```
//=============================================================================
// linear.cpp
// This file contains excerpts from projects involving linear data structures
// December 2017
//=============================================================================



//=============================================================================
// LIST AS ARRAY
//=============================================================================
template <class T>
class List
{
private:
        T               *array;
        int             DEFAULT_LIST_SIZE = 10;
        int             capacity;
        int     size;

public:

                                List                (void);             //default
                List                (const List<T> &c); //copy constructor
                ~List               (void);             //destructor
        bool            isEmpty                 (void);
        int             length              (void) const;
        T&                      operator[]              (int i);        //index operator
        string          toString            (void)const ;
        void            append              (T c);
        void            insert              (T, int);
        void            remove              (int);
        List<T>         operator+               (const List<T> c) const ;
        List<T>         operator=               (List<T> c) const ;
        void        clear                   (void);
  friend ostream& operator<<    (ostream &os, List<T> c)
  {
    for (int i = 0; i < c.size; i++)
      os << c[i] << " ";

    return os;
  }
};
class IndexError { };

//=============================================================================
// default constructor
//=============================================================================
template <class T>
                List<T>::List   (void)
{
        array = new T[DEFAULT_LIST_SIZE];
        capacity = DEFAULT_LIST_SIZE;
        size = 0;
}

//=============================================================================
// insert
// inserts indicated item at indicated position
//=============================================================================
template <class T>
void    List<T>::insert         (T item, int pos)
{
```

```
        if (pos > size or pos < 0)
        {
                //cout << "Error. Invalid index.\n";
                //exit(1);
    throw IndexError();
        }
        else if (size + 1 > capacity)
        {
                T *oldArray = array;
                capacity *= 2;
                array = new T[capacity];

                for (int i = 0; i < pos; i++)
                        array[i] = oldArray[i];

                array[pos] = item;

                for (int i = pos; i < size; i++)
                        array[i+1] = oldArray[i];

                size += 1;
                delete[] oldArray;
        }
        else
        {

                T *oldArray = array;
                array = new T[capacity];

                for (int i = 0; i < pos; i++)
                        array[i] = oldArray[i];

                array[pos] = item;

                for (int i = pos; i < size; i++)
                        array[i+1] = oldArray[i];

                size += 1;
                delete[] oldArray;
        }

}

//==========================================================================
// remove
// removes item from list at indicated position
//==========================================================================
template <class T>
void    List<T>::remove         (int pos)
{
        if (pos >= size or pos < 0)
        {
                //cout << "Error. Invalid index.\n";
                //exit(1);
    throw IndexError();
        }
        T *oldArray = array;
        array = new T[capacity];

        for (int i = 0; i < pos; i ++)
                array[i] = oldArray[i];

        for (int i = pos; i < size; i++)
```

```
                array[i] = oldArray[i+1];

        size -= 1;
        delete[] oldArray;
}

//========================================================================
// operator []
// returns item in list at indicated index
//========================================================================
template <class T>
T&              List<T>::operator[]     (int i)
{
        if (i >= size or i < 0)
        {
                //cout << "Error. Invalid index\n";
                //exit(1);
                throw IndexError();
        }
        return array[i];
}

//========================================================================
// clear
// removes items from list
//========================================================================
template <class T>
void    List<T>::clear                  (void)
{
        T *oldArray = array;
        array = new T[capacity];
        size = 0;
        delete[] oldArray;
}

//========================================================================
// LIST AS LINKED LIST
//========================================================================

template <class T>
class List
{
private:
        struct Node
        {
                T     data;
                Node *  next;
        };

        Node *head;

public:
                                List            (void);                                 //defa
ult constructor
                        List            (const List<T> &src);   //copy constructor
                        ~List           (void);                                 //destructor
  T&                    operator[]              (int x);
  List<T>               operator+       (const List<T> &l);
  List<T>               operator=       (const List<T> &src);
  bool                  isEmpty         (void);
  int                   length          (void);
  string                toString                (void);
  void                  append          (T x);
```

```
  void                  insert          (T i, int p);
  void                  remove          (int p);
  void                  clear           (void);

  friend ostream & operator<< ( ostream &os, List<T> &l)
  {
        Node *ptr = l.head;
        while (ptr != NULL)
        {
        os << ptr->data << " ";
        ptr = ptr->next;
        }
        return os;
  }

};
class IndexError { };
//========================================================================
//default constructor
//creates an empty list
//========================================================================
template <class T>
        List<T>::List (void)
{
        head = NULL;
}

//========================================================================
// insert
// inserts an item at the given position
//========================================================================

template <class T>
void    List<T>::insert (T i, int p)
{
        if (p > length() || p < 0)
        {
                //cout << "Error: invalid index.\n";
                //exit(1);                                              //prints error
 message and exits program
    throw IndexError();
        }
        Node *ptr = head;
        int count = 0;
        if (p == 0)
        {
                Node *qtr = new Node;
                qtr->data = i;
                qtr->next = ptr;
                head = qtr;
        }
        else
        {
                while (count < (p-1))                           //need to stop one before desi
red index to
                {                                                                       //make
 new node in the right place
                        ptr = ptr->next;
                        count++;                                                        //goes through
 list until desired index
                }
                Node *qtr = new Node;                           //sets new node equal to given
 item and
```

```
                qtr->data = i;                                          //pointing to next nod
e
                qtr->next = ptr->next;
                ptr->next = qtr;                                        //creates link to new
node
        }
}


//========================================================================
// remove
// removes item at given index
//========================================================================

template <class T>
void    List<T>::remove (int p)
{
        if (p >= length() || p < 0)
        {
                //cout << "Error: invalid index.\n";
                //exit(1);                                      //prints error message and qui
ts program
    throw IndexError();
        }
        Node *ptr = head;
        Node *qtr = head;
        Node *rm;
        int count = 0;
        if (p == 0)
        {
                rm = ptr;
                delete rm;
                head = ptr->next;
        }
        else
        {
                while (count < (p-1))
                {
                        ptr = ptr->next;                        //goes through list until inde
x right before p
                        qtr = qtr->next;
                        count++;
                }
                ptr = ptr->next;
                rm = ptr;
                ptr = ptr->next;
                qtr->next = ptr;
                delete rm;

        }
}

//========================================================================
//operator[]
//returns item in list at given index
//========================================================================

template <class T>
T&      List<T>::operator[]     (int x)
{
        if (x >= length() || x < 0)
        {
                // << "Error: Invalid index.\n";
```

```
                //exit(1);                                          //prints error if invalid inde
x
                throw IndexError();
        }
        int count = 0;
        Node *ptr = head;
        while (count < x)
        {
                ptr = ptr->next;                        //goes through list until it reaches g
iven index
                count++;
        }
        return ptr->data;                               //returns value at index
}

//=========================================================================
// clear
// deletes the links from the list
//=========================================================================
template <class T>
void List<T>::clear (void)
{
        Node *ptr, *qtr;
        ptr = head;
        qtr = head;
        while (ptr != NULL)
        {
                qtr = ptr->next;                        //sets node equal to item after ptr
        delete ptr;                                     //deletes ptr
        ptr = qtr;                                      //resets ptr to item originally after
        }
        delete ptr;                                             //deletes head
        head = NULL;
}

//=========================================================================
// STACK
//=========================================================================

template <class T>
class Stack
{
private:
        List<T> list;



public:
                        Stack                   (void);
                        ~Stack                  (void);
                        Stack                   (const Stack<T> &);
        void            push                    (T);
        T                       pop                                     (void);
        T                       peek                    (void);
        int                     length                  (void) const;
        bool            isEmpty                 (void) const;
        void            clear                   (void);


        Stack<T>        operator=               (const Stack<T> &);

        friend ostream & operator<< (ostream &os, const Stack<T> &s)
        {
```

```
                os << s.list;
                return os;
        }

};
//=========================================================================
// default constructor
//=========================================================================
template <class T>
                Stack<T>::Stack         (void)
{
        //secretly creates private variables, no code needed
}

//=========================================================================
// insert
//=========================================================================
template <class T>
void            Stack<T>::push          (T item)
{
        list.insert(item, 0);
}
//=========================================================================
// remove
//=========================================================================
template <class T>
T               Stack<T>::pop           (void)
{
        T temp = list[0];
        list.remove(0);
        return temp;
}
//=========================================================================
// clear
//=========================================================================
template <class T>
void    Stack<T>::clear         (void)
{
        list.clear();
}
```