<u>Linear Data Structures</u>

Proficiency: Mastery with Distinction

In the field of computer science knowledge of linear data structures is vital. Some of the most commonly used data structures are linear data structures. Lists, stacks, queues are all fundamental structures that are used and modified to be used in other data structures. The knowledge of how these data structures work, how they're implemented and proper usage is vital. Last year in CS 173 we thoroughly studied stacks, lists and queues. We implemented a stack using linked lists, then modified the class to use doubly linked lists and finally we implemented the stack using dynamically allocated arrays. All three implementations of the stack are attached. A stack is a last in first out structure, meaning the first item is added to the stack and the next items are added on top of the previous item (stacked on one another).It is easy to visualize a stack as a vertical container with a hole at the top. A Stack has four operations: pop(), peek(), push(x), and isEmpty(). Push(x) adds item x to the top of the stack. Pop() removes and returns the item at the top of the stack. Peek() does the same as pop() but does not remove the item. Lastly, isEmpty() returns a boolean value, true if the stack has no items in it and false else wise.

A queue, unlike a stack, is a first in first out data structure, meaning the first item added to the queue is first in line to leave the queue. A good visualization of a queue is a horizontal pipe; the first thing you push in the left side of the pipe will be the first thing that comes out of the right side of the pipe. A queue can be implemented almost identically to a stack, with a few changes here and there. Popular elementary ways of implementing a stack are arrays, linked lists and doubly linked lists. A queue is like a line in the grocery store (hence why British call lines 'queues'), the early you get your order in, the early you will be served. A queue has three operations: isEmpty(), enqueue(x), dequeue(). isEmpty() for a queue is the same as is empty for a stack. Enqueue(x) adds the item x to the end of the queue. Dequeue() removes the front item from the queue. A more advanced type of queue is the Minimum Priority Queue which we implemented this year using a heap. A Minimum Priority Queue is a type of queue where the the priority of one item may be more important than another. The important items are kept at the front of the queue, sort of like a celebrity cutting the line to get into a restaurant.

Last is the list ADT. A list class in computer is essentially just a computerized form of the way we use lists everyday. Unlike a queue and stack, when an item is added to a list has noImagine making a list of guests to invite to a party. You start with person 1, then add person 2 and so on. A list ADT is just a linear sequence of items, where an item can be accessed or edited by knowing its index in the list. A basic list has five operations: add(x, i), remove(x), append(x), isEmpty() and size(). Add(x, i) varies from append(x) in the fact that add(x, i) puts item x at the specified index i, while append adds item x to the end of the list. Remove(x) removes the first instance of x in the list. isEmpty() acts the same as queue and stack. Size() returns the amount of items in the list.

As you can see linear data structures are all somewhat similar, but each has unique characteristics that may make one ADT better suited for a problem than another. All of these structures can be used to hold integers, strings, chars, etc. After implementing all of these ADTs I feel like I have more than masted Linear Data Structures.

```
/*
Taylor Heilman
list class using doubly linked lists
list.cc
*/

//#include "list.h"
#include <stdlib.h>
#include <iostream>
using namespace std;

//============================================
//      Default Constructor
//============================================
template<class T>
List<T>::List ( void )
{
        head = NULL;
        tail = NULL;
        size = 0;

}

//============================================
//      Destructor
//============================================
template<class T>
List<T>::~List ( void )
{
        dealloc();
}

//============================================
//      Copy Constructor
//============================================
template<class T>
List<T>::List ( const List<T>& source )
{
        copy(source);

}

//============================================
//      Assignment Operator
//============================================
template<class T>
List<T> & List<T>:: operator=   (const List<T>& source)
{
        if(this != &source)
        {
                dealloc();
                copy(source);
        }

        return *this;
}



 //===========================================
 //     Append
 //===========================================
template<class T>
void List<T>::append  (const T& x)
  {
          Node<T> * temp;
          temp = new Node<T>;

         if (head == NULL)               // appending to empty list
         {
```

```
                head = temp;
                tail = temp;
                temp -> item = x;
                temp -> next = NULL;
                temp -> prev = NULL;
                size++;
        }

        else                            // apending to end of list
        {
                tail->next = temp;
                temp->item = x;
                temp->prev = tail;
                temp->next = NULL;
                tail = temp;
                size++;

        }

  }




//============================================
//      Insert
//============================================
template<class T>
void List<T>::insert   (int index, const T & x)
{
        Node<T> * temp;
        temp = new Node<T>;

        if (index < 0 or index > size)          //indes out of bounds
        {
                delete temp;
                throw IndexError();

        }

        else if (size == 0)                     // empty list
        {
                temp->item = x;
                temp->prev = NULL;
                temp->next = NULL;
                head = temp;
                tail = temp;
                size++;

        }

        else if (index == 0 )                   // inserting to first spot
        {
                 temp->item = x;
                 temp->prev = NULL;
                 temp->next = head;
                 head->prev = temp;
                 head = temp;
                 size++;

        }
        else if (index == size)                 // inserting to last spot
        {
                append(x);
        }

        else                                    // inserting in middle
        {
                temp = head;
```

```
                for(int i=0; i < index; i++)
                {
                        temp = temp-> next;
                }

                Node<T> * temp2;
                temp2 = new Node<T>;
                temp2->next = temp;
                temp2->prev = temp->prev;
                temp2->item = x;
                temp->prev = temp2;
                temp2->prev->next = temp2;
                size++;

        }

}



//==========================================
//      String
//==========================================
template<class T>
string  List<T>::str()
 {

         string str = "";
         Node<T>*temp = head;
         char Reason [50];
         str += "[";
         while (temp != NULL)
         {
                 if (temp -> next != NULL)
                 {
                         sprintf(Reason,"%d", temp -> item);
                         str+= Reason;
                         str += ", ";
                 }

                 else
                         if(temp -> next == NULL)
                         {
                                 sprintf(Reason, "%d", temp -> item);
                                 str+= Reason;
                         }
                         temp = temp -> next;
         }
         str += "]";
         return str;
 }

//==========================================
//      Index
//==========================================
template<class T>
int List<T>::index  ( const T & x )
 {

         Node<T> * temp = head;
         int place = 0;

         while (temp != NULL and temp->item != x)
         {
                 temp = temp->next;
                 place++;
         }

         if (temp == NULL)      // if list is empty or item isn't in list
                 return -1;
```

```
         else
                 return place;  // return index of the item

}


//============================================
//      Pop
//============================================
template<class T>
T   List<T>::pop (int index)
{
        T x;
        int y = size -1;

        if ( head == NULL)                              //empty list
        {
                throw IndexError();

        }
        else if (index == y)                    // popping last item
        {
                Node<T> * temp = tail;
                x = tail->item;
                tail = tail->prev;
                tail->next = NULL;
                delete temp;
                size--;
                return x;

        }

        else if( index == 0)                    // popping first item
        {
                Node<T> * temp = head;
                x = head -> item;
                head = head-> next;
                head->prev = NULL;
                delete temp;
                size--;
                return x;
        }


        else if(index > 0 and index < y)        // poppoing from middle of list
        {
                Node<T> * temp = head;
                for(int i=0; i<index; i++)
                {
                        temp = temp->next;
                }
                x = temp->item;
                (temp->prev)->next = temp->next;
                (temp->next)->prev = temp->prev;
                delete temp;
                size--;
                return x;


        }

        else                                    // no index given
        {
                Node<T> * temp = tail;
                x = tail->item;
                tail = tail  ->prev;
                tail->next = NULL;
                delete temp;
                size--;
                return x;
```

```
        }

}




//==========================================
//      Indexing Operator
//==========================================
template<class T>
T &  List<T>::operator[]  (int index)
{
        if (index < 0 or index > size-1)        //index out of bounds
                throw IndexError();
        else
        {
                Node<T> * temp = _find(index);
                return temp->item;

        }
}


//==========================================
//      resetForward
//==========================================
template<class T>
void   List<T>::resetForward(void)
{
        currentFwd = head;

}

//==========================================
//      next
//==========================================
template<class T>
T    List<T>::next()
{
        Node<T> * temp = currentFwd;
        if (temp == NULL)
                throw StopIteration();
        else
        {
                T z = currentFwd->item;
                currentFwd = currentFwd->next;
                return z;
        }
}

//==========================================
//      resetReverse
//==========================================
template<class T>
void  List<T>::resetReverse(void)
{
        currentRev = tail;
}

//==========================================
//      prev
//==========================================
template<class T>
T    List<T>::prev  (void)
{
        Node<T> * temp = currentRev;
        if (temp == NULL)
                throw StopIteration();
```

```
        else
        {
                T z = currentRev->item;
                currentRev = currentRev->prev;
                return z;
        }
}



//============================================
//      copy
//============================================
template<class T>
void  List<T>::copy  (const List<T>& source)
{
        Node<T> *snode, *node;                  // deep copy

        snode = source.head;
        if (snode)
        {
                node = head = new Node<T>(snode->item);
                snode = snode->next;
        }
        else
                head = NULL;
        while(snode)
        {
                node->next = new Node<T>(snode->item);
                node = node->next;
                snode = snode->next;
        }
        size = source.size;
}




//============================================
//      dealloc
//============================================
template<class T>
void   List<T>::dealloc     ()
{
        Node<T> * temp = head;

        while( temp != NULL )
        {
                head = head->next;
                delete temp;
                temp = head;
        }

        delete temp;
}



//============================================
//      _find
//============================================
template<class T>
Node<T>*  List<T>:: _find  (int index)
{
        Node<T> * temp = head;
        for(int i=0; i<index; i++)
                temp = temp->next;

        return temp;
}
```

```
//==========================================================================
// Matt Kretchmar
// April 1, 2015
// list.h
//
// This file contains the class definition for a List ADT class.
// ** Do not modify the Node<T> or List classes. **
// ** You will modify the StopIteration class at the bottom **
//
//==========================================================================
// List ADT
//
// The List class implements a sequence of stored items all of the same datatype.
// There are methods to add and remove items from the List, to query the List for
// an item, to index into the list at a specific location, and to iterate through
// the list.
//
// Default Constructor: creates an empty List (no items)
// Copy Constructor:    creates a new List that is an exact copy of an existing List.
// Destructor:          cleans up the memory for an existing List to be deleted.
// Assignment Operator: makes a copy of an existing List for the assigned List.
//
// length():            returns the number of items in the List.
// append(ItemType &x): adds item x to the end of the existing List.  Note that
//                      duplicate items are permitted.
// insert(i,x):         inserts item x at location i in the List.  The existing
//                      items are moved towards the end of the List to make room
//                      for the new item.  Valid values for i are 0 to length().
//                      If length() is the index, this will add the new item to the
//                      end of the list (such as in append).
// pop(i):              removes and returns item at index i from the list.  Valid
//                      values for i are 0 to length()-1.  The argument is optional
//                      will default to removing the last item in the list if i is not gi
ven.
// operator[i]:         access (by reference) the item at index i.  Valid values
//                      for i are 0 to length()-1.  The access by reference allows
//                      the user to change the value at this index.
// index(x):            returns the index of the first occurrence of item x in
//                      the List, returns -1 if x is not in the list.
// resetForward():      resets the forward iterator to the front of the list.
// resetReverse():      resets the backward iterator to the end of the list.
// next()               returns the value of the next item in the list using the
//                      forward iterator location.  The forward iterator is then
//                      moved to the next item.
// prev()               returns the value of the next item in the list using the
//                      backward iterator location.  The backward iterator is then
//                      moved to the next (previous) item.
// str()                Converts the List into a string, follows Python format.
//                      Example: "[1, 2, 3]" or "[]"
// cout <<              Overloads the cout << operator for printing. Follows the
//                      same format as in str().
//==========================================================================



#include <iostream>
#include <stdio.h>
#include <stdlib.h>
using namespace std;

#ifndef LIST_H
#define LIST_H

template <class T>        // where you can change the type of item stored in the list
struct Node
{
    T               item;                                      // data item stored in thi
s link
    Node *      next;                                      // pointer to next link in li
st
```

```
    Node *      prev;                                  // pointer to previous link i
n list

    Node () { next = prev = NULL; }                    // default constructor
    Node (const T & x) { next = prev = NULL;         // constructor with item
                                item = x; }
};

template <class T>
class List
{
  public:
    List();                                            // default constructor
    List(const List<T>& source);                          // copy constructor
    ~List();                                           // destructor

    List<T> &      operator=   (const List<T>& source);            // assignment operator
    int         length      ()    const { return size; }    // return the length of the l
ist
    void        append      (const T& x);            // append an item to the end of the
list
    void        insert      (int index, const T& x); // insert an item in position index
    T    pop         (int index = -1);               // delete item at position index (or
 last
                                                       // item if no index given)
    T &  operator[]  (int index);                    // indexing operator
    int         index       ( const T &x );          // return the index of the first occ
urrence of x
    string      str();                                 // return the string represen
tation

    void        resetForward(void);                    // reset forward iterator to
the head of the list
    T    next();                                     // return the next item in the list
and advance
                                                       // forward iterator pointer
    void        resetReverse(void);                    // reset reverse iterator to
the tail of the list
    T    prev        (void);                         // return the prev item in the list
and advance
                                                       // reverse iterator pointer
 private:
    Node<T>     *head,                                 // head of the linked list
                *tail,                                 // tail of the linked list
                *currentFwd,                           // current pointer for the fo
rward iterator
                *currentRev;                           // current pointer for the re
verse iterator
    int         size;                                  // length of the list


    void        copy        (const List<T>& source);          // copy source list to thi
s list
    void        dealloc     ();                        // deallocate the list
    Node<T>*       _find       (int index);               // return a pointer to the
 node in position index


    friend ostream&    operator<<  (ostream& os, const List<T>& l)
    {
        /*
         string str = "";
         Node<T> * temp = head;
         char Reason [50];
         str += "[";
         while (temp != NULL)
         {
                 if (temp -> next != NULL)
                 {
                         sprintf(Reason,"%d", temp -> item);
```

```
                        str+= Reason;
                        str += ", ";
                }

                else
                        if(temp -> next == NULL)
                        {
                                sprintf(Reason, "%d", temp -> item);
                                str+= Reason;
                        }
                        temp = temp -> next;
        }
        str += "]";
        return str;
        */                              //Returns error
    }

};

//===========================================================================
// IndexError
// This class implements an exception for an indexing error.
//===========================================================================
class IndexError {                                          // index error exception
public:
    IndexError() {};
    ~IndexError() {};
    const char *Reason () const { return "Index out of bounds."; }
};


//===========================================================================
// StopIteration
// This class implements an exception for iterating (forward or backward) beyond
// the start/end of the list.
//===========================================================================
class StopIteration {
public:
        StopIteration() {};
        ~StopIteration() {};
        const char *Reason () const { return "Iteration error \n System self destructing
in\n 3... \n 2... \n 1...";}

};      // stop iteration exception


#endif
#include "list.cc"
```

```
/*
Taylor Heilman
Stack class using linked lists
Stack2.cc
*/


#include "Stack2.h"
#include <stdlib.h>
#include <iostream>
using namespace std;

//============================================
//                      Default Constructor
//============================================
Stack::Stack ( void )
{
    Link * head = NULL;
    top = 0;

}

//============================================
//                              Destructor
//============================================

Stack::~Stack   ( void )
{
    Link * temp = head;
        while( temp != 0 ) {
        Link* next = temp->next;
        delete temp;
                temp = next;
}
        head = 0;
}

//============================================
//                              Push
//============================================
void        Stack::push    ( int item )
{
        Link * temp;
    temp = new Link;
    temp -> item = item;
    temp -> next = head;
    head = temp;
    top++;
}

//============================================
//                              Pop
//============================================
int         Stack::pop     ( void )
{
    if ( head == NULL )
    {
        cout << "Error: cannot pop from empty stack\n";
        exit(1);
    }

    else
        {

        Link * temp = head;
                head = head -> next;
                int x = temp -> item;
                delete temp;
                top--;
                return x;
```

```
        }


}

//============================================
//                              Size
//============================================
int         Stack::size    ( void )
{
    return top;
}
```

```
//===========================================
// Matt Kretchmar
// March 9, 2015
// Stack.h
//===========================================

#include <iostream>
using namespace std;

#ifndef STACK_H
#define STACK_H

#define DEFAULT_CAPACITY 5

class Stack
{
private:
    int top;        // index of top (empty) item
    int capacity;   // size of array for stack
    int *stack;     // dynamically allocated array
                    // to hold stack
public:
               Stack   ( void );
              ~Stack   ( void );
    void       push    ( int item );
    int        pop     ( void );
    int        size    ( void );
};

#endif
```

```
/*
Taylor Heilman
March 5, 2015

project6.cc
Project 6: Stacks With Dynamic Arrays
The goal of this project is to implement
stack behavior using dynamically allocated arrays.
*/


#include <iostream>
using namespace std;

int main ( void)
{
        int * list;             // pointer
        list = new int[5];      // dynamically allocating 5 places
        int capacity = 5;       // initialize capacity of allocated memory
        int length = 0;         // initialize length of array
        int num;
        char letter;            // p,q,s,x
        while (true)
        {
                if (length == capacity)  // Array is full
                {
                        int * tmp = new int [capacity + 5];  //Create a new, larger array

                        for(int i=0; i < length; i++)
                                tmp[i] = list[i];                       // copy old array
 into new, larger array

                        delete [] list;         // delete old array
                        list = tmp;             // change pointer to new array
                        capacity = capacity + 5;   // add 5 to capacity of array
                }



                cin >> letter;   // p, q, s, or x

                //====================================================================
                // Quit
                //====================================================================

                if (letter == 'q')  // Quit
                {
                        delete [] list;         // delete the allocated memory
                        exit(1);                        // quit program
                }

                //====================================================================
                // Push
                //====================================================================

                else if (letter == 'p')    // Push
                {
                        cin >> num;                             // value added to array
                        list[length] = num;             // set array index to input numbe
r
                        length  ++;                             // size of array increase
s by 1, move pointer up
                }


                //====================================================================
                // Pop
                //====================================================================

                else if (letter == 'x')         // Pop
```

```
                {
                        if (length == 0)        // empty array
                        {
                                delete [] list;         // delete the allocated memory
                                exit(1);                        // quit program
                        }
                        cout << list[length-1] << '\n' ;        // print number at top of
 stack
                        length--;                                                       /
/ length of array decreases by 1
                }


                //====================================================================
                // Size
                //====================================================================

                else if (letter == 's')         // Size
                {
                        cout << length << endl;   // Print amount of items in array
                }



                //====================================================================
                // For Troubleshooting
                //====================================================================


                //for (int j=0;j<length;j++)            // Print out the array
                //      cout <<  list[j] << endl;

                //cout << "capacity: " << capacity << '\n' ;  //See the size of allocated
 memory
        }


        return 0;
        }
```