```
/*
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;
        }
```

```
// pq.h
// This MinPriorityQueue template class assumes that the class KeyType has
// overloaded the < operator and the << stream operator.

#ifndef PQ_H
#define PQ_H

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

template <class KeyType>
class MinPriorityQueue : public MinHeap<KeyType>
{
  public:
    MinPriorityQueue();              // default constructor
    MinPriorityQueue(int n);         // construct an empty MPQ with capacity n
    MinPriorityQueue(const MinPriorityQueue<KeyType>& pq);  // copy constructor

    KeyType* minimum() const;                    // return the minimum element
    KeyType* extractMin();                       // delete the minimum element and return it
    void decreaseKey(int index, KeyType* key);   // decrease the value of an element
    void insert(KeyType* key);                   // insert a new element
    bool empty() const;                          // return whether the MPQ is empty
    int length() const;                          // return the number of keys
    std::string toString() const;                // return a string representation of the MPQ
    bool find(KeyType* key);
    string findCode(KeyType* key, int lenght);

    // Specify that MPQ will be referring to the following members of MinHeap<KeyType>.

    using MinHeap<KeyType>::A;
    using MinHeap<KeyType>::heapSize;
    using MinHeap<KeyType>::capacity;
    using MinHeap<KeyType>::parent;
    using MinHeap<KeyType>::swap;
    using MinHeap<KeyType>::heapify;


    /* The using statements are necessary to resolve ambiguity because
       these members do not refer to KeyType.  Alternatively, you could
       use this->heapify(0) or MinHeap<KeyType>::heapify(0).
    */
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinPriorityQueue<KeyType>& pq);

class FullError { };    // MinPriorityQueue full exception
class EmptyError { };   // MinPriorityQueue empty exception
class KeyError { };     // MinPriorityQueue key exception
class IndexError { };     // MinPriorityQueue key exception

#include "pq.cpp"

#endif
```

88

call MinHeap constructors via initializer lists

```
#include "pq.h"
```

pq.cpp included from pq.h in template class

```
/*
 *      Default Constructor
 *      Precondition:
 *      Postcondition:
 */
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue()
{

}



/*
 *      Construct an empty MinPriority Queue with capacity n
 *      Precondition:
 *      Postcondition:
 */
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(int n)
{
        MinHeap<KeyType> heap(n);
}



/*
 *      Copy Constructor
 *      Precondition: MinPriorityQueue pq  must be a legitimate MinPriorityQueue
 *      Postcondition: The target MinPriorityQueue is a copy of the other MinPriorityQue
 */
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(const MinPriorityQueue<KeyType>& pq)
{
        heapSize = pq.heapSize;
        capacity = pq.capacity;

        A = new KeyType*[pq.capacity];

        for (int i = 0; i < pq.capacity; i++)
        {
                A[i] = pq.A[i];
        }
}
```

call MinHeap copy constructor

```
/*
 *      Return the Minimum Element
 *      Precondition: A valid MinPriorityQueue with Size >= 0
 *      Postcondition: Returns the smallest element in the Queue
 */
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::minimum() const
{
        if (heapSize <= 0)
        {
                throw EmptyError();
        }
```

```
        else
        {
                return A[0];
        }
}



/*
 *      Delete the Minimum Element and return it
 *      Precondition: A valid MinPriorityQueue with Size >= 0
 *      Postcondition: Returns the smallest element, deletes said element, and keeps a valid M
in Heap
 */
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::extractMin()
{
        if (heapSize <= 0)
        {
                throw EmptyError();
        }

        else
        {
                KeyType* min = A[0];
                A[0] = A[heapSize-1];
                heapSize--;
                heapify(0);
                return min;
        }
}




/*
 *      Decrease the value of an element
 *      Precondition: A valid MinPriorityQueue with Size >= 0,
 *      Postcondition: The element at the inputted index has the value of the inputted key
 */
template <class KeyType>
void MinPriorityQueue<KeyType>::decreaseKey(int index, KeyType* key)
{
        if (index >= heapSize || index < 0)
        {
                throw IndexError();
        }
        if (key > A[index])
        {
                throw KeyError();
        }
        else
        {
                A[index] = key;
                while (index > 0 && *A[parent(index)] > *A[index])
                {
                        swap(index, parent(index));
                        index = parent(index);
                }
        }
```

```
}



/*
 *      Insert a New Element
 *      Precondition: a valid MinPriorityQueue with Heapsize != capacity
 *      Postcondition: a valid MinPriorityQueue containing the inputted key
 */
template <class KeyType>
void MinPriorityQueue<KeyType>::insert(KeyType* key)
{
        if (heapSize == capacity)
        {
                throw FullError();
        }
        else
        {
                A[heapSize] = key;
                heapSize++;
                int index = heapSize-1;
                decreaseKey(index, key);


        }
}


template <class KeyType>
bool MinPriorityQueue<KeyType>::find(KeyType* key)
{
        for(int i = 0; i < heapSize; i++)
        {
                if (A[i]->name == key->name)
                        return true;

        }

        return false;
}


template <class KeyType>
string MinPriorityQueue<KeyType>::findCode(KeyType* key, int length)
{
        string codenum;
        //cout << heapSize << endl;
        for(int i = 0; i < length; i++)
        {
                //cout << "name = " << A[i]->name << endl;
                if (A[i]->name == key->name)
                {
                        //cout << "got here" << endl;
                        codenum = A[i]->code;
                        return codenum;
                }
        }
        return "NIF";
}
```

these are not needed & facilitate an inefficient algorithm — see below.

```
/*
 *      Return whether the MPQ is empty
 *      Precondition: A valid MinPriorityQueue with Size >= 0
 *      PostCondition:  returns true id heapsize = 0, returns false if heapsize > 0
 */
template <class KeyType>
bool MinPriorityQueue<KeyType>::empty() const
{
        return heapSize == 0;
}



/*
 *      Return the numbers of keys
 *      Precondtion: a Valid MPQ
 *      Postcondition: returns the length of MPQ
 *
 */
template <class KeyType>
int MinPriorityQueue<KeyType>::length() const
{
        return heapSize;
}



/*
 *      return a string representation of the MPQ
 *      Precondtion: a valid MPQ
 *      Postcondition: a printed string of the MPQ in a list
 */

template <class KeyType>
std::string MinPriorityQueue<KeyType>::toString() const
{
        std::ostringstream sstream;

        sstream << "[";
        for (int i = 0; i < heapSize; i++)
        {
                sstream << *(A[i])  << " ";
        }

        std::string s = sstream.str();
        string st = s.substr(0, s.size()-1);
        st.append("]");
        return st;
}
```

```
// test_pq.cpp
// Testing if our stuff works
#include <iostream>
#include <stdlib.h>
#include <assert.h>
#include "pq.h"

using namespace std;

void test_pqInsert()
{
        MinPriorityQueue<int> west(10);
        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        assert(west.toString() == "[2,5,24,14]");
        cout << "Insert Assertion Passed" << endl;
}

void test_pqMin()
{
        MinPriorityQueue<int> west(10);
        try
        {
                west.minimum();
                assert(false);
        }
        catch(EmptyError x)
        {
                cout<< "caught" << endl;
        }

        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        assert(*west.minimum() == 2);
        cout << "Minimum Assertion Passed" << endl;
}

void test_pqExtractMin()
{
        MinPriorityQueue<int> west(10);
        try
        {
                west.extractMin();
```

```
                assert(false);
        }
        catch(EmptyError y)
        {
                cout<< "caught2" << endl;
        }


        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        assert(*west.extractMin() == 2);
        assert(west.toString() == "[5,14,24]");
        cout << "Extract Minimum Assertion Passed" << endl;
}

void test_pqEmpty()
{
        MinPriorityQueue<int> west(10);
        assert(west.empty() == true);
        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        assert(west.empty() == false);
        cout << "Empty Assertion Passed" << endl;
}

void test_pqLength()
{
        MinPriorityQueue<int> west(10);
        assert(west.length() == 0);
        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        assert(west.length() == 4);
        cout << "Length Assertion Passed" << endl;
```

```
}

void test_pqCopy()
{
        MinPriorityQueue<int> west(10);
        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        MinPriorityQueue<int> east(west);
        assert(east.toString() == "[2,5,24,14]");
        cout << "Copy Assertion Passed" << endl;
}


void test_pqDecreasekey()
{
        MinPriorityQueue<int> west(10);
        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        int *z = new int;
        *z = 9;
        try
        {
                west.decreaseKey(9,z);
                assert(false);
        }
        catch(IndexError t)
        {
                cout << "caught3" << endl;
        }
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        MinPriorityQueue<int> east(west);
        cout << west.toString() << endl;
        assert(east.toString() == "[2,5,24,14]");
        cout << "Decreasekey Assertion Passed" << endl;


}


int main()
{
        test_pqInsert();
        test_pqMin();
        test_pqExtractMin();
```

```
        test_pqEmpty();
        test_pqLength();
        test_pqCopy();
        test_pqDecreasekey();
        return 0;
}
```

```
#include "pq.h"

using namespace std;

class Node {
  public:
        char name;                  // character
    int f;                                  // frequency
        Node *left, *right;             // pointers to the left and right child
    string code;                    // prefix code

        Node()              //default constructor
    {
        name = (char) 0;
        code = "";
        f = 0;
        left = right = 0;
    }

    Node(int ff, char n = (char) 0)     //constructor that creates a node
    {
        name = n;
        f = ff;
        left = right = 0;
        code = "";
    }

    ~Node()                             //destructor
    {
        delete left;
        delete right;
    }


    bool operator<(Node &comp)
    {
        return f < comp.f;
    }

    bool operator>(Node &comp)
    {
        return f > comp.f;
    }

    bool operator==(Node &comp)
    {
        return f  == comp.f;
    }


        static void display(Node*, bool);
        static void encode(Node*);


    friend ostream& operator <<(ostream&, Node&);
};
```

NULL

Don't deallocate in destructor because not allocated in constructor.

```
#include "node.h"
#include <stdlib.h>
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
using namespace std;


string codes;
Node *root;
string head;

/*
 *              Searches the Minimum Priority Queue for a node that matches the character give
n and adds its code to the codes global variable
 *              Precondition: A valid MPQ must exist, and name must be a valid char
 *              Postcondition: The correct corresponding code of the char given is added to th
e codes global variable
 */
void searchName(char name, Node* node, bool leaf = 1)
{

    if (node == 0)
    {
        return ;
    }

    searchName(name, node->left, leaf);

    if (leaf == 1)
    {
        if (node->name == name)         //no children, is a leaf
        {
                codes += node->code;
        }
    }

    searchName(name,node->right, leaf);
}




/*
 *              Creates the character key that appears in our compressed file
 *              Precondition: A valid Huffman coded MPQ exists
 *              Postcondition: The header string is filled with character keys
 */
void header(Node* node, bool leaf = 1)
{
    if (node == 0)
    {
        return;
    }

    header(node->left, leaf);

    if (leaf == 1)
    {
        if (node->left == 0 && node->right == 0)        //no children, is a leaf
        {
```

```
                string f = to_string(node->f);
            char name = node->name;
            string code = node->code;
            head = head + name + "[" + f + "]" + "(" + code + ")";
        }
    }

    header(node->right, leaf);
}
```

```
/*
 *              Decompressed the given sourcefile, and creates a new file with the contents of
 the decompressed file
 *              Precondition: compressed is a valid sourcefile, and output is a valid name of
an output file
 *              Postcondition: the source file is decompressed and the output file contains th
e decompressed contents
 */
void decompress(string compressed, string output)               //compressed file, output file
{
        string str;
        string decomp;
        bool child;
        int i = 0;
        int j=0;
        int bit;
        string fcode;
        string bits;
        unsigned char buffer;

        ifstream file(compressed);              // sourcefile

        while(getline(file, str))
        {
                decomp += str;          // read in source file
        }


        int unique =0;
        int realnum;
        int t = 0;
        string name;
        string code;

        while(t<1)
        {

                if (decomp[i+1] = '[')
                {
                        name = name+decomp[i];                  //NAME
```

? use well-named boolean to make this clearer

```
                i=i+2;
                while(decomp[i] != ']')
                {
                        int num = num + decomp[i];
                        i++;
                }

                i=i+2;
                while(decomp[i] != ')')
                {
                        code = code+decomp[i];
                        i++;
                }
                code += ',';

        }
unique++;
        i++;


        if( decomp[i] == '$' &&  decomp[i+1] == '$')
                t = 1;
}

int namelen = name.length();
string aname[unique];
string acode[unique];
for (int h = 0; h < namelen; h++)
{

        aname[h] =  name[h];
}



int comma =0;
for (int j = 0; j < unique; j++)
{
        string codearray = "";
        while(code[comma] != ',')
        {
                codearray = codearray + code[comma];
                comma++;
        }
        acode[j] =  codearray;
        comma++;
}



int len2 = decomp.length();


i = i+2;

for( i; i< len2; i++)
{
        buffer = decomp[i];

        for(j = 0; j < 8; j++)
        {
                bit = buffer >>7;
                string stringbit = to_string(bit);
```

This is very hard to follow — use better variable names & comments!

```
                        bits = bits+ stringbit;
                        buffer = buffer << 1;
                }

        }


        ofstream outputFile;
        outputFile.open (output);


        int count = 0;
        int bitsL = bits.length();
        while (count < bitsL)
        {
                bool match = false;
                string check = "";
                while (!match)
                {
                        if ((bits[count + 1] != '0') && (bits[count + 1] !=  '1'))      //If t
he input is over, quit the loop
                        {
                                return;
                        }
                        check = check + bits[count];
                        for (int k = 0; k < unique; k++)
                        {
                                if (check == acode[k])
                                {
                                        outputFile << aname[k];
                                        match = true;
                                }
                        }
                        count++;
                }
        }

}




/*
 *              Compresses the give sourcefile given the root node
 *              Precondition: A valid sourcefile and root node is passed int
 *              Postcondition: The output file is a correctly compressed version of the source
 file.
 */
void compress(Node* node, string input, int length, string output)
{
        char item;
        unsigned char buffer = 0;
        int i = 0;
        int count = 0;
        int num = 0;
        ofstream outputFile;
        outputFile.open (output);                       //should be *argv[3]
        string precode;

        header(root);           //creates header
        head = head + '$';
```

```
        head = head + '$';

        outputFile << head;             //send header to output file
        outputFile << "\n";


        while(input[i] != ''')                          //traversing source file
        {

                item = input[i];
                searchName(item, node, 1);                              //adds
 input[i] code to 'codes'

                i++;
        }



        int eight=  codes.length();
        int remainder = eight % 8;
        eight = 8-remainder;


        i = 0;          //reset i


        while(codes[i] == '0' || codes[i] == '1')               //iterate through codes string
        {
                if (codes[i] == '1')
                {
                        buffer = buffer << 1;           //shift left
                        buffer = buffer | 1;                    //add 1 to end of bufffer
                        count ++;
                        num++;
                }

                else
                {
                        buffer = buffer << 1;           //shift left
                        buffer = buffer | 0;            // add 0 to end of buffer
                        count ++;
                        num++;
                }


                if(count == 8)                          //8 bits of buffer have been filled
                {
                        outputFile << buffer;           //add buffer to mid file
                        buffer = 0;
                        count = 0;
                }

                i++;
        }


        i = 0;
        if (count > 0 )
        {
                for (i=0; i< eight; i++)
                {
                        buffer = buffer << 1;           //shift left
```

↑ root

- Codes should not be global – side effect makes code harder to follow
- codes can be a really long string – problem for large files

-3

Factor out common code

does nothing

```
                    buffer = buffer | 0;                    // add 0 to end of buffer
                }
                outputFile << buffer;           //add buffer to mid file with extra 0's
        }

        outputFile.close();

}
```

```
/*
 *              Creates a string representation of the tree
 *              Precondition: a valid tree with a root node exists
 *              Postcondition: A correct representation of the tree is displayed
 */
void Node::display(Node* node, bool leaf = 1)
{
    if (node == 0)
    {
        return;
    }

    display(node->left, leaf);

    if (leaf == 1)
    {
        if (node->left == 0 && node->right == 0)        //no children, is a leaf
        {
            cout << *node << ", ";
        }
    }
    else
    {
        cout << *node << ", ";
    }

    display(node->right, leaf);
}
```

Should be in a node.cpp file.

```
/*
 *              Creates the codes for each character from the tree
 *              Precondition: a valid tree with a root exists
 *              Postcondition: Every node in the tree has a correct code made of 0's and 1's
 */
void Node::encode(Node* node)           // determines prefix code for each char
{
        if (node == 0)
        {
        return;
    }
```

```
    if (node->left != 0)        //checking node isn't a leaf
    {
        node->left->code = node->code + "0";    //left child adds code 0
                encode(node->left);                                         //call encode(
);
    }
    if (node->right != 0)       //checking node isn't a leaf
    {
        node->right->code = node->code + "1";  //right child adds code 1
        encode(node->right);
    }
}

/*
 *              Overloads the << operator in order to print out a string representation of the
 node
 */
ostream& operator <<(ostream &out, Node &node)
{
    out << node.name << "(" << node.f << ")" << ":" << node.code;
    return out;
}




/*
 *              The huffman code that creates the tree out of the MPQ
 *              Precondition: a valid MPQ is inputted
 *              Postcondition: A compressed file is created
 */

                                // pq with nodes and chars  # of nodes  input file    output f
ile
void HuffmanCode(MinPriorityQueue<Node>* q, int length, string input, string output)
{

    for (int j = 0; j < length - 1; j++)
    {
        Node *left = q->extractMin();                   //Node x = smallest freq char in PQ
        Node *right  = q->extractMin();                 //Node y = 2nd smallest freq char in P
Q
        Node *z = new Node(left->f + right->f, '#');    // parent node of nodeLeft and nodeRig
ht
        z->left = left;                             //connected by 0 edge
        z->right = right;                           // connected by 1 edge
        q->insert(z);                                                   // insert into PQ
    }

    root = q->extractMin();             //most frequent char in file


   cout << "Full tree (inorder):\n";
   Node::display(root, 0);                                  //Tree with nodes


    Node::encode(root);                                     // determine prefix codes based off tr
ee

        cout << "\nHuffman Code:\n";
        Node::display(root);
    cout << "\n";
```

```
        compress(root, input, length, output);

}



/*
 *              Creates the nodes for the MPQ out of the input file
 *              Precondition: A valid sourcefile exists
 *              Postcondition: A valid compressed outfile with the given name is created
 */
void begin(string input, string output)
{
        int length = 0;
        int i = 0;
        char sentinal = '‘';
        string str;
        string source;

        ifstream file(input);           // sourcefile

        while(getline(file, str))
        {
                source += str;          // read in source file
        }
        source += "‘";          //insert sentinal character

        while(source[i] != sentinal)
        {
                length++;       // find length of file
                i++;

        }

    MinPriorityQueue<Node> *t = new MinPriorityQueue<Node> (length); //create new PQ
        int unique = 0;         // amount of unique chars in source file


        for(int j=0; j<length; j++)
        {
                char name = source[j];                  //looking at character in source at in
dex j
                Node *n = new Node(0, name);    // create new node
                if (!t->find(n))                                // check if node is already in
 PQ
                {
                        int frequency = 0;                      // node isn't in PQ
                        for(int k = j;k<length;k++)
                        {
                                if(name == source[k])           // find frequency of char in s
ource file
                                {
                                        frequency++;
                                }
                        }
                        t->insert(new Node(frequency, name));   //insert node into PQ with fre
```

problematic if file won't fit in memory!

source.length() ?

This is really inefficient — just pass over input once & don't search MPQ.

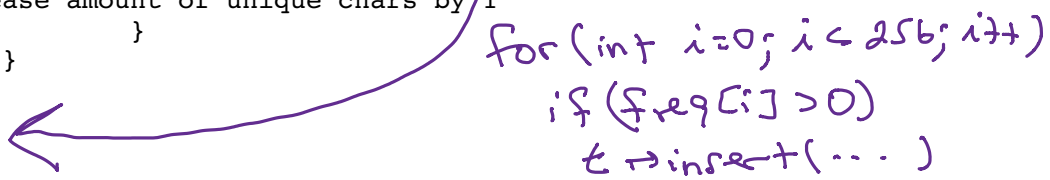
memory leak!

int freq[256];

freq[name]++;

```
q, char
                        unique++;
// increase amount of unique chars by 1
                }
        }

        HuffmanCode(t, unique, source, output); //call HuffmanCode (PQ t, chars #, source, out
put)
}
```




```
int main(int argc, char** argv)
{
        if (argc != 4)
        {
                return 1;       //not properly inputted
        }

        else
        {
                string arg1 = argv[1];
                if(arg1 == "-c")        //encode .txt file, and creating .huff file
                {
                        string sourceFile = argv[2];
                        string outputFile = argv[3];
                        begin(sourceFile, outputFile);
                }
                else if (arg1 == "-d")  //decompress, decode
                {
                        string sourceFile = argv[2];
                        string outputFile = argv[3];
                        decompress(sourceFile, outputFile);
                }
                else
                {
                        return 1; // improper input
                }
        }


        return 0;
}
```