TNode.h

```
#ifndef ____TNODE____H____
#define ____TNODE____H____

class TNode {
	public: int Data;
	public: TNode* Parent;
	public: TNode* Left;
	public: TNode* Right;

	public: TNode();
};

#endif
```

TNode.cpp

```
#include "TNode.h"

TNode::TNode() {
	this->Data = 0;
	this->Parent = this->Left = this->Right = 0;
}
```

Field.h

```
#ifndef ____FIELD____H____
#define ____FIELD____H____
#include "TNode.h"
#include <cstdlib>

enum TChar {SPACE, LEFT, CENTER, RIGHT, NUMBER};

class Field {
	private: const int cellWidth;
	private: int height;
	private: int width;
	private: TNode*** matrix;
	private: TChar** chars;

	public: Field(TNode*, int, int);
	private: void init(TNode*);
	public: ~Field();
	private: void fillMatrix(TNode*, int, int&);
	private: void fillChars();
	private: void printChars(char, int);
	private: TNode* getNextNode(int, int);
	private: int getSpaces(int);
	public: void display(TNode* node = NULL);
};
```

```
#endif
```

Field.cpp

```
#include "Field.h"
#include <iostream>
using namespace std;

Field::Field(TNode* node, int _height, int _width)
	: cellWidth(4) {
	this->height = _height;
	this->width = _width;
	this->init(node);
}

void Field::init(TNode* node) {
	this->matrix = new TNode** [this->height];
	this->chars = new TChar* [this->height];
	for (int i = 0; i < this->height; ++i) {
		this->matrix[i] = new TNode* [this->width];
		this->chars[i] = new TChar [this->width];
		for (int j = 0; j < this->width; ++j) {
			this->matrix[i][j] = NULL;
			this->chars[i][j] = SPACE;
		}
	}
	int index = 0;
	this->fillMatrix(node, 0, index);
	this->fillChars();
}

Field::~Field() {
	for (int i = 0; i < this->height; ++i) {
		delete [] this->matrix[i];
		delete [] this->chars[i];
	}
	delete [] this->matrix;
	delete [] this->chars;
}

void Field::fillMatrix(TNode* node, int row, int& col) {
	if (node == NULL) {
		return;
	}
	fillMatrix(node->Left, row+1, col);
	this->matrix[row][col++] = node;
	fillMatrix(node->Right, row+1, col);
}

void Field::fillChars() {
	for (int i = 0; i < this->height; ++i) {
		for (int j = 0; j < this->width; ++j) {
			if (this->matrix[i][j] == NULL) {
```

```
                        continue;
                  }
                  this->chars[i][j] = NUMBER;
                  if (this->matrix[i][j]->Left != NULL) {
                        for (int k = j-1; k >= 0; --k) {
                              if (this->matrix[i+1][k] == NULL) {
                                    this->chars[i][k] = CENTER;
                                    continue;
                              }
                              this->chars[i][k] = LEFT;
                              break;
                        }
                  }
                  if (this->matrix[i][j]->Right != NULL) {
                        for (++j; j < this->width; ++j) {
                              if (this->matrix[i+1][j] == NULL) {
                                    this->chars[i][j] = CENTER;
                                    continue;
                              }
                              this->chars[i][j] = RIGHT;
                              break;
                        }
                  }
            }
      }
}

void Field::printChars(char C, int count) {
      for (int i = 0; i < count; ++i) {
            cout << C;
      }
}

TNode* Field::getNextNode(int i, int j) {
      TNode* node = NULL;
      for (; j < this->width; ++j) {
            if (this->matrix[i][j] != NULL) {
                  node = this->matrix[i][j];
                  break;
            }
      }
      return node;
}

int Field::getSpaces(int number) {
      int spaces = this->cellWidth;
      if (number < 0) {
            number *= -1;
            --spaces;
      }
      while (number > 0) {
            number /= 10;
            --spaces;
      }
      return spaces;
}
```

```
void Field::display(TNode* node) {
	int data, spaces;
	TNode* nextNode;
	for (int i = 0; i < this->height; ++i) {
		for (int j = 0; j < this->width; ++j) {
			switch (this->chars[i][j]) {
				case SPACE:
					this->printChars(' ', this->cellWidth);
					break;
				case LEFT:
					nextNode = getNextNode(i, j);
					data = nextNode->Data;
					spaces = nextNode == node ? 2 : getSpaces(data);
					this->printChars(' ', spaces/2);
					this->printChars((char)218, 1);
					this->printChars((char)196, this->cellWidth-1);
					break;
				case CENTER:
					this->printChars((char)196, this->cellWidth);
					break;
				case RIGHT:
					this->printChars((char)196, this->cellWidth-1);
					this->printChars((char)191, 1);
					this->printChars(' ', spaces/2 + spaces%2);
					break;
				case NUMBER:
					if (this->matrix[i][j]->Left == NULL) {
						data = this->matrix[i][j]->Data;
						spaces = this->matrix[i][j] == NULL ?
							this->cellWidth-2 : getSpaces(data);
						this->printChars(' ', spaces/2);
					}

					if (this->matrix[i][j] == node) { cout << "##"; }
					else { cout << data; }

					if (this->matrix[i][j]->Right == NULL) {
						this->printChars(' ', spaces/2 + spaces%2);
					}
					break;
			}
		}
		cout << endl;
	}
}
```

Tree.h

```
#ifndef ____TREE____H____
#define ____TREE____H____
```

```
#include "TNode.h"
#include "Field.h"
#include <cstdlib>

enum ChildNode { TREE25, TREE26LEFT, TREE26RIGHT, TREE27, TREE28, TREE29 };

class Tree {
	private: TNode* root;
	private: TNode* current;
	private: int nodeCount;
	private: int level;
	private: Field* field;

	public: Tree();
	public: ~Tree();
	public: void make();
	public: void make(int[], int, int, ChildNode); //Tree 25, 26, 27, 28, 29
	public: void make(int); //Tree 30
	private: void free();
	public: void display(TNode* node = NULL);
	public: void infix(); //Tree 12
	public: void prefix(); //Tree 13
	public: void postfix(); //Tree 14
	public: void infixToN(int&, int); //Tree 15
	public: void postfixFromN(int&, int); //Tree 16
	public: void prefixBetween(int&, int, int); //Tree 17
	public: int getNodeCount() const; //Tree 2
	public: int getLeftCount(bool isLeft = false); //Tree 5
	public: int getLeafCount(); //Tree 6
	public: int getRightLeafCount(bool isRight = false); // Tree 8
	public: int getNodeCountK(int); //Tree 3
	public: int getLeafCountK(int); //Tree 20
	private: void setLevel(int currentLevel = 0);
	public: int getLevel(); //Tree 9
	public: int getLevelNodeCount(int, int level = 0); //Tree 18
	public: int getDataSum(); //Tree 4
	public: int getLeafDataSum(); //Tree 7
	public: void levelNodeCountToArr(int[], int); //Tree 10
	public: void levelNodeSumToArr(int[], int); //Tree 11
	public: int getMaxData(); //Tree 19
	public: int getMinData(); //Tree 20
	public: int getMinLeafData(); //Tree 21
	public: int getMaxInternalData(); //Tree 22
	public: TNode* getFirstNodePrefix(int); //Tree 23
	public: TNode* getLastNodeInfix(int); //Tree 24
	public: bool hasOddData(); //Tree 24
	public: int getMaxOddData(); //Tree 24
};

#endif

Tree.cpp
#include "Tree.h"
#include <iostream>
using namespace std;
```

```
Tree::Tree() {
	this->root = NULL;
	this->field = NULL;
	this->nodeCount = 0;
	this->level = -1;
}

Tree::~Tree() {
	if (this->current != this->root) {
		this->current = this->root;
	}
	this->free();
	delete this->field;
}

void Tree::make() {
	if (this->root == NULL) {
		this->root = new TNode();
		this->nodeCount++;
		cout << "Root's data: ";
		cin >> this->root->Data;
		this->current = this->root;
	}
	int answer;
	cout << "Where to go? (0-exit; 1-left; 2-right; 3-parent):\t";
	cin >> answer;
	if (answer == 0) { this->current = this->root; return; }
	if (answer > 3 || answer < 0) { this->make(); return; }
	if (answer == 3) { this->current = this->current->Parent; this->make();
return; }
	TNode* newLeaf = new TNode();
	this->nodeCount++;
	cout << "data:\t";
	cin >> newLeaf->Data;
	newLeaf->Parent = this->current;
	switch (answer) {
		case 1: this->current->Left = newLeaf; break;
		case 2: this->current->Right = newLeaf; break;
	}
	this->current = newLeaf;
	this->make();
}

void Tree::make(int arr[], int index, int N, ChildNode child) {
	if (index >= N) {
		this->current = this->root;
		return;
	}
	if (this->root == NULL) {
		this->root = new TNode();
		this->nodeCount++;
		this->root->Data = arr[index++];
		this->current = this->root;
	}
	if (index >= N) {
		this->current = this->root;
```

```
		return;
	}
	TNode* newLeaf = new TNode();
	this->nodeCount++;
	newLeaf->Data = arr[index++];
	newLeaf->Parent = this->current;
	switch (child) {
		case TREE25:
			this->current->Right = newLeaf;
			break;
		case TREE26LEFT:
			this->current->Left = newLeaf;
			child = TREE26RIGHT;
			break;
		case TREE26RIGHT:
			this->current->Right = newLeaf;
			child = TREE26LEFT;
			break;
		case TREE27:
			if (this->current->Data % 2 != 0) {
				this->current->Left = newLeaf;
			} else {
				this->current->Right = newLeaf;
			}
			break;
		case TREE28:
			this->current->Left = newLeaf;
			if (index < N) {
				newLeaf = new TNode();
				this->nodeCount++;
				newLeaf->Data = arr[index++];
				newLeaf->Parent = this->current;
				this->current->Right = newLeaf;
			}
			break;
		case TREE29:
			TNode* tempNode = NULL;
			if (index < N) {
				tempNode = new TNode();
				this->nodeCount++;
				tempNode->Data = arr[index++];
				tempNode->Parent = this->current;
			}
			if (this->current->Data % 2 != 0) {
				this->current->Left = newLeaf;
				this->current->Right = tempNode;
			} else {
				this->current->Right = newLeaf;
				this->current->Left = tempNode;
			}
			newLeaf = tempNode;
			break;
	}
	this->current = newLeaf;
	this->make(arr, index, N, child);
}
```

```
void Tree::make(int K) {
	if (K < 1) {
		this->current = this->root;
		return;
	}
	TNode* newLeaf = new TNode();
	this->nodeCount++;
	newLeaf->Data = K;
	newLeaf->Parent = this->current;
	if (this->root == NULL) {
		this->current = this->root = newLeaf;
	} else if (2*K > this->current->Data) {
		this->current->Right = newLeaf;
	} else {
		this->current->Left = newLeaf;
	}

	if (K == 1) {
		this->current = this->root;
		return;
	}
	this->current = newLeaf;
	this->make(K/2);
	if (K % 2 != 0) {
		this->current = newLeaf;
		this->make(K - K/2);
	}
}

void Tree::free() {
	if (this->current == NULL) {
		this->current = this->root;
		return;
	}
	TNode* node = this->current;
	this->current = node->Left;
	this->free();
	this->current = node->Right;
	this->free();
	this->current = node;
	delete this->current;
	this->nodeCount--;
	this->current = NULL;
}

void Tree::display(TNode* node) {
	if (this->root == NULL) {
		return;
	}
	if (this->field == NULL) {
		this->field = new Field(this->root, this->getLevel()+1, this-
>getNodeCount()+1);
	}
	field->display(node);
}
```

```
void Tree::infix() {
	if (this->current == NULL) {
		this->current = this->root;
		return;
	}
	TNode* node = this->current;
	this->current = node->Left;
	this->infix();
	cout << node->Data << '\t';
	this->current = node->Right;
	this->infix();
}

void Tree::prefix() {
	if (this->current == NULL) {
		this->current = this->root;
		return;
	}
	TNode* node = this->current;
	cout << node->Data << '\t';
	this->current = node->Left;
	this->prefix();
	this->current = node->Right;
	this->prefix();
}

void Tree::postfix() {
	if (this->current == NULL) {
		this->current = this->root;
		return;
	}
	TNode* node = this->current;
	this->current = node->Left;
	this->postfix();
	this->current = node->Right;
	this->postfix();
	cout << node->Data << '\t';
}

void Tree::infixToN(int &index, int N) {
	if (this->current == NULL) {
		this->current = this->root;
		return;
	}
	TNode* node = this->current;
	this->current = node->Left;
	this->infixToN(index, N);
	++index;
	if (index <= N) {
		cout << node->Data << '\t';
	} else {
		this->current = this->root;
		return;
	}
	this->current = node->Right;
```

```
	this->infixToN(index, N);
}

void Tree::postfixFromN(int& index, int N) {
	if (this->current == NULL) {
		this->current = this->root;
		return;
	}
	TNode* node = this->current;
	this->current = node->Left;
	this->postfixFromN(index, N);
	this->current = node->Right;
	this->postfixFromN(index, N);
	++index;
	if (index >= N) {
		cout << node->Data << '\t';
	}
}

void Tree::prefixBetween(int& index, int N1, int N2) {
	if (this->current == NULL) {
		this->current = this->root;
		return;
	}
	TNode* node = this->current;
	++index;
	if (N1 <= index && index < N2) {
		cout << node->Data << '\t';
	} else if (index > N2) {
		this->current = this->root;
		return;
	}
	this->current = node->Left;
	this->prefixBetween(index, N1, N2);
	this->current = node->Right;
	this->prefixBetween(index, N1, N2);
}

int Tree::getNodeCount() const {
	return this->nodeCount;
}

int Tree::getLeftCount(bool isLeft) {
	if (this->current == NULL) {
		this->current = this->root;
		return 0;
	}
	int count = isLeft ? 1 : 0;
	TNode* node = this->current;
	this->current = node->Left;
	count += this->getLeftCount(true);
	this->current = node->Right;
	count += this->getLeftCount();
	return count;
}
```

```
int Tree::getLeafCount() {
	if (this->current == NULL) {
		this->current = this->root;
		return 0;
	}
	int count = this->current->Left == NULL && this->current->Right == NULL ?
1 : 0;
	TNode* node = this->current;
	this->current = node->Left;
	count += this->getLeafCount();
	this->current = node->Right;
	count += this->getLeafCount();
	return count;
}

int Tree::getRightLeafCount(bool isRight) {
	if (this->current == NULL) {
		this->current = this->root;
		return 0;
	}
	int count = 0;
	if (isRight && this->current->Left == NULL && this->current->Right ==
NULL) {
		count = 1;
	}
	TNode* node = this->current;
	this->current = node->Left;
	count += this->getRightLeafCount();
	this->current = node->Right;
	count += this->getRightLeafCount(true);
	return count;
}

int Tree::getNodeCountK(int K) {
	if (this->current == NULL) {
		this->current = this->root;
		return 0;
	}
	TNode* node = this->current;
	int count = node->Data == K ? 1 : 0;
	this->current = node->Left;
	count += this->getNodeCountK(K);
	this->current = node->Right;
	count += this->getNodeCountK(K);
	return count;
}

int Tree::getLeafCountK(int K) {
	if (this->current == NULL) {
		this->current = this->root;
		return 0;
	}
	TNode* node = this->current;
	int count = 0;
	if (node->Data == K && node->Left == NULL && node->Right == NULL) {
		count = 1;
```

```
	}
	this->current = node->Left;
	count += this->getLeafCountK(K);
	this->current = node->Right;
	count += this->getLeafCountK(K);
	return count;
}

void Tree::setLevel(int currentLevel) {
	if (this->current == NULL) {
		this->current = this->root;
		return;
	}
	if (currentLevel > this->level) {
		this->level = currentLevel;
	}
	TNode* node = this->current;
	this->current = node->Left;
	this->setLevel(currentLevel+1);
	this->current = node->Right;
	this->setLevel(currentLevel+1);
}

int Tree::getLevel() {
	if (this->level < 0) {
		this->setLevel();
	}
	return this->level;
}

int Tree::getLevelNodeCount(int L, int level)  {
	if (this->current == NULL) {
		this->current = this->root;
		return 0;
	}
	TNode* node = this->current;
	this->current = node->Left;
	int count = this->getLevelNodeCount(L, level+1);
	if (level == L) {
		cout << node->Data << '\t';
		++count;
	} else if (level > L) {
		this->current = this->root;
		return count;
	}
	this->current = node->Right;
	count += this->getLevelNodeCount(L, level+1);
	return count;
}

int Tree::getDataSum() {
	if (this->current == NULL) {
		this->current = this->root;
		return 0;
	}
	int sum = this->current->Data;
```

```
	TNode* node = this->current;
	this->current = node->Left;
	sum += this->getDataSum();
	this->current = node->Right;
	sum += this->getDataSum();
	return sum;
}

int Tree::getLeafDataSum() {
	if (this->current == NULL) {
		this->current = this->root;
		return 0;
	}
	int sum = 0;
	if (this->current->Left == NULL && this->current->Right == NULL) {
		sum = this->current->Data;
	}
	TNode* node = this->current;
	this->current = node->Left;
	sum += this->getLeafDataSum();
	this->current = node->Right;
	sum += this->getLeafDataSum();
	return sum;
}

void Tree::levelNodeCountToArr(int arr[], int index) {
	if (this->current == NULL) {
		this->current = this->root;
		return;
	}
	arr[index]++;
	TNode* node = this->current;
	this->current = node->Left;
	this->levelNodeCountToArr(arr, index+1);
	this->current = node->Right;
	this->levelNodeCountToArr(arr, index+1);
}

void Tree::levelNodeSumToArr(int arr[], int index) {
	if (this->current == NULL) {
		this->current = this->root;
		return;
	}
	arr[index] += this->current->Data;
	TNode* node = this->current;
	this->current = node->Left;
	this->levelNodeSumToArr(arr, index+1);
	this->current = node->Right;
	this->levelNodeSumToArr(arr, index+1);
}

int Tree::getMaxData() {
	int maximal;
	if (this->current->Left == NULL && this->current->Right == NULL) {
		maximal = this->current->Data;
		this->current = this->root;
```

```
		return maximal;
	}
	TNode* node = this->current;
	int data;
	maximal = node->Data;
	if (node->Left != NULL) {
		this->current = node->Left;
		data = this->getMaxData();
		if (data > maximal) {
			maximal = data;
		}
	}
	if (node->Right != NULL) {
		this->current = node->Right;
		data = this->getMaxData();
		if (data > maximal) {
			maximal = data;
		}
	}
	return maximal;
}

int Tree::getMinData() {
	int minimal;
	if (this->current->Left == NULL && this->current->Right == NULL) {
		minimal = this->current->Data;
		this->current = this->root;
		return minimal;
	}
	TNode* node = this->current;
	int data;
	minimal = node->Data;
	if (node->Left != NULL) {
		this->current = node->Left;
		data = this->getMinData();
		if (data < minimal) {
			minimal = data;
		}
	}
	if (node->Right != NULL) {
		this->current = node->Right;
		data = this->getMinData();
		if (data < minimal) {
			minimal = data;
		}
	}
	return minimal;
}

int Tree::getMinLeafData() {
	int minimal;
	if (this->current->Left == NULL && this->current->Right == NULL) {
		minimal = this->current->Data;
		this->current = this->root;
		return minimal;
	}
```

```
	TNode* node = this->current;
	int data;
	bool inited = false;
	if (node->Left != NULL) {
		this->current = node->Left;
		minimal = this->getMinLeafData();
		inited = true;
	}
	if (node->Right != NULL) {
		this->current = node->Right;
		data = this->getMinLeafData();
		if (!inited) {
			minimal = data;
		} else if (data < minimal) {
			minimal = data;
		}
	}
	return minimal;
}

int Tree::getMaxInternalData() {
	if (this->current->Left == NULL && this->current->Right == NULL) {
		this->current = this->root;
		return 0;
	}
	TNode* node = this->current;
	int data, maximal = node->Data;
	if (node->Left != NULL && (node->Left->Left != NULL || node->Left->Right
!= NULL)) {
		this->current = node->Left;
		data = this->getMaxInternalData();
		if (data > maximal) {
			maximal = data;
		}
	}
	if (node->Right != NULL && (node->Right->Left != NULL || node->Right-
>Right != NULL)) {
		this->current = node->Right;
		data = this->getMaxInternalData();
		if (data > maximal) {
			maximal = data;
		}
	}
	return maximal;
}

TNode* Tree::getFirstNodePrefix(int data) {
	if (this->current == NULL) {
		this->current = this->root;
		return NULL;
	}
	TNode* node = this->current;
	if (node->Data == data) {
		return node;
	}
	this->current = node->Left;
```

```
	TNode* tempNode = this->getFirstNodePrefix(data);
	if (tempNode == NULL) {
		this->current = node->Right;
		tempNode = this->getFirstNodePrefix(data);
	}
	return tempNode;
}

TNode* Tree::getLastNodeInfix(int data) {
	if (this->current == NULL) {
		this->current = this->root;
		return NULL;
	}
	TNode* node = this->current;
	this->current = node->Left;
	TNode* resNode = this->getLastNodeInfix(data);
	if (node->Data == data) {
		resNode = node;
	}
	this->current = node->Right;
	TNode* tempNode = this->getLastNodeInfix(data);
	if (tempNode != NULL) {
		resNode = tempNode;
	}
	return resNode;
}

bool Tree::hasOddData() {
	if (this->current == NULL) {
		this->current = this->root;
		return false;
	}
	TNode* node = this->current;
	if (node->Data % 2 != 0) {
		return true;
	}
	this->current = node->Left;
	bool hasIt = this->hasOddData();
	if (!hasIt) {
		this->current = node->Right;
		hasIt = this->hasOddData();
	}
	return hasIt;
}

int Tree::getMaxOddData() {
	int maximal;
	if (this->current->Left == NULL && this->current->Right == NULL) {
		maximal = this->current->Data;
		this->current = this->root;
		return maximal;
	}
	TNode* node = this->current;
	int data;
	bool inited = false;
	if (node->Data % 2 != 0) {
```

```
			maximal = node->Data;
			inited = true;
		}
		if (node->Left != NULL) {
			this->current = node->Left;
			data = this->getMaxOddData();
			if (!inited) {
				maximal = data;
				inited = true;
			} else if (data > maximal) {
				maximal = data;
			}
		}
		if (node->Right != NULL) {
			this->current = node->Right;
			data = this->getMaxOddData();
			if (!inited) {
				maximal = data;
				inited = true;
			} else if (data > maximal) {
				maximal = data;
			}
		}
		return maximal;
}
```

Tree 1

```
#include "TNode.h"
#include <iostream>
using namespace std;

int main() {
		TNode* P1 = new TNode();
		cout << "P1->Data = ";
		cin >> P1->Data;

		P1->Left = new TNode();
		cout << "P1->Left->Data = ";
		cin >> P1->Left->Data;

		P1->Right = new TNode();
		cout << "P1->Right->Data = ";
		cin >> P1->Right->Data;

		cout << "P1->Data = " << P1->Data << endl;
		cout << "P1->Left->Data = " << P1->Left->Data << endl;
		cout << "P1->Right->Data = " << P1->Right->Data << endl;
		cout << "P1->Left = " << P1->Left << endl;
		cout << "P1->Right = " << P1->Right << endl;

		delete P1->Left;
		delete P1->Right;
		delete P1;
```

```
	P1 = NULL;
	return 0;
}
```

Tree 2

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n' << tree.getNodeCount();
	return 0;
}
```

Tree 3

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	int K;
	cout << "\nK = ";
	cin >> K;
	cout << tree.getNodeCountK(K);
	return 0;
}
```

Tree 4

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n' << tree.getDataSum();
	return 0;
}
```

Tree 5

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n' << tree.getLeftCount();
	return 0;
}
```

Tree 6

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n' << tree.getLeafCount();
	return 0;
}
```

Tree 7

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n' << tree.getLeafDataSum();
	return 0;
}
```

Tree 8

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
```

```
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n' << tree.getRightLeafCount();
	return 0;
}
```

Tree 9

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n' << tree.getLevel();
	return 0;
}
```

Tree 10

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	int level = tree.getLevel();
	int* arr = new int [level+1];
	for (int i = 0; i <= level; ++i) {
		arr[i] = 0;
	}
	tree.levelNodeCountToArr(arr, 0);
	for (int i = 0; i <= level; ++i) {
		cout << arr[i] << '\t';
	}
	delete [] arr;
	arr = NULL;
	return 0;
}
```

Tree 11

```
#include "Tree.h"
#include <iostream>
using namespace std;
```

```
int main() {
	Tree tree;
	tree.make();
	tree.display();
	int level = tree.getLevel();
	int* arr = new int [level+1];
	for (int i = 0; i <= level; ++i) {
		arr[i] = 0;
	}
	tree.levelNodeSumToArr(arr, 0);
	for (int i = 0; i <= level; ++i) {
		cout << arr[i] << '\t';
	}
	delete [] arr;
	arr = NULL;
	return 0;
}
```

Tree 12

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	tree.infix();
	return 0;
}
```

Tree 13

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	tree.prefix();
	return 0;
}
```

Tree 14

```
#include "Tree.h"
```

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

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	tree.postfix();
	return 0;
}
```

Tree 15

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	int N;
	cout << "N = ";
	cin >> N;
	int index = 0;
	tree.infixToN(index, N);
	return 0;
}
```

Tree 16

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	int N;
	cout << "N = ";
	cin >> N;
	int index = 0;
	tree.postfixFromN(index, N);
	return 0;
}
```

Tree 17

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	int N1, N2;
	cout << "N1 = ";
	cin >> N1;
	cout << "N2 = ";
	cin >> N2;
	int index = 0;
	tree.prefixBetween(index, N1, N2);
	return 0;
}
```

Tree 18

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	int level = tree.getLevel();
	int L;
	cout << "L = ";
	cin >> L;
	int N = 0;
	if (L <= level+1) {
		N = tree.getLevelNodeCount(L);
	}
	cout << "\nN = " << N;
	return 0;
}
```

Tree 19

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
```

```
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	int maximal = tree.getMaxData();
	int count = tree.getNodeCountK(maximal);
	cout << "maximal = " << maximal << endl;
	cout << "count = " << count << endl;
	return 0;
}
```

Tree 20

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	int minimal = tree.getMinData();
	int count = tree.getLeafCountK(minimal);
	cout << "minimal = " << minimal << endl;
	cout << "leafCount = " << count << endl;
	return 0;
}
```

Tree 21

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	cout << tree.getMinLeafData();
	return 0;
}
```

Tree 22

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
```

```
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	cout << tree.getMaxInternalData();
	return 0;
}
```

Tree 23

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	int minimal = tree.getMinData();
	TNode* node = tree.getFirstNodePrefix(minimal);
	tree.display(node);
	return 0;
}
```

Tree 24

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	Tree tree;
	tree.make();
	tree.display();
	cout << '\n';
	TNode* node = NULL;
	if (tree.hasOddData()) {
		int maximalOdd = tree.getMaxOddData();
		node = tree.getLastNodeInfix(maximalOdd);
		tree.display(node);
	} else {
		cout << node;
	}
	return 0;
}
```

Tree 25

```
#include "Tree.h"
#include <iostream>
```

```
using namespace std;

int main() {
	int N;
	cout << "N = ";
	cin >> N;
	int* arr = new int [N];
	for (int i = 0; i < N; ++i) {
		cin >> arr[i];
	}
	Tree tree;
	tree.make(arr, 0, N, TREE25);
	tree.display();
	delete [] arr;
	arr = NULL;
	return 0;
}
```

Tree 26

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	int N;
	cout << "N = ";
	cin >> N;
	int* arr = new int [N];
	for (int i = 0; i < N; ++i) {
		cin >> arr[i];
	}
	Tree tree;
	tree.make(arr, 0, N, TREE26LEFT);
	tree.display();
	delete [] arr;
	arr = NULL;
	return 0;
}
```

Tree 27

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	int N;
	cout << "N = ";
	cin >> N;
	int* arr = new int [N];
	for (int i = 0; i < N; ++i) {
```

```
            cin >> arr[i];
      }
      Tree tree;
      tree.make(arr, 0, N, TREE27);
      tree.display();
      delete [] arr;
      arr = NULL;
      return 0;
}
```

Tree 28

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
      int N;
      cout << "N = ";
      cin >> N;
      int* arr = new int [N];
      for (int i = 0; i < N; ++i) {
            cin >> arr[i];
      }
      Tree tree;
      tree.make(arr, 0, N, TREE28);
      tree.display();
      delete [] arr;
      arr = NULL;
      return 0;
}
```

Tree 29

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
      int N;
      cout << "N = ";
      cin >> N;
      int* arr = new int [N];
      for (int i = 0; i < N; ++i) {
            cin >> arr[i];
      }
      Tree tree;
      tree.make(arr, 0, N, TREE29);
      tree.display();
      delete [] arr;
      arr = NULL;
      return 0;
```

```
}
```

Tree 30

```
#include "Tree.h"
#include <iostream>
using namespace std;

int main() {
	int N;
	cout << "N = ";
	cin >> N;
	Tree tree;
	tree.make(N);
	tree.display();
	return 0;
}
```