main.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <utility>
#include <random>

#include "vertex.h"
#include "LinkedList.h"
#include "custom_vec.h"



#define INF 0x3f3f3f3f

typedef std::pair<custom_vec<vertex*>, int> order;

//parses input file and constructs adjList and degreesList
void parseInput(std::ifstream& file, LinkedList<vertex*>*& adjList,
custom_vec<LinkedList<vertex*>*>*& degreeList);
order smallestLastOrdering(LinkedList<vertex*>* adjList,
custom_vec<LinkedList<vertex*>*>* degreeList);
order welshPowellOrdering(LinkedList<vertex*>* adjList,
custom_vec<LinkedList<vertex*>*>* degreeList);
order uniformRandomOrdering(LinkedList<vertex*>* adjList);
order largestLastOrdering(LinkedList<vertex*>* adjList,
custom_vec<LinkedList<vertex*>*>* degreeList);
order largestEccentricityOrdering(LinkedList<vertex*>* adjList);
order distanceFromHighestDegreeVertexOrdering(LinkedList<vertex*>* adjList,
custom_vec<LinkedList<vertex*>*>* degreeList);
int color_graph(custom_vec<vertex*>& ordering);

//utility functions
int min_index(custom_vec<LinkedList<vertex*>*>*& v){
    int min = INF;
    int index = 0;
    for(int i = 0; i < v->vsize(); i++) {
        if (v->at(i)->getLength() < min && v->at(i)->getLength() >= 1) {
            min = v->at(i)->getLength();
            index = i;
        }
    }
```

```
    return index;
}
int max_index(custom_vec<LinkedList<vertex*>*>*& v){
    int max = 0;
    int index = 0;
    for(int i = 0; i < v->vsize(); i++) {
        if (v->at(i)->getLength() > max && v->at(i)->getLength() >= 1) {
            max = v->at(i)->getLength();
            index = i;
        }
    }
    return index;
}
int calc_average_degree(custom_vec<vertex*>& colored_vertices){
    int total = 0;
    for(int i = 0; i < colored_vertices.vsize(); i++)
        total += colored_vertices.at(i)->origDegree;

    return total / colored_vertices.vsize();
}
int max_del_degree(custom_vec<vertex*>& colored_vertices){
    int max = 0;
    for(int i = 0; i < colored_vertices.vsize(); i++){
        if(colored_vertices.at(i)->currDegree > max)
            max = colored_vertices.at(i)->currDegree;
    }

    return max;
}
bool is_connected(vertex*& v, int key){
    v->P.currentToHead();
    for(auto iter = v->P.currentNode(); iter != nullptr; iter = iter->getNext()){
        if(iter->getData()->id == key)
            return true;
    }

    return false;
}
void print_results(custom_vec<vertex*> ordering, int num_colors){

    for(int i = 0; i < ordering.vsize(); i++){
        std::cout << "\nVertex "            << ordering.at(i)->id << ":"
                  << "\n\t color: "         << ordering.at(i)->colorVal
                  << "\n\t orig degree: "   << ordering.at(i)->origDegree
                  << "\n\t deg delete: "    << ordering.at(i)->currDegree
                  << endl;
    }
```

```
    std::cout << "\n Total colors used: "      << num_colors
              << "\n Average Original Degree: " << calc_average_degree(ordering)
              << "\n Max Deleted Degree: "      << max_del_degree(ordering)
              << std::endl;
}
void shuffle(custom_vec<vertex*>& v){
    //initialize random device
    std::random_device rd;
    std::default_random_engine g(rd());

    std::uniform_int_distribution<int> dist(0, v.vsize() -1);
    for(int i = 0; i < 100; i++){

        v.swap(0, dist(g));
    }
}
int DFS_Util(LinkedList<vertex*>*& adjList, vertex* src, vertex* dest,
custom_vec<bool>& visited){

    //mark visited
    visited.at(src->id-1) = true;

    if(src->id == dest->id){
        return 0;
    }
    //traverse neighbors
    for (auto iter = src->P.getHead(); iter != nullptr; iter = iter->getNext()){
        //if vertex not visited, traverse
        if(!visited.at(iter->getData()->id - 1)){
            return 1 + DFS_Util(adjList, iter->getData(), dest, visited);
        }
    }

    return 0;

}
int DFS_Search(LinkedList<vertex*>*& adjList, vertex* src, vertex* dest) {

    //initialize visited vector
    custom_vec<bool> visited;
    for (int i = 0; i < adjList->getLength(); i++){
        visited.push_back(false);
    }

    int distance = DFS_Util(adjList, src, dest, visited);
```

```
    return distance;
}
bool check_connection(LinkedList<vertex*> list, int key){
    for(auto iter = list.getHead(); iter != nullptr; iter = iter->getNext()){
        if( iter->getData()->id == key){
            return true;
        }
    }

    return false;
}


int main(int argc, char** argv) {

    //read in input file
    ifstream file;
    file.open(argv[1], std::ifstream::in);

    //adjacency list
    LinkedList<vertex*>* adjList;

    //list of degrees
    custom_vec<LinkedList<vertex*>*>* degreeList;

    parseInput(file, adjList, degreeList);

    std::cout<< "~~~~~~~~~~ SMALLEST LAST VERTEX ORDERING ~~~~~~~~~~
\n\n" << std::endl;
    order result = smallestLastOrdering(adjList, degreeList);
    print_results(result.first, result.second);
    std::cout << "Max Clique of Graph: " << result.second << "\n" << std::endl;


    // UNCOMMENT TO USE. ONE AT A TIME

//    std::cout<< "~~~~~~~~~~ WELSH-POWELL ORDERING ~~~~~~~~~~ \n\n" <<
std::endl;
//    order result2 = welshPowellOrdering(adjList, degreeList);
//    print_results(result2.first, result2.second);
//
//    std::cout<< "~~~~~~~~~~ UNIFORM RANDOM ORDERING ~~~~~~~~~~ \n\n" <<
std::endl;
//    order result3 = uniformRandomOrdering(adjList);
//    print_results(result3.first, result3.second);
//
//    std::cout<< "~~~~~~~~~~ LARGEST LAST VERTEX ORDERING ~~~~~~~~~~
```

```
\n\n" << std::endl;
//    order result4 = largestLastOrdering(adjList, degreeList);
//    print_results(result4.first, result4.second);
//
//    std::cout<< "~~~~~~~~~~ LARGEST ECCENTRICITY ORDERING ~~~~~~~~~~
\n\n" << std::endl;
//    order result5 = largestEccentricityOrdering(adjList);
//    print_results(result5.first, result5.second);
//
//    std::cout<< "~~~~~~~~~~ DISTANCE FROM HIGHEST DEGREE VERTEX
ORDERING ~~~~~~~~~~ \n\n" << std::endl;
//    order result6 = distanceFromHighestDegreeVertexOrdering(adjList, degreeList);
//    print_results(result6.first, result6.second);


   return 0;
}


void parseInput(std::ifstream& file, LinkedList<vertex*>*& adjList,
custom_vec<LinkedList<vertex*>*>*& degreeList){

   string line;
   custom_vec<int> input;

   //open file
   if( file.is_open()){
      while(file >> line){

         //add values to input custom_vec
         input.push_back(stoi(line));

         //handle comments and move to next line
         getline(file, line);
      }
   }
   else{
      cout << "Failed to open file" << endl;
   }

   /*~~~~~~~~~~~ BUILD ADJACENCY LIST ~~~~~~~~~~~*/

   int numVertices = input[0];

   adjList = new LinkedList<vertex*>();

   //create vertices
```

```
    for(int i = 1; i <= numVertices; i++)
        adjList->insertAtTail(new vertex(i, 0, -1, 0));

    //add edges to vertices
    for(int i = 1; i <= numVertices; i++){
        //if last vertex loop until end of input custom_vec
        if (i == numVertices){
            for(int j = input[i]; j < input.vsize(); j++){
                //add edge input[j] to vertex i
                adjList->operator[](i-1)->getData()->addEdge(*(adjList) , input[j]);
            }
        }
        else{
            for(int j = input[i]; j < input[i+1]; j++){
                //add edge input[j] to vertex i
                adjList->operator[](i-1)->getData()->addEdge(*(adjList) , input[j]);
            }
        }
    }

    /*~~~~~~~~~~~ BUILD DEGREES LIST ~~~~~~~~~~~*/

    //initialize degrees custom_vec
    degreeList = new custom_vec<LinkedList<vertex*>*>();
    for(int i = 0; i <= numVertices; i++)
        degreeList->push_back(new LinkedList<vertex*>);



    while(adjList->currentNode() != nullptr){

        //insert vertex into corresponding degree index
        int degree = adjList->currentNode()->getData()->currDegree;
        adjList->currentNode()->getData()->origDegree = degree;
        degreeList->at(degree)->insertAtTail(adjList->currentNode()->getData());

        adjList->nextCurrent();
    }



}

int color_graph(custom_vec<vertex*>& ordering){

    //initialize color set
    custom_vec<int> color_set;
```

```
color_set.push_back(1);

//color first vertex
ordering.at(ordering.vsize() - 1)->colorVal = 1;

for(int i = ordering.vsize()-2; i >= 0; i--){

    bool colored = false;

    //try each color in color set
    for(int k = 1; k <= color_set.vsize(); k++){
        bool isNeighboringColor = false;

        //check if color is neighboring color
        while(ordering.at(i)->P.currentNode() != nullptr){
            vertex* neighbor = ordering.at(i)->P.currentNode()->getData();

            if(neighbor->colorVal != -1 && neighbor->colorVal == k){
                isNeighboringColor = true;
                ordering.at(i)->P.nextCurrent();
                break;
            }
            ordering.at(i)->P.nextCurrent();
        }

        //reset iter
        ordering.at(i)->P.currentToHead();

        //if not a neighboring color, color this vertex with k
        if(!isNeighboringColor){
            ordering.at(i)->colorVal = k;
            colored = true;
            break;
        }
    }

    //if all of the colors are colors of neighbors then add new color
    if(!colored){
        color_set.push_back(color_set.vsize() + 1);
        ordering.at(i)->colorVal = color_set.at(color_set.vsize() - 1);
    }


}

return color_set.vsize();
```

```
}

//~~~~~~~~~~ ORDERING ALGORITHMS ~~~~~~~~~~//
custom_vec<vertex*> smallestLastOrderingUtil(LinkedList<vertex*>* adjList,
custom_vec<LinkedList<vertex*>*>* degreeList){

    custom_vec<vertex*> ordering;
    int deletedCount = 0;

    //iterate until all vertices are deleted off degreeList
    while(deletedCount < adjList->getLength()){

        //loop through degreeList and find smallest populated degree list
        int min = min_index(degreeList);
        vertex* V = degreeList->at(min)->getHead()->getData();

        //set smallest vertex V to deleted
        V->deleted = -1;

        //delete vertex from degreeList
        degreeList->at(min)->remove(degreeList->at(min)->getHead()->getData());
        deletedCount++;

        //add vertex to ordering list
        ordering.push_back(V);

        //for every neighbor U of V -> insert U into degree i-1 list
        V->P.currentToHead();
        while(V->P.currentNode() != nullptr){

            //check if U has been deleted
            if(V->P.currentNode()->getData()->deleted == -1){
                V->P.nextCurrent();
                continue;
            }
            //remove U from degree i list
            vertex* U = degreeList->at(V->P.currentNode()->getData()->currDegree)-
>remove(V->P.currentNode()->getData())->getData();

            //insert U into degree i-1 list
            degreeList->at(--U->currDegree)->insertAtTail(U);


            V->P.nextCurrent();

        }
```

```
        //reset iter
        V->P.currentToHead();


    }

    //reverse ordering
    custom_vec<vertex*> rev_ordering;
    for(int i = ordering.vsize() - 1; i > -1; i--)
        rev_ordering.push_back(ordering.at(i));

    return rev_ordering;
}

order smallestLastOrdering(LinkedList<vertex*>* adjList,
custom_vec<LinkedList<vertex*>*>* degreeList){

    //call smallestLastOrdering
    custom_vec<vertex*> ordering = smallestLastOrderingUtil(adjList, degreeList);

    int num_colors = color_graph(ordering);

    return std::make_pair(ordering, num_colors);
}

order welshPowellOrdering(LinkedList<vertex*>* adjList,
custom_vec<LinkedList<vertex*>*>* degreeList){

    //initialize color
    int color = 1, colored = 0;

    //order of coloring
    custom_vec<vertex*> color_order;

    //add vertices to list in descending order
    custom_vec<vertex*> vertices;
    for(int i = degreeList->vsize() - 1; i > -1; i--){

        //check if any vertices with degree i
        if(degreeList->at(i)->getHead() == nullptr)
            continue;

        //add all vertices of degree i to vertices list
        while(degreeList->at(i)->currentNode() != nullptr){
            vertices.push_back(degreeList->at(i)->currentNode()->getData());
            degreeList->at(i)->nextCurrent();
```

```
        }
    }


    //color vertices
    for(int i = 0; i < vertices.vsize() && colored < vertices.vsize(); i++) {

        //color uncolored vertex
        if (vertices.at(i)->colorVal == -1) {
            vertices.at(i)->colorVal = color;
            color_order.push_back(vertices.at(i));
            colored++;
        }

        //color all vertices not connected to vertex i
        for(int k = i + 1; k < vertices.vsize(); k++){
            bool connected = is_connected(vertices.at(k), vertices.at(i)->id);

            if(!connected && vertices.at(k)->colorVal == -1){
                vertices.at(k)->colorVal = color;
                color_order.push_back(vertices.at(k));
                colored++;
            }
        }

        //change color
        color++;

    }


    return std::make_pair(color_order, color-1);

}

order uniformRandomOrdering(LinkedList<vertex*>* adjList) {



    //store vertices in vector for shuffling
    custom_vec<vertex *> vertices;
    for (auto iter = adjList->getHead(); iter != nullptr; iter = iter->getNext()){
        vertices.push_back(iter->getData());
    }

    //randomly shuffle vertices
    shuffle(vertices);
```

```
    int num_colors = color_graph(vertices);

    return std::make_pair(vertices, num_colors);

}

custom_vec<vertex*> largestLastOrderingUtil(LinkedList<vertex*>*& adjList,
custom_vec<LinkedList<vertex*>*>* degreeList){
    custom_vec<vertex*> ordering;
    int deletedCount = 0;

    //iterate until all vertices are deleted off degreeList
    while(deletedCount < adjList->getLength()){

        //loop through degreeList and find largest populated degree list
        int max = max_index(degreeList);
        vertex* V = degreeList->at(max)->getHead()->getData();

        //set largest vertex V to deleted
        V->deleted = -1;

        //delete vertex from degreeList
        degreeList->at(max)->remove(degreeList->at(max)->getHead()->getData());
        deletedCount++;

        //add vertex to ordering list
        ordering.push_back(V);

        //for every neighbor U of V -> insert U into degree i-1 list
        V->P.currentToHead();
        while(V->P.currentNode() != nullptr){

            //check if U has been deleted
            if(V->P.currentNode()->getData()->deleted == -1){
                V->P.nextCurrent();
                continue;
            }
            //remove U from degree i list
            vertex* U = degreeList->at(V->P.currentNode()->getData()->currDegree)-
>remove(V->P.currentNode()->getData())->getData();

            //insert U into degree i-1 list
            degreeList->at(--U->currDegree)->insertAtTail(U);


            V->P.nextCurrent();
```

```
        }

        //reset iter
        V->P.currentToHead();

    }

    //reverse ordering
    custom_vec<vertex*> rev_ordering;
    for(int i = ordering.vsize() - 1; i > -1; i--)
        rev_ordering.push_back(ordering.at(i));

    return rev_ordering;
}

order largestLastOrdering(LinkedList<vertex*>* adjList,
custom_vec<LinkedList<vertex*>*>* degreeList){
    //call largestLastOrdering
    custom_vec<vertex*> ordering = largestLastOrderingUtil(adjList, degreeList);

    int num_colors = color_graph(ordering);

    return std::make_pair(ordering, num_colors);
}

order largestEccentricityOrdering(LinkedList<vertex*>* adjList){

    //initialize random device
    std::random_device rd;
    std::default_random_engine g(rd());

    //distribution of vertex ids
    std::uniform_int_distribution<int> dist(0, adjList->getLength() - 1);

    //randomly select vertex calculate eccentricity of
    vertex* center = adjList->operator[](dist(g))->getData();

    //initialize eccentricity bucket list for bucket sort
    custom_vec<custom_vec<vertex*>> eccentricityList;
    for(int i = 0; i < adjList->getLength(); i++)
        eccentricityList.push_back(custom_vec<vertex*>());


    //calculate eccentricities from other vertices to randomly chosen vertex
    for(auto iter = adjList->getHead(); iter != nullptr; iter = iter->getNext()){
        if(iter->getData()->id == center->id)
```

```
            continue;
        int eccentricity = DFS_Search(adjList, iter->getData(), center);
        eccentricityList.at(eccentricity).push_back(iter->getData());
    }

    //create ordering based off max eccentricity
    custom_vec<vertex*> ordering;
    for(int i = eccentricityList.vsize()-1; i >= 0; i--){
        if(eccentricityList.at(i).vsize() > 0){
            for(int k = 0; k < eccentricityList.at(i).vsize(); k++)
                ordering.push_back(eccentricityList.at(i).at(k));
        }
    }

    //add center to the end
    ordering.push_back(center);


    int num_colors = color_graph(ordering);

    return std::make_pair(ordering, num_colors);

}

order distanceFromHighestDegreeVertexOrdering(LinkedList<vertex*>* adjList,
custom_vec<LinkedList<vertex*>*>* degreeList){

    //find vertex with largest degree
    vertex* largestDegreeVertex;
    for(int i = degreeList->vsize() - 1; i >=0; i--){
        if(degreeList->at(i)->getHead() != nullptr){
            largestDegreeVertex = degreeList->at(i)->getHead()->getData();
            break;
        }
    }

    //initialize bucket
    custom_vec<custom_vec<vertex*>> dist_bucket;
    for(int i = 0; i < degreeList->vsize() - 1; i++){
        dist_bucket.push_back(custom_vec<vertex*>());
    }

    //find distances from each vertex to largest degree vertex
    for(auto iter = adjList->getHead(); iter != nullptr; iter = iter->getNext()){
        if(iter->getData()->id == largestDegreeVertex->id)
            continue;
        int distance = DFS_Search(adjList, iter->getData(), largestDegreeVertex);
```

```
        dist_bucket.at(distance).push_back(iter->getData());
    }

    //create ordering sorted by largest distance from largest degree vertex
    custom_vec<vertex*> ordering;
    for(int i = dist_bucket.vsize() - 1; i >= 0; i--){
        if(dist_bucket.at(i).vsize() > 0){
            for(int k = 0; k < dist_bucket.at(i).vsize(); k++)
                ordering.push_back(dist_bucket.at(i).at(k));
        }
    }

    //add largest degree vertex to ordering
    ordering.push_back(largestDegreeVertex);

    int num_colors = color_graph(ordering);

    return std::make_pair(ordering, num_colors);
}
```

vertex.h

```
#ifndef FINAL_PROJECT_VERTEX_H
#define FINAL_PROJECT_VERTEX_H

#include "LinkedList.h"

class vertex {

public:
    int id;
    int currDegree;
    int origDegree;
    int colorVal;
    int deleted;

    //edge list
    LinkedList<vertex*> P;
```

```
    //list of vertices of same current degree
    LinkedList<vertex*>* sameDegree;

    //pointer for order deleted list
    LinkedList<vertex*>* orderDeleted;



    vertex();
    vertex(const vertex&);
    vertex(int, int, int, int);
    ~vertex();
    vertex& operator=(const vertex&);
    bool operator==(const vertex);

    void addEdge(LinkedList<vertex*>& adjList, int v);


};

#endif //FINAL_PROJECT_VERTEX_H
```

vertex.cpp

```
#include "vertex.h"

vertex::vertex(){

}

vertex::vertex(int id, int currDegree, int colorVal, int deleted)
    : id(id), currDegree(currDegree), colorVal(colorVal), deleted(deleted),
origDegree(currDegree) {}

vertex::vertex(const vertex& v){
```

```
}

vertex::~vertex() {}

vertex& vertex::operator=(const vertex& v) {
    this->id = v.id;
    this->currDegree = v.id;
    this->P = v.P;
    this->deleted = v.deleted;
    this->colorVal = v.colorVal;
    this->sameDegree = v.sameDegree;
    this->orderDeleted = v.orderDeleted;

    return *this;
}

bool vertex::operator==(const vertex v) {
    return (this->id == v.id);
}

void vertex::addEdge(LinkedList<vertex*>& adjList, int v) {

    //get ptr to vertex node on adjacency list
    vertex* edgePtr = adjList[v-1]->getData();

    //add edge to edge list P
    P.insertAtTail(edgePtr);

    //increment current degree
    currDegree++;

}
```

custom_vec.h

```
#ifndef FINAL_PROJECT_CUSTOM_VEC_H
```

```
#define FINAL_PROJECT_CUSTOM_VEC_H

#include <stdexcept>
#include <iostream>

using namespace std;

template<typename T>

class custom_vec
{
public:

    custom_vec(int = 10);
    custom_vec(const custom_vec<T>&);
    ~custom_vec();
    void push_back(T);
    void remove(); //remove last element of the vector
    custom_vec<T>& operator=(const custom_vec<T> &);
    T& operator[](const int);
    T& at(int);
    bool operator==(const custom_vec<T>&);
    int vsize() const;
    void setSize(int);
    //iterators
    void moveToStart();      // Reset position
    void moveToEnd();     // Set at end
    void prev();       // Back up
    void next(); // Next
    int currPos() const;   //return current position
    void moveToPos(int pos); // move current list position to pos
    int end();
    int begin();
    void clear();
    T& getValue() const; // get current element
    bool linearSearch(T);
    void swap(int, int);



private:
    T* data;
    int size; // amount of spaces used
    int capacity; //total number of spaces available
    void resize();
    int curr;
};
```

```
template<typename T> custom_vec<T> :: custom_vec(int x){
    size = 0;
    capacity = x;
    data = new T[x];
    curr = 0;
}

template<typename T> custom_vec<T> :: custom_vec(const custom_vec<T>& v){
    size = v.size;
    capacity = v.capacity;
    curr = 0;
    data = new T[v.vsize()];
    for(int i = 0; i < v.size;i++)
        data[i] = v.data[i];
}

template<typename T> custom_vec<T> ::  ~custom_vec(){
    if(data != nullptr)
        delete[] data;
}

template<typename T>
void custom_vec<T> :: push_back(T  p){
    if(size == capacity){
        capacity *= 2;
        this->resize();
    }
    data[size++] = p;

}

template<typename T>
void custom_vec<T> :: remove(){
//    T* temp = new T[capacity];
//    size = size - 1;
//    for(int i = 0; i < size;i++)
//            temp[i] = data[i];
//    delete[] data;
//    data = new T[capacity];
//    for(int j = 0;j < size;j++)
//        data[j] = temp[j];
//    delete[] temp;
    size--;

}
```

```
template<typename T>
void custom_vec<T> :: clear(){
    if(data != nullptr){
        delete[] data;
        size = 0;
        capacity = 10;
        data = new T[10];
    }
}




template<typename T>
void custom_vec<T> :: resize(){
    T* temp = new T[capacity * 2];
    for(int i = 0;i < size;i++){
        temp[i] = data[i];
    }
    delete[] data;
    data = temp;

}

template<typename T>
T& custom_vec<T> :: operator[](const int x) {
    if(x > vsize())
        throw std::out_of_range("Index out of range");
    return data[x];
}

template<typename T>
T& custom_vec<T> :: at(int x){
    if(x > vsize())
        throw std::out_of_range("Index out of range");
    return data[x];
}


template<typename T>
custom_vec<T>& custom_vec<T>:: operator= (const custom_vec<T>& v){

    size = v.size;
    capacity = v.capacity;
    for(int i = 0; i < v.size;i++){
        data[i] = v.data[i];
    }
```

```
    return *this;

}

template<typename T>
bool custom_vec<T> :: operator==(const custom_vec<T>& v){
    if(this->size != v.size)
        return false;
    for(int i = 0;i < size;i++){
        if(this-> data[i] != v[i])
            return false;
    }
    return true;
}

template<typename T>
void custom_vec<T> :: swap(int i, int j){

    if(i >= this->size || i < 0 || j >= this->size || j <0)
        throw std::out_of_range("Indexes out of range");

    auto temp = data[i];
    data[i] = data[j];
    data[j] = temp;

}

template<typename T>
int custom_vec<T> :: vsize() const{
    return this->size;
}

template<typename T>
void custom_vec<T> :: setSize(int x) { size = x; }

template<typename T>
void custom_vec<T> :: moveToStart() { curr = 0; }       // Reset position

template<typename T>
void custom_vec<T> :: moveToEnd() { curr = size; }     // Set at end

template<typename T>
void custom_vec<T> :: prev() { if (curr != 0) curr--; }      // Back up

template<typename T>
```

```
void custom_vec<T> :: next() { if (curr < size) curr++; } // Next

template<typename T>
int custom_vec<T> :: currPos() const { return curr;}

template<typename T>
int custom_vec<T> :: end(){
   moveToEnd();
   return curr;
}

template<typename T>
int custom_vec<T> :: begin(){
   moveToStart();
   return curr;
}

template<typename T>
// Set current list position to "pos"
void custom_vec<T> :: moveToPos(int pos) {
   if((pos<0)&&(pos>size))
      throw std::out_of_range("pos out of range");
   curr = pos;
}

template<typename T>
T& custom_vec<T> :: getValue() const { // Return current element
   if((curr<=0)&&(curr>size))
      throw std::out_of_range("No current element");
   return data[curr];
}


template<typename T>
bool custom_vec<T> :: linearSearch(T key){
   for(int i = 0; i < this->vsize(); i++){
      if(key == (*this)[i])
         return true;
   }
   return false;
}


#endif //FINAL_PROJECT_CUSTOM_VEC_H
```

linkedlist.h

```
#include <cstdlib>
#include <iostream>

#ifndef HOMEWORK_1_LINKEDLIST_H
#define HOMEWORK_1_LINKEDLIST_H


#include "node.h"
#include <iostream>

using namespace std;
template<typename T>

class LinkedList
{
public:
    LinkedList();
    LinkedList( LinkedList & ll);
    ~LinkedList();
    void insertAtHead(T x);
    void insertAtTail(T x);
    void insertBefore(T x ,Node<T>*);
    void clear();
    bool isEmpty();
    Node<T>* remove(T x);
    void printList();

    Node<T>* currentNode();
    void currentToHead();
    void nextCurrent();
    int getLength();
    Node<T>* getTail();
    Node<T>* getHead();
    LinkedList<T>& operator=( const LinkedList<T> &);
    Node<T>* operator[](int);
    int contains(T x);

private:
```

```
    Node<T>* head; //represents the first node in the list
    Node<T>* tail; // represents the last node
    Node<T>* current; // iterator used to represent the current node
    int index;
    int length;



};

//default constructor
template<typename T>
LinkedList<T> :: LinkedList(){
    head = nullptr;
    tail = nullptr;
    current = head;
    index = 0;
    length = 0;

}

//copy constructor
template<typename T>
LinkedList<T> :: LinkedList( LinkedList & ll){
    length = ll.length;
    Node<T>* temp = ll.head;
    while(temp != nullptr){
        insertAtTail(temp()->getData());
        temp = temp->next;

    }

}

//destructor
template<typename T>
LinkedList<T> :: ~LinkedList(){
    this->clear();

}

//inserts a node at the head of the linkedlist
template<typename T>
void LinkedList<T> :: insertAtHead(T x){
    Node<T>* nde = new Node<T>(x);
    if(isEmpty()){
        head = nde;
```

```
        tail =nde;
        nde->next = nullptr;
        nde->prev = nullptr;
    }
    else{
        Node<T>* temp = new Node<T>();
        temp = head;
        head->prev = nde;
        nde->prev = nullptr;
        head = nde;
        nde->next = temp;
        delete temp;
    }
    length++;

}

//inserts a node at the tail of the linkedlist
template<typename T>
void LinkedList<T> :: insertAtTail(T x){
    Node<T>* nde = new Node<T>(x);
    if(isEmpty()){
        head = nde;
        tail = nde;
        nde->next = nullptr;
        nde->prev = nullptr;
    }
    else{
        tail->next = nde;
        nde->next = nullptr;
        nde->prev = tail;
        tail = nde;
    }
    length++;
    currentToHead();
}


template<typename T>
void LinkedList<T> :: insertBefore(T x,Node<T>* nde){
    Node<T>* before = new Node<T>(x);
    Node<T>* temp = before->prev;
    Node<T>* it = head;
    while(it != nullptr){
        if(it== before)
            break;
```

```
        it = it->next;
    }
    nde->next = before;
    nde->prev = temp;
    temp->next = nde;
    before->prev = nde;
    delete temp;
    length++;
}

template<typename T>
void LinkedList<T> :: clear(){
    if(head != nullptr){
        while(tail != nullptr){
            Node<T>* temp = tail->prev;
            delete tail;
            tail = temp;
        }
    }
}

template<typename T>
Node<T>* LinkedList<T> :: remove(T x ){
    Node<T>* temp = head;
    while(temp != nullptr){
        if(temp->getData() == x)
            break;
        temp = temp->next;
    }
    if(temp != this->head && temp != this->tail){
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
    }
    else if(temp == this->head && temp != this->tail){
        head = head->next;
        temp->next->prev = nullptr;

    }
    else if(temp == this->tail && temp == this->head){
        temp->prev = temp->next = tail = head = nullptr;
    }
    else if(temp == this->tail){
        tail = tail->prev;
        temp->prev->next = nullptr;
    }
    else{
        head = tail = nullptr;
```

```
    }

    length--;
    this->currentToHead();
    return temp;
}

template<typename T>
bool LinkedList<T> :: isEmpty(){
    if(head == nullptr)
        return true;
    else
        return false;
}

template<typename T>
void LinkedList<T> :: printList(){
    Node<T>* temp = head;
    while(temp != nullptr){
        cout << "Data: " << current->getData() << endl;
        temp = temp->next;
    }
}

template<typename T>
int LinkedList<T> :: contains(T x){
    Node<T>* temp = head;
    index = 0;
    while(temp != nullptr){
        if(temp->getData() == x)
            return index;
        temp = temp->next;
        index++;
    }
    return -1;
}

template<typename T>
Node<T>* LinkedList<T> :: operator [](int x){
    Node<T>* temp = head;
    index = 0;
    while(temp != nullptr){
        if(index == x)
            return temp;
        temp = temp->next;
        index++;
    }
```

```
    return temp;
}




template<typename T>
LinkedList<T>& LinkedList<T> :: operator=(  const LinkedList<T> &list){
    if(this->length > 0){
        this->clear();
    }

    Node<T>* temp = list.head;
    while(temp != nullptr){
        insertAtTail(temp->getData());
        temp = temp->next;
    }


    return *this;
}


//iterators
template<typename T>
Node<T>*LinkedList<T>::currentNode(){ return current; }


template<typename T>
void LinkedList<T> :: currentToHead(){ current = head; index = 0;}


template<typename T>
void LinkedList<T> :: nextCurrent(){ current = current->next; index++; }


template<typename T>
int LinkedList<T> :: getLength() { return length;}

template<typename T>
Node<T>* LinkedList<T> :: getTail(){ return tail;}

template<typename T>
Node<T>* LinkedList<T> :: getHead(){ return head;}

#endif //HOMEWORK_1_LINKEDLIST_H
```

node.h

```
#ifndef FINAL_PROJECT_NODE_H
#define FINAL_PROJECT_NODE_H

template<typename T>

class Node
{
public:
    Node(T);
    Node(const Node<T>& );
    ~Node();
    T& getData(); //returns the data
    void setData(T x  ); //sets the data
    Node<T>* getNext();
    Node<T>* operator=(Node<T>*);


private:

    T data;
    Node<T> * next;
    Node<T> * prev;
    template<typename U>
    friend class LinkedList;


};


template<typename T>
Node<T>* Node<T> :: operator =(Node<T>* n){
    data = n->data;
    next = n->next;
    prev = n->prev;
}


template<typename T>
Node<T> :: Node(T x){
```

```
    data = x;
    next = nullptr;
    prev = nullptr;

}

template<typename T>
Node<T> :: Node(const Node<T> & n){
    data = n.data;
    next = n.next;
    prev = n.prev;

}

template<typename T>
Node<T> :: ~Node(){}

template<typename T>
void Node<T> :: setData(T x) {data = x;}

template<typename T>
T& Node<T> :: getData(){return data;}

template<typename T>
Node<T>* Node<T>:: getNext() {return next; }

#endif //FINAL_PROJECT_NODE_H
```