CSCE 221 Cover Page
Programming Assignment #5
Due November 13th at midnight to CSNet

First Name Raymond Last Name Zhu UIN 923008555

User Name rawrbyte E-mail address rawrbyte@tamu.edu

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero. According to the University Regulations, Section 42, scholastic dishonesty are including: acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. Disciplinary actions range from grade penalties to expulsion read more: Aggie Honor System Office

Type of sources			
People	Peer Teachers		
Web pages (provide URL)	Stackoverflow		
Printed material	textbook		
Other Sources			

I certify that I have listed all the sources that I used to develop the solutions/codes to the submitted work.

"On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work."

Your Name Raymond Zhu Date November 13th, 2015

1. Program Description

   (a) Taking implementations from a doubly linked list, this program takes characters and creates sets/union sets and combines them together. The program utilizes ADT structures to create an efficient program that allows the user to manage sets through a disjointset class.

2. Purpose of the Assignment

   (a) The purpose of this programming assignment is to learn how to manage sets through implementations of a doubly linked list.

3. Data Structures Description

   (a) A modified doubly linked list was implemented in this assignment. During the implementation of this abstract data type, the most important aspect that stood out for this ADT to me is the use of two reference pointers representative and trailer and the listsize. New functions within the DListNode class can get and set these nodes and in addition, track the listsize of the given set.

4. Alogorithm Description

   - MakeSet $O(1)$: given a key and a element, the function creates a set with a single element
   - Union $O(1)$: given two nodes, the function executes a union operation on the two nodes, resulting in a single set with all the nodes from both sets. Path compression and union by rank were implemented.
   - FindSet $O(1)$: given a node or its key, the function returns the representative of the set that contains this node

   FindSet has a best and worst case of O(1). Union has a best case of O(1) and a worst case of O(log(n))

5. Program Organization and Description of Classes

```
template <typename T>
class DisjointSet {
private:
        vector<DListNode<T>*> nodeLocator;
public:
        ~DisjointSet();
        DisjointSet(int n);
        vector<DListNode<T>*> getNodeLocator() const;
        DListNode<T>* MakeSet(int key, T node);
        DListNode<T>* Union(DListNode<T>& nodeI, DListNode<T>&
             nodeJ);
        DListNode<T>* FindSet(DListNode<T> node);
        DListNode<T>* FindSet(int nodeKey);
};
```

```
template <typename T>
DisjointSet<T>::~DisjointSet(){
        for(int i = 0; i < nodeLocator.size(); ++i){
                if(nodeLocator[i] != NULL){
                        while(nodeLocator[i]->getNext() !=
                            NULL){
                                nodeLocator[i]->delete_after()
                                    ;
                        }
                        delete nodeLocator[i];
                }
        }
}

template <typename T>
DisjointSet<T>::DisjointSet(int n){}

template <typename T>
vector<DListNode<T>*> DisjointSet<T>::getNodeLocator() const{
        return nodeLocator;
}

template <typename T>
DListNode<T>* DisjointSet<T>::MakeSet(int key, T node){
        DListNode<T> *temp = new DListNode<T>(key, node);
        temp->setRepresentative(temp);
        temp->setTrailer(temp);
        nodeLocator.push_back(temp->getRepresentative());
        return temp;
}

template <typename T>
DListNode<T>* DisjointSet<T>::Union(DListNode<T>& nodeI,
    DListNode<T>& nodeJ){
        DListNode<T>* temp;
        DListNode<T>* smaller;
        DListNode<T>* dummy;
        if(nodeI.getListSize() >= nodeJ.getListSize()){
                temp = &nodeI;
                smaller = &nodeJ;
                dummy = &nodeJ;
        }else{
                temp = &nodeJ;
                smaller = &nodeI;
                dummy = &nodeI;
        }
        if(temp->getRepresentative() != smaller->
            getRepresentative()){
                smaller->setPrevious(temp->getRepresentative()
                    ->getTrailer());
                temp->getRepresentative()->getTrailer()->
                    setNext(smaller);
                temp->getRepresentative()->setTrailer(smaller
                    ->getTrailer());
                while(smaller->getTrailer() != temp->
                    getRepresentative()->getTrailer()){
                        smaller->setRepresentative(temp->
```

```
                        getRepresentative());
                    temp->getRepresentative()->setListSize
                        (temp->getRepresentative()->
                        getListSize()+smaller->
                        getRepresentative()->getListSize()
                        );
                    smaller->setListSize(temp->
                        getRepresentative()->getListSize()
                        );
                    smaller = smaller->getNext();
                }
                temp->getRepresentative()->setListSize(temp->
                    getRepresentative()->getListSize()+smaller
                    ->getRepresentative()->getListSize());
                smaller->setListSize(temp->getRepresentative()
                    ->getListSize());
                dummy->setRepresentative(temp->
                    getRepresentative());
        }else{
                cout << "The two sets are the same!" << endl;
        }
        return temp;
}

template <typename T>
DListNode<T>* DisjointSet<T>::FindSet(DListNode<T> node){
        return nodeLocator[node.getKey()-1]->getRepresentative
            ();
}

template <typename T>
DListNode<T>* DisjointSet<T>::FindSet(int nodeKey){
        return nodeLocator[nodeKey-1]->getRepresentative();
}

template <typename T>
ostream& operator<<(ostream& out, const DisjointSet<T>& ds){
        for(int i = 0; i < ds.getNodeLocator().size(); ++i){
                if(ds.getNodeLocator()[i]->getRepresentative()
                    == ds.getNodeLocator()[i]){
                        out << "{ " ;
                        DListNode<T> *temp = ds.getNodeLocator
                            ()[i];
                        while(temp != ds.getNodeLocator()[i]->
                            getTrailer()){
                                out << temp->getKey() << ":"
                                    << temp->getElem() << " ";
                                temp = temp->getNext();
                        }
                        out << temp->getKey() << ":" << temp->
                            getElem() << " ";
                        temp = temp->getNext();
                        out << "}" << endl;
                }
        }
        return out;
}
```

```
template <typename T>
class DListNode {
        private:
                int key;
                int listSize;
                T obj;
                DListNode *prev, *next, *representative;
                DListNode *trailer; //just the representative
                    node has this pointer assigned
        public:
                DListNode(int k, T e = T(), DListNode *p =
                    NULL, DListNode *n = NULL)
                : key(k), obj(e), prev(p), next(n) { listSize
                    = 1; }
                T getElem() const { return obj; }
                T& getElemt() { return obj; }
                DListNode<T> * getNext() const { return next;
                    }
                DListNode<T> * getPrev() const { return prev;
                    }
                void setNext(DListNode* n) { this->next = n; }
                void setPrevious(DListNode* p) { this->prev =
                    p; }
                DListNode<T>* insert_before(T d); // insert
                    the int before this node
                // return a pointer to the inserted node
                DListNode<T>* insert_after(T d); // insert the
                     int after this node
                // return a pointer to the inserted node
                void delete_before(); // delete the node
                    before this node
                void delete_after(); // delete the node after
                    this node
                int getKey() { return key; }
                DListNode<T>* getRepresentative() const;
                DListNode<T>* getTrailer() const;
                void setRepresentative(DListNode* rep);
                void setTrailer(DListNode* trail);
                int getListSize();
                void setListSize(int lSize);
};

template <typename T>
DListNode<T>* DListNode<T>::insert_before(T d) {
        DListNode<T>* temp = new DListNode(d);  //temp pointer
             // 1 OPERATION

        if(prev!= NULL){   //if previous is not null // 2
            OPERATION
                temp -> next = this;     //initialize next // 2
                     OPERATION
                temp -> prev = prev;    //initialize previous
                    // 2 OPERATION
                prev -> next = temp;    // insert in the list
                    // 2 OPERATION
                prev = temp;           // 1 OPERATION
```

```
        }
        else{       //if null than do a simple insert
                temp -> next = this;  // 2 OPERATION
                prev = temp;    // 1 OPERATION
        }

        return temp; // 1 OPERATION
}

template <typename T>
DListNode<T>* DListNode<T>::insert_after(T d) {
        DListNode<T>* temp = new DListNode(d);   // 2
            OPERATION
        if(next != NULL){   //if next is not null do this // 2
             OPERATION
                temp -> next = next;      //initialize next
                    and previous // 2 OPERATION
                temp -> prev = this;      // 2 OPERATION
                next -> prev = temp;   //make the connection
                    to insert // 2 OPERATION
                next = temp; // 1 OPERATION
        }
        else{ //if null than do a simple insert
                temp -> prev = this;  // 2 OPERATION
                next = temp; // 1 OPERATION
        }

        return temp;  // 1 OPERATION
}

template <typename T>
void DListNode<T>::delete_before() {
        if(prev!= NULL){   //check if something is there  // 2
             OPERATION
                DListNode<T>* temp = prev;          // 1
                    OPERATION
                temp -> prev-> next = temp-> next;     //change
                     the interconnections  // 4 OPERATION
                temp -> next -> prev = temp -> prev; // 4
                    OPERATION
                delete temp;         //delete the element // 1
                    OPERATION
        }
        else{
                cout << ">Error: Nothing is there :(" << endl;
        }
}

template <typename T>
void DListNode<T>::delete_after() {
        if(next != NULL){ //check if something is there
                DListNode<T>* temp = next;  // 1 OPERATION
                next -> prev = this; //change the
                    interconnections // 2 OPERATION
                next = next -> next; // 2 OPERATION
                delete temp;  //delete the element // 1
                    OPERATION
```

```
        }
        else{
                cout << ">Error: Nothing is there :(" << endl;
        }
}

template <typename T>
DListNode<T>* DListNode<T>::getRepresentative() const{
        return representative;
}

template <typename T>
DListNode<T>* DListNode<T>::getTrailer() const{
        return trailer;
}

template <typename T>
void DListNode<T>::setRepresentative(DListNode* rep){
        representative = rep;
}

template <typename T>
void DListNode<T>::setTrailer(DListNode* trail){
        trailer = trail;
}

template <typename T>
int DListNode<T>::getListSize(){
        return listSize;
}

template <typename T>
void DListNode<T>::setListSize(int lSize){
        listSize = lSize;
}
```

6. Instructions to Compile and Run your Program

   - type "make" to makeall
   - and then
   - ./main to execute the program

7. Input and Output Specifications

   - None

8. Logical Exceptions

   (a) No logical error has been found in testing from the program itself

9. C++ object orientated or generic programming features

   (a) Generic programming with templates

10. Tests

```
[rawrbyte]@sun ~/CSCE221/lab5> (18:28:23 11/12/15)
:: ./main
Sets:
{ 1:a }
{ 2:b }
{ 3:c }
{ 4:d }
{ 5:e }

a.Union(c, d)
Sets:
{ 1:a }
{ 2:b }
{ 3:c 4:d }
{ 5:e }

a.Union(d, e)
Sets:
{ 1:a }
{ 2:b }
{ 3:c 4:d 5:e }

a.find(a): a
a.find(d): c
a.Union(a, b)
Sets:
{ 1:a 2:b }
{ 3:c 4:d 5:e }

a.Union(a, e)
Sets:
{ 3:c 4:d 5:e 1:a 2:b }

a.find(a): c
a.find(e): c
listSize(a): 5
listSize(e): 5
```