item);
            result.removeHead();
        }
    }
    else if (_head && !otherList->_head) {
        while (result._head) {
            insertHead(result._head->_item);
            result.removeHead();
        }
    }

    // ENTER YOUR ANSWER ABOVE.
}
```

```cpp
void List::mergeSort() {
    if (_size == 1) {
        return;
    }
    else {
        // modify this function
        // each time you split a list
        cout << SPLITSTR;
        // original unsplitted list print
        print();
        cout << INTOSTR << endl;
        // original splitted list print (but it's halved)
        // the other splitted list print
        List* otherList = new List();
        split(otherList);
        print();
        otherList->print();
        mergeSort();
        otherList->mergeSort();
        // after you sorted the two lists
        cout << MERGESTR << endl;
        // original splitted list print (but it's halved)
        // the other splitted list print
        print();
        otherList->print();
        // after you merged the above two sorted lists
        cout << INTOSTR;
        // print the merged sorted single list
        merge(otherList);
        print();
```

```cpp
void List::bubbleSort() {

    // TODO: Implement bubble sort on thi
    //       Print the linked list after
    //       using print(false) function
    //       The list should be sorted in

    // ENTER YOUR ANSWER BELOW.
    if (_size == 0 || _size == 1) {
        print();
    }
    else {
        for (int i = 0; i < _size - 1; i++) {
            if (_head->_item > _head->_next->_item) {
                ListNode* temp = _head;
                _head = _head->_next;
                temp->_next = _head->_next;
                _head->_next = temp;
            }
            ListNode* cur = _head->_next;
            ListNode* parent = _head;
            while (cur->_next) {
                if (cur->_item > cur->_next->_item) {
                    parent->_next = cur->_next;
                    cur->_next = cur->_next->_next;
                    parent->_next->_next = cur;
                    parent = parent->_next;
                }
                else {
                    parent = cur;
                    cur = cur->_next;
                }
            }
            print();
```

```cpp
void partyGoing(string filename) {
    ifstream inFile(filename.c_str());
    if (!inFile) {
        cout << "File not found!" << endl;
        return;
    }

    cout << "For the file input of " << '"' << filename << '"' << '.' << endl;
    int A, B, C, D;
    inFile >> A >> B >> C >> D;

    bool** adjMatrix = new bool* [A + 1];

    int* totalFriends = new int[A + 1];

    int* goingFriends = new int[A + 1];

    bool* isGoing = new bool[A + 1];

    int* q = new int[A + 1];
    int qFront = 0;
    int qRear = 0;

    for (int i = 0; i <= A; i++) {
        adjMatrix[i] = new bool[A + 1];
        totalFriends[i] = 0;
        goingFriends[i] = 0;
        isGoing[i] = false;
        for (int j = 0; j <= A; j++) {
            adjMatrix[i][j] = false;
        }
    }

    for (int k = 0; k < B; k++) {
        int X, Y;
        inFile >> X >> Y;
        if (!adjMatrix[X][Y]) {
            adjMatrix[X][Y] = true;
            adjMatrix[Y][X] = true;
            totalFriends[X]++;
            totalFriends[Y]++;
        }
    }
    inFile.close();
```

```cpp
void Graph::BF(int source) {
    int* dist = new int[_nv];
    for (int i = 0; i < _nv; i++) {
        dist[i] = INFINITY_WEIGHT;
        if (i == source) {
            dist[i] = 0;
        }
    }

    for (int i = 0; i < _nv + 1; i++) {
        int relax = 0;
        for (int j = 0; j < _ne; j++) {
            if (dist[_edges[j].v1] + _edges[j].weight < dist[_edges[j].v2]) {
                dist[_edges[j].v2] = dist[_edges[j].v1] + _edges[j].weight;
                relax++;
            }
        }

        if (relax == 0) {
            break;
        }

        for (int j = 0; j < _nv - 1; j++) {
            cout << dist[j] << ",";
        }
        cout << dist[_nv - 1];
        cout << endl;

        if (i == _nv) {
            cout << NEGATIVECYCLESTRING << endl;
        }
    }
}
```

```cpp
    isGoing[D] = true;
    q[qRear++] = D;

    while (qFront != qRear) {
        int u = q[qFront++];

        for (int v = 1; v <= A; v++) {
            if (adjMatrix[u][v]) {
                goingFriends[v]++;
                if (isGoing[v]) {
                    continue;
                }

                if (totalFriends[v] > 0 && (goingFriends[v] * 2 >= totalFriends[v])) {
                    isGoing[v] = true;
                    q[qRear++] = v;
                }
            }
        }
    }

    if (isGoing[C]) {
        cout << C << " will go for the party" << endl;
    }
    else {
        cout << C << " will not go for the party" << endl;
    }

    cout << "Classmate(s) who will go to the party is/are:" << endl;
    for (int i = 1; i <= A; i++) {
        if (isGoing[i]) {
            cout << i << " ";
        }
    }
    cout << endl;

    for (int i = 0; i <= A; i++) {
        delete[] adjMatrix[i];
    }
    delete[] adjMatrix;
    delete[] totalFriends;
    delete[] goingFriends;
    delete[] isGoing;
    delete[] q;
}
```