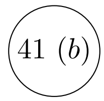
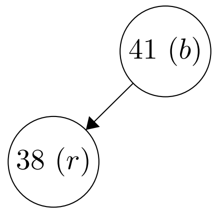
Ben Liepert and Gezim Saciri - Project 7

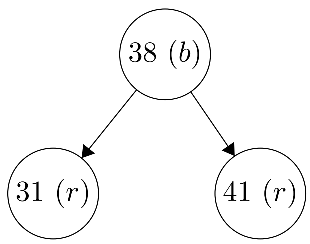
CS271 Data Structures

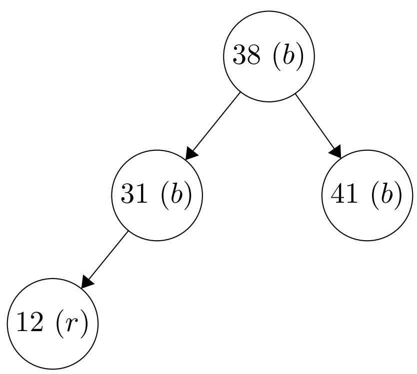
Wednesday, April 19th

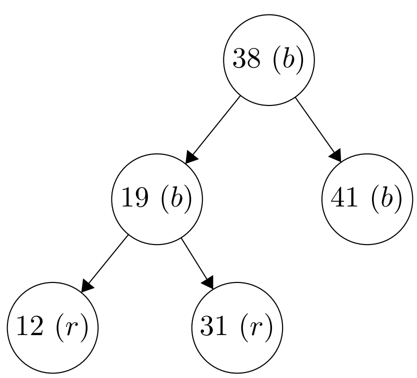
1.

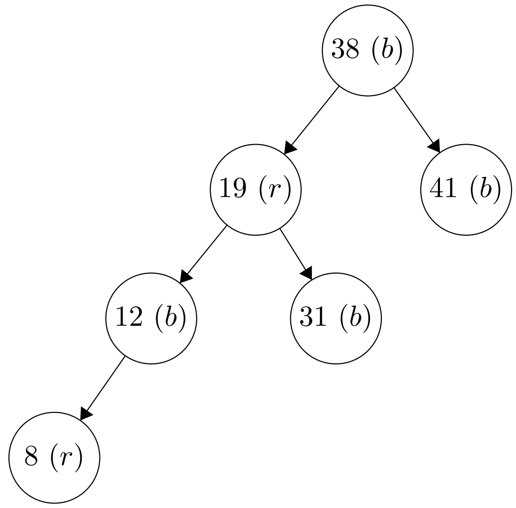












2. Proof Remember that if a node in an RBT is red, its children must be black. Thus, the longest possible path from a node to its leaf must alternate between red and black nodes. If we think of this path as the right subtree of the root, our left subtree must have equal black height. The shortest this path can possibly be, in order for the black height property to be satisfied, is if it is only comprised of black nodes. Thus, the right subtree has one red node for every black node on the path, because they are alternating. In other words, there would be 2*b nodes in this path where b is the number of black nodes. We know because of the black height property that there must be exactly b black nodes in the path down the left subtree, meaning there are two times the number of nodes in the right. The largest path is bound by 2 times the length of the shortest path because we don't necessarily need red nodes in the right subtree.

5.) On the *un-randomized* movies file, Binary Search Tree average time to insert over 100 trials: 4.9064
Hash Table average time to insert over 100 trials: 0.0535483
Red Black Tree average time to insert over 100 trials: 0.0258046
On the words file, the times averaged over 100 trials are:
BST: took so long for one trial (more than a minute) that we gave up and concluded it is terrible.
Hash Table: .485501 seconds
RBT: .194656 seconds
The fact that the RBT implementation of our dictionary class is faster than the Hash Table, which is faster than the BST, makes sense. For the BST, the alphabetic order of the movies causes an the BST to insert all values on the next right subtree, effectively creating a linked list. This will cause the BST to insert in $O(n^2)$ time. However, when we randomize our moviesmpaa.txt, our BST preforms much faster at avg time of 0.0711234, which is comparable to our Hash Table. Our Hash Table implementation is faster due to the fact that it inserts values at the head of its linked lists, which allows it to preform in constant time $O(1)$. However, due to implementation of our dictionary class, we must first see if a movie is already in the hash table, which in the worst case, where all movies are in the same slot, causes the runtime to be linear. Due to this, our runtime was not the optimal constant time a bare-bones hash table is capable of. The RBT, which is the most efficient implementation, runs insert in $O(lgn)$. This is due to the fact that a RBT can only have height $O(lgn)$ for n nodes, it only takes $O(lgn)$ to traverse to the farthest leaf and insert it. The insert-fixup function's while loop runs in $O(lgn)$ time for worst case, which doesn't affect the overall runtime. Therefore, we can see that RBT is the most efficient in terms of the dictionary implementation, due to its balanced tree which allows us to search for duplicates quickest in $O(lgn)$ worst case time and insert in $O(lgn)$ time.

```
//Gezim Saciri & Ben Liepert
//source file for dict.h
#include <iostream>
#include <string>
#include <sstream>
#include "dict.h"
using namespace std;

//=======================================
// get
template <class KeyType>
KeyType*  Dictionary<KeyType>::get(const KeyType& k){
  return RedBlackTree<KeyType>::get(k);
}

//=======================================
// insert
template <class KeyType>
void  Dictionary<KeyType>::insert(KeyType *k){
  if(RedBlackTree<KeyType>::get(*k) == NULL){ //can't double insert
    RedBlackTree<KeyType>::insert(k);
  }
}

//=======================================
// remove
template <class KeyType>
void  Dictionary<KeyType>::remove(const KeyType& k){
  RedBlackTree<KeyType>::remove(k);
}
```

```
//Gezim Saciri & Ben Liepert
//dict.h
#ifndef DICT_H
#define DICT_H

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

template <class KeyType>
class Dictionary : public RedBlackTree<KeyType>
{
  public:
    Dictionary():RedBlackTree<KeyType>() { } //default
    Dictionary(const Dictionary<KeyType>& d):RedBlackTree<KeyType>(d) { }

    KeyType*    get(const KeyType& k); //want const at the end here?
    void        insert(KeyType *k);
    void        remove(const KeyType& k);
    bool        empty() const { return RedBlackTree<KeyType>::empty(); }
};

#include "dict.cpp"

#endif
```

```
# Makefile for set
#****************************************************************

CPPOPTIONS = -std=c++11 -g
LDOPTIONS =

# ***************************************************************
# Entry to bring the package up to date
#    The "make all" entry should be the first real entry

all: test_rbt test_node test_dict test_movie test_query

test_rbt: rbt.h rbt.cpp test_rbt.cpp node.h node.cpp
	g++ $(CPPOPTIONS) -o test_rbt test_rbt.cpp

test_node: node.h node.cpp test_node.cpp
	g++ $(CPPOPTIONS) -o test_node test_node.cpp

test_dict: dict.h dict.cpp test_dict.cpp
	g++ $(CPPOPTIONS) -o test_dict test_dict.cpp

test_movie: movie.h movie.cpp test_movie.cpp
	g++ $(CPPOPTIONS) -o test_movie test_movie.cpp

test_query: movie.h movie.cpp query_movies.cpp dict.h dict.cpp rbt.h rbt.cpp
	g++ $(CPPOPTIONS) -o query_movies query_movies.cpp


# ***************************************************************
# Standard entries to remove files from the directories
#    tidy  -- eliminate unwanted files
#    clean -- delete derived files in preparation for rebuild

tidy:
	rm -f ,* .,* *~ core a.out *.err

clean: tidy
	rm -f *.o
```

```
//Gezim Saciri & Ben Liepert
//source file for movie.h
#include <iostream>
#include <sstream>
#include <string>
#include "movie.h"
using namespace std;

//==============================
// defualt constructor
Movie::Movie(){
  title = "Drew Bapst: The Musical";
  cast = "Drew Bapst  Josh Blaz  Brian Tran";
}

// =============================
// primary constructor
Movie::Movie(std::string &t, std::string &c){
  title = t;
  cast = c;
}

// =============================
// copy constructor
Movie::Movie(const Movie& m){
  title = m.title;
  cast = m.cast;
}

// =============================
// destructor
Movie::~Movie(){
 //no mem allocation
}

string Movie::toString(){
  stringstream ss;
  ss << "(" << title << ":: " << cast << ")";
  return ss.str();
}

//========================================
// < operator
bool Movie::operator<(const Movie &m){ //define comparison to return title
  return title < m.title;
}

//========================================
// != operator
bool Movie::operator!=(const Movie &m){ //define comparison to return title
  return title != m.title;
}
```

```
//Gezim Saciri & Ben Liepert
//movie.h
#ifndef MOVIE_H
#define MOVIE_H

#include <iostream>

class Movie
{
  public:
    Movie();
    Movie(std::string &t, std::string &c);
    Movie(const Movie& m);
    ~Movie();

    bool operator<(const Movie &m);
    bool operator!=(const Movie &m);
    std::string toString();
    std::string title;
    std::string cast;
};

#include "movie.cpp"

#endif
```

```
//Gezim Saciri & Ben Leipert
//source file for node.h
#include <iostream>
#include <string>
#include <sstream>
#include "node.h"
using namespace std;

//==============================
// defualt constructor
// pre-condition: nothing
// post-condition: creates an empty node
template <class KeyType>
Node<KeyType>::Node(){
  //errythang empty
  key = NULL;
  parent = NULL;
  leftChild = NULL;
  rightChild = NULL;
  color = BLACK;
}

// =============================
// primary constructor
// Pre-condition: nothing
// Post-condition: creates a basic node
// =============================
template <class KeyType>
Node<KeyType>::Node(KeyType &k){
  key = k.key;
  parent = NULL;
  leftChild = NULL;
  rightChild = NULL;
  color = BLACK;
}

// =============================
// copy constructor
// Pre-condition: requires an existing Node objects
// Post-condition: constructs a copy of the object passed in.
// =============================
template <class KeyType>
Node<KeyType>::Node(const Node<KeyType>& n){
  key = n.key;
  parent = n.parent;
  leftChild = n.leftChild;
  rightChild = n.rightChild;
  color = n.color;
}

// =============================
// destructor
// =============================
template <class KeyType>
Node<KeyType>::~Node(){
    // we don't use new anywhere
}

// =============================
// < operator overwrite
// Pre-condition: requires two Nodes for the comparison
// Post-condition: returns a boolean < comparison of the objects
// =============================
```

```
template <class KeyType>
bool Node<KeyType>::operator<(const Node<KeyType> &n){
    return *key < *n.key;
}

// =============================
// to string
// Pre-condition: needs a Node object
// Post-condition: object remains unchanged, returns object represented as a string
// =============================
template <class KeyType>
string Node<KeyType>::toString() const{
        stringstream ss;

  if (key == NULL){
    if(color == RED){
      ss << "(" << ", RED)";
    }else{
      ss << "(" << ", BLACK)";
    }
  }else{
    if(color == RED){
      ss << "("  << *key << ", RED)";
    }else{
      ss << "("  << *key << ", BLACK)";
    }
  }
        return ss.str();
}

template <class KeyType>
ostream& operator<<(ostream& stream, const Node<KeyType> &n ){
  stream << n.toString();
  return stream;
}
```

```
//Gezim Saciri & Ben Liepert
//node.h
#ifndef NODE_H
#define NODE_H

#include <iostream>

enum colors {BLACK, RED};

template <class KeyType>
class Node
{
  public:
    Node();             // default constructor
    Node(KeyType &k);        // construct a Node
    Node(const Node& n);  // copy constructor
    ~Node(); // destructor

    KeyType* key; //ptr to key of our node
    Node<KeyType>* parent;
    Node<KeyType>* leftChild;
    Node<KeyType>* rightChild;
    colors color;

    std::string toString() const;
    bool operator<(const Node &n);
};

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

#include "node.cpp"

#endif
```

```
//Gezim Saciri & Ben Liepert
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include "dict.h"
#include "movie.h"
#include <sys/time.h>

using namespace std;

void randomizeFile(string iFile, string oFile){
  //=============================================
  // count lines in original file
  ifstream inputFile1(iFile);
  if (!inputFile1.is_open()){
    cout << "Could not open input file (1), exiting.\n";
    exit(1);
  }
  string line1;
  int numLines = 0;
  while (getline(inputFile1, line1)){
    numLines++;
  }
  //=============================================
  // for every line, pick a random number between 1 and #lines,
  // take this line in the original and write it to the randomized file
  ofstream outputFile(oFile, ios_base::app); //we want to append to existing file
  if (!outputFile.is_open()){
    cout << "Could not open output file, exiting.\n";
    exit(1);
  }
  vector<int> visited;
  int randomLine = 0;
  for(int i = 0; i < numLines; i++){ // for every line
    bool getNewRandom = true;
    // while we need a new random number
    while(getNewRandom){
      randomLine = rand() % numLines; //pick one
      getNewRandom = false;
      // make sure it’s not in our vector
      for (int j = 0; j < visited.size(); j++){
        if(visited[j] == randomLine){
          getNewRandom = true;
          break;
        }
      }
      //if getNewRandom is still false, we got a unique random #
    }
    visited.push_back(randomLine); //put the number in our vector bc we’re using it
    int count = 0;
    string line;
    ifstream inputFile;
    inputFile.open(iFile); //open the input file from the top
    while(getline(inputFile, line)){ //iterate over every line in orig. file
      if(count == randomLine){ //if out random # matches current line #
        outputFile << line << ’\n’; //write to randomized file
        break; //break so we don’t go over all the other lines
      }
      count++; //increment line #
    }
  }
}
```

```
int main(){
  //randomizeFile("movies_mpaa.txt", "movies_mpaa_random.txt");

  //used this for testing timing

  double timeCount = 0.0;
  for (int i = 0; i < 100; i++){
    timeval timeBefore, timeAfter;
    long diffSeconds, diffUSeconds;
    ifstream inputFile("words");
    string line;
    //int iTabIndex;
    Dictionary<Movie> dict;
    gettimeofday(&timeBefore, NULL);
    while (getline(inputFile, line)){
      //iTabIndex = line.find('\t');
      Movie *m = new Movie;
      m->title = line;//line.substr(0,iTabIndex);
      //m->cast = line.substr(iTabIndex+1);
      dict.insert(m);
    }
    gettimeofday(&timeAfter, NULL);
    diffSeconds = timeAfter.tv_sec - timeBefore.tv_sec;
    diffUSeconds = timeAfter.tv_usec - timeBefore.tv_usec;
    timeCount += (diffSeconds + diffUSeconds/1000000.0);
  }
  cout << "Average time over 100 trials: " << timeCount/100 << endl;
/*
  ifstream inputFile("movies_mpaa.txt");
  if (!inputFile.is_open()){
    cout << "Could not open input file (1), exiting.\n";
    exit(1);
  }
  string line = "";
  int iTabIndex = -12;
  Dictionary<Movie> dict;
  int counter = 0;
  while (getline(inputFile, line)){
    iTabIndex = line.find('\t');
    Movie *m = new Movie;
    m->title = line.substr(0,iTabIndex);
    m->cast = line.substr(iTabIndex+1);
    dict.insert(m);
    Movie N(*m);
    assert(dict.get(N) == m); //proves that the thing is in there bruh
    counter++;
  }
  cout << "conter = " << counter << endl;

  //cout << "Enter a movie title (or exit to stop): ";
  string t = "";
  while(t != "exit"){
    cout << "Enter a movie title (or exit to stop): ";
    getline(cin, t);
    Movie M;
    M.title = t;
    Movie *n = new Movie;
    n = dict.get(M);
    if (n == NULL && t != "exit"){
      cout << "Couldn't find that movie!\n" << endl;
    }else if (t != "exit"){
      cout << "Cast: " << n->cast << endl;
```

```
    }
  }
  */
}
```

```
// Ben Liepert & Gezim Saciri
// source file for RedBlackTree class
#include <iostream>
#include <sstream>
#include <cassert>
#include "rbt.h"
using namespace std;

//=======================================
// default constructor
template <class KeyType>
RedBlackTree<KeyType>::RedBlackTree(){
  nil = new Node<KeyType>;
  root = nil;
}

//=======================================
// destructor
template <class KeyType>
RedBlackTree<KeyType>::~RedBlackTree(){
  destroy(root); //I think this is all we need as opposed to the above
}

//=======================================
// copy constructor
// pre condition: A RedBlackTree that we will copy into this
// post condition: A new RedBlackTree that is a copy of the one passed in
template <class KeyType>
RedBlackTree<KeyType>::RedBlackTree(const RedBlackTree<KeyType>& rbt){
  nil = new Node<KeyType>;
  root = copy(rbt.root, nil, rbt.nil); //need to pass in the other tree's nil to know when to
stop traversing
}

//=======================================
// get
// pre condition: A RedBlackTree and a KeyType k that it will search for
// post condition: The RedBlackTree(unchanged) and will return value key or nil
template <class KeyType>
KeyType*  RedBlackTree<KeyType>::get(const KeyType& k) {
  Node<KeyType>* found = search(k);
  return (found == nil) ? NULL : found->key; //NULL if not in rbt, key otherwise
}

//=======================================
// insert
// pre condition: A RedBlackTree to insert values into, and a KeyType pointer k to insert
// post condition: The RedBlackTree with the KeyType pointer k inserted into the tree
template <class KeyType>
void  RedBlackTree<KeyType>::insert(KeyType *k){

  Node<KeyType> *par = nil; //keep track of parent w/this
  Node<KeyType> *c = root;
  while (c != nil){
    par = c;
    if (k == NULL){
      cout << "k is NULL\n";
      exit(1);
    }
    if (*k < *c->key){ //need to deref to compare vals
      c = c->leftChild;
    }else{
      c = c->rightChild;
```

```
    }
  }
  Node<KeyType> *i = new Node<KeyType>; //create node to insert
  i->color = RED;
  i->key = k; //key stores a pointer, k is a ptr
  i->parent = par;
  i->leftChild = nil;
  i->rightChild = nil;
  if (par == nil){ //empty tree
    root = i;
  }else if(*i->key < *par->key){
    par->leftChild = i;
  }else{
    par->rightChild = i;
  }
  insertFixup(i);
}
//=======================================
// insertFixup
// pre condition: A RBT that is not following the properties of the RBT after an insertion
// post condition: A RBT that follows all properties of RBT’s
template <class KeyType>
void RedBlackTree<KeyType>::insertFixup(Node<KeyType> *current){
  while(current->parent->color == RED){
    if(current->parent == current->parent->parent->leftChild){
      Node<KeyType> *uncle = current->parent->parent->rightChild;
      if(uncle->color == RED){          // case 1
        current->parent->color = BLACK;
        uncle->color = BLACK;
        current->parent->parent->color = RED;
        current = current->parent->parent;
      }
      else{
        if(current == current->parent->rightChild){ // case 2
          current = current->parent;
          leftRotate(current);
        }
        current->parent->color = BLACK;   // case 2 cont. and 3
        current->parent->parent->color = RED;
        rightRotate(current->parent->parent);
      }
    }
    else{
      Node<KeyType> *uncle = current->parent->parent->leftChild;
      if(uncle->color == RED){         // case 4
        current->parent->color = BLACK;
        uncle->color = BLACK;
        current->parent->parent->color = RED;
        current = current->parent->parent;
      }
      else{
        if(current == current->parent->leftChild){   // case 5
          current = current->parent;
          rightRotate(current);
        }
        current->parent->color = BLACK;        // case 5 cont. and 6
        current->parent->parent->color = RED;
        leftRotate(current->parent->parent);
      }
    }
  }
  root->color = BLACK;
}
```

```
//=====================================
// leftRotate
// pre condition: Takes an unbalanced RBT and given node and preforms a left rotation
// post condition: Balances out the RBT to fufill the RBT property
template <class KeyType>
void RedBlackTree<KeyType>::leftRotate(Node<KeyType> *node){
  Node<KeyType> *right = node->rightChild;
  node->rightChild = right->leftChild;
  if(right->leftChild != nil){
    right->leftChild->parent = node;
  }
  right->parent = node->parent;
  if(node->parent == nil){
    root = right;
  }
  else if(node == node->parent->leftChild){
    node->parent->leftChild = right;
  }
  else{
    node->parent->rightChild = right;
  }
  right->leftChild = node;
  node->parent = right;
}
//=====================================
// rightRotate
// pre condition: Takes an unbalanced RBT and given node and preforms a right rotation
// post condition: Balances out the RBT to fufill the RBT property
template <class KeyType>
void RedBlackTree<KeyType>::rightRotate(Node<KeyType> *node){
  Node<KeyType> *left = node->leftChild;
  node->leftChild = left->rightChild;
  if(left->rightChild != nil){
    left->rightChild->parent = node;
  }
  left->parent = node->parent;
  if(node->parent == nil){
    root = left;
  }
  else if(node == node->parent->rightChild){
    node->parent->rightChild = left;
  }
  else{
    node->parent->leftChild = left;
  }
  left->rightChild = node;
  node->parent = left;
}
//=====================================
// remove
// pre condition: A RedBlackTree that contains values(else EmptyError) and a value k to remove
// post condition: The RedBlackTree with the value k removed from it
template <class KeyType>
void  RedBlackTree<KeyType>::remove(const KeyType& k){
  if(root == nil){ //can't remove if empty
    throw EmptyError();
  }
  Node<KeyType>* par = nil;
  Node<KeyType>* n = root;
  while (k != *n->key){ //get node to be removed
    par = n;
    if (k < *n->key){
      n = n->leftChild;
```

```
    }else{
      n = n->rightChild;
    }
  }
  if(n->leftChild == nil){
    transplant(n, n->rightChild);
  }else if(n->rightChild == nil){
    transplant(n, n->leftChild);
  }else{
    Node<KeyType>* successor = search(*min(n->rightChild));
    if(successor->parent != n){
      transplant(successor, successor->rightChild);
      successor->rightChild = n->rightChild;
      successor->rightChild->parent = successor;
    }
    transplant(n, successor);
    successor->leftChild = n->leftChild;
    successor->leftChild->parent = successor;
    delete n;
  }
}
//=======================================
// maximum
// pre condition: A RedBlackTree with at least one value
// post condition: The RedBlackTree(unchanged) and returns the maximum key
template <class KeyType>
KeyType*  RedBlackTree<KeyType>::maximum() const{
  if (root == nil){
    throw EmptyError();
  }
  Node<KeyType>* n = root;
  while(n->rightChild != nil){
    n = n->rightChild;
  }
  return n->key;
}
//=======================================
// max (private)
// pre condition: A RedBlackTree with at least one value
// post condition: The RedBlackTree and returns the minimum value
template <class KeyType>
KeyType*  RedBlackTree<KeyType>::min(Node<KeyType>* node) const{
  if (node == nil){
    throw EmptyError();
  }
  Node<KeyType>* n = node;
  while(n->leftChild != nil){
    n = n->leftChild;
  }
  return n->key;
}
//=======================================
// minimum
// pre condition: A RedBlackTree with at least one value
// post condition: The RedBlackTree and returns the minimum value
template <class KeyType>
KeyType*  RedBlackTree<KeyType>::minimum()const{
  if (root == nil){ //root shouldn't be NULL either
    throw EmptyError();
  }
  Node<KeyType>* n = root;
  while(n->leftChild != nil){
    n = n->leftChild;
```

```
  }
  return n->key;
}
//======================================
// min (private)
// pre condition: A RedBlackTree with at least one value
// post condition: The RedBlackTree and returns the minimum value
template <class KeyType>
KeyType*  RedBlackTree<KeyType>::max(Node<KeyType> *node) const{
  if (node == nil){
    throw EmptyError();
  }
  Node<KeyType>* n = node;
  while(n->rightChild != nil){
    n = n->rightChild;
  }
  return n->key;
}
//======================================
// successor
// pre condition: A RedBlackTree with at least one value
// post condition: The RedBlackTree(uchanged) and returns the successor
template <class KeyType>
KeyType*  RedBlackTree<KeyType>::successor(const KeyType& k){
  if (root == nil){
    throw EmptyError();
  }
  if(k == *maximum()){ // max has no successor
    return NULL;
  }
  Node<KeyType>* n = search(k); //get the node whose value is k
  if(n == nil){
    return NULL;
  }
  Node<KeyType> *nRC = n->rightChild;
  if(nRC != nil){
    return (min(nRC)); //make a private one now;
  }
  Node<KeyType>* par = n->parent;
  while(par != nil && n == par->rightChild){
    n = par;
    par = par->parent;
  }
  return par->key;
}

//======================================
// predecessor
// pre condition: A RedBlackTree with at least one value
// post condition: The RedBlackTree(uchanged) and returns the predecessor
template <class KeyType>
KeyType*  RedBlackTree<KeyType>::predecessor(const KeyType& k){
  if (root == nil){
    throw EmptyError();
  }
  if(k == *minimum()){ //min has no predecessor
    return NULL;
  }
  Node<KeyType>* n = search(k); //get the node whose value is k
  if(n == nil){
    return NULL;
  }
  Node<KeyType> *nLC = n->leftChild;
```

```
  if(nLC != nil){
    return (max(nLC));
  }
  Node<KeyType>* par = n->parent;
  while(par != nil && n == par->leftChild){
    n = par;
    par = par->parent;
  }
  return par->key;
}

//======================================
//  assignment operator
// pre condition: A RedBlackTree to set equal to another passed in tree rbt
// post condition: The RedBlackTree which is equal to the passed in tree rbt
template <class KeyType>
RedBlackTree<KeyType>&  RedBlackTree<KeyType>::operator=(const RedBlackTree<KeyType> &rbt){
  destroy(root);
  Node<KeyType>* traverse = rbt.root;
  root = copy(traverse, nil, rbt.nil);
}

//======================================
//  search
// pre condition: A RedBlackTree and a value k to search for
// post condition: The RedBlackTree(unchanged) and returns the node with the value k
template <class KeyType>
Node<KeyType>*  RedBlackTree<KeyType>::search(const KeyType& k){
  Node<KeyType>* n = root;
  while(n != nil && *n->key != k){
    if(!(*n->key < k) && (*n->key != k)){
      n = n->leftChild;
    }else{
      n = n->rightChild;
    }
  }
  return n;
}

//======================================
//  transplant
// pre condition: A RedBlackTree and two nodes within that tree
// post condition: The RedBlackTree where the two nodes have swapped places
template <class KeyType>
void  RedBlackTree<KeyType>::transplant(Node<KeyType>* rem, Node<KeyType>* rep){
  if(rem->parent == nil){
    root = rep;
  }else if(rem == rem->parent->leftChild){
    rem->parent->leftChild = rep;
  }else{
    rem->parent->rightChild = rep;
  }
  if(rep != nil){
    rep->parent = rem->parent;
  }
}

//===========================================
// these three functions call the ones below
// this avoids exposing node pointers
template <class KeyType>
std::string  RedBlackTree<KeyType>::inOrder(){
  std::string tree = "{";
```

```
  return inOrd(root, tree) + " }";
}
template <class KeyType>
std::string  RedBlackTree<KeyType>::preOrder(){
  std::string tree = "{";
  return preOrd(root, tree) + " }";
}
template <class KeyType>
std::string  RedBlackTree<KeyType>::postOrder(){
  std::string tree = "{";
  return postOrd(root, tree) + " }";
}

//======================================
// inOrd
// pre condition: A pointer node and a string tree
// post condition: Only returns the string tree which contains the keys of the nodes inOrd
template <class KeyType>
string  RedBlackTree<KeyType>::inOrd(Node<KeyType> *node, string& tree)const {
  if (node == NULL){
    cout << "node = NULL" << endl;
  }
  if (node != nil){
    inOrd(node->leftChild, tree);
    tree += (" " + node->toString());
    inOrd(node->rightChild, tree);
  }
  return tree;
}

//======================================
// preOrd
// pre condition: A pointer node and a string tree
// post condition: Only returns the string tree which contains the keys of the nodes preOrd
template <class KeyType>
string  RedBlackTree<KeyType>::preOrd(Node<KeyType> *node, std::string& tree)const{
  if (node != nil){
    tree += (" " + node->toString());
    inOrd(node->leftChild, tree);
    inOrd(node->rightChild, tree);
  }
  return tree;
}

//======================================
// postOrd
// pre condition: A pointer node and a string tree
// post condition: Only returns the string tree which contains the keys of the nodes postOrd
template <class KeyType>
string  RedBlackTree<KeyType>::postOrd(Node<KeyType> *node, std::string& tree)const{
  if (node != nil){
    inOrd(node->leftChild, tree);
    inOrd(node->rightChild, tree);
    tree += (" " + node->toString());
  }
  return tree;
}

//======================================
// copy
// pre condition: A RedBlackTree and a node that is set to the root of the tree to copy
// post condition: The RedBlackTree that is the copy of the passed in tree
template <class KeyType>
```

```
Node<KeyType>* RedBlackTree<KeyType>::copy(Node<KeyType>* traverse, Node<KeyType>* parent,  No
de<KeyType>* NIL){
  if(traverse == NIL){ //assuming we don’t need to check if traverse is NULL?
    return nil;
  }
  Node<KeyType> *n = new Node<KeyType>;
  n->key = traverse->key;
  n->color = traverse->color;
  n->parent = parent;
  n->leftChild = copy(traverse->leftChild, traverse, NIL);
  n->rightChild = copy(traverse->rightChild, traverse, NIL);
  return n;
}

//=====================================
// destroy
// pre condition: A RedBlackTree and a node to the root of that tree
// post condition: Deallocates the memory of the RedBlackTree
template <class KeyType>
void RedBlackTree<KeyType>::destroy(Node<KeyType>* traverse){
  if(traverse == nil){
    return;
  }
  destroy(traverse->leftChild);
  destroy(traverse->rightChild);
  delete traverse; //get to bottom of left or right subtree, delete, recursive call
}
```

```
// Ben Liepert & Gezim Saciri
// rbt.h
#ifndef RBT_H
#define RBT_H

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

template <class KeyType>
class RedBlackTree
{
  public:
    RedBlackTree();
    ~RedBlackTree();
    RedBlackTree(const RedBlackTree<KeyType>& rbt);

    bool                empty() const { return root == nil; } // return true if empty; false o/
w
    KeyType *           get(const KeyType& k); // return first element with key equal to k
    void                insert(KeyType *k); // insert k into the tree
    void                remove(const KeyType& k); // delete first element with key equal to k
    KeyType *           maximum() const; // return the maximum element
    KeyType *           minimum() const; // return the minimum element
    KeyType *           successor(const KeyType& k); // return the successor of k
    KeyType *           predecessor(const KeyType& k); // return the predecessor of k

    std::string         inOrder(); // return string of elements from an inorder traversal
    std::string         preOrder(); // return string of elements from a preorder traversal
    std::string         postOrder(); // return string of elements from a postorder traversal

    RedBlackTree&  operator=(const RedBlackTree<KeyType> &rbt);

  private:
    Node<KeyType>* root; //bst’s root
    Node<KeyType>* nil; //nil pointer
    std::string         inOrd(Node<KeyType> *node, std::string &tree) const;
    std::string         preOrd(Node<KeyType> *node, std::string &tree) const;
    std::string         postOrd(Node<KeyType> *node, std::string &tree) const;

    Node<KeyType>*      search(const KeyType& k); //find a ptr to the node whose key = k
    void                transplant(Node<KeyType>* rem, Node<KeyType>* rep); //variables in func
tion call subject to change
    KeyType *           min(Node<KeyType> *node) const; // return the minimum element
    KeyType *           max(Node<KeyType> *node) const; // return the maximum element
    void                insertFixup(Node<KeyType> *node);
    void                leftRotate(Node<KeyType> *node);
    void                rightRotate(Node<KeyType> *node);

    Node<KeyType>*      copy(Node<KeyType>* traverse, Node<KeyType>* parent, Node<KeyType>* NIL
); //creates a copy of a rbt with the root of another bst
    void                destroy(Node<KeyType>* node);
};

class EmptyError { };

#include "rbt.cpp"

#endif
```

```
//Gezim Saciri & Ben Liepert
//test file for the Dictionary class
#include <iostream>
#include <sstream>
#include "dict.h"
#include <cassert>
using namespace std;

void test_empty(){
  Dictionary<string> d;
  assert(d.empty());
  string a = "alpaca";
  d.insert(&a);
  assert(!d.empty());
  d.remove(a);
  assert(d.empty());
}

void test_get(){
  Dictionary<string> d;
  assert(d.empty());
  string a = "hi my name is Drew Bapst";
  d.insert(&a);
  assert(d.get(a) == &a);
  string b = "hi my name is Drew Bapst";
  assert(d.get(b) == &a);
  d.remove(a);
  assert(d.get(b) == NULL);
  assert(d.get(a) == NULL);
}

void test_insert(){
  Dictionary<string> d;
  assert(d.empty());
  string a = "alphabet";
  string b = "barnacle";
  d.insert(&a);
  assert(!d.empty());
  d.insert(&b);
}

void test_remove(){
  Dictionary<string> d;
  assert(d.empty());
  string a = "skert";
  d.insert(&a);
  string b = "drewbapst";
  d.insert(&b);
  string c = "a";
  string d2 = "gezim";
  string e = "mami";
  d.insert(&c);
  d.insert(&d2);
  d.insert(&e);
  d.remove(c);
  assert(d.get(c) == NULL);
  d.remove(d2);
  assert(d.get(d2) == NULL);
  d.remove(b);
  assert(d.get(b) == NULL);
  d.remove(a);
  assert(d.get(a) == NULL);
  d.remove(e);
```

```
  assert(d.empty());
  assert(d.get(e) == NULL);
}

int main(){
  test_empty();
  test_get();
  test_insert();
  test_remove();
  cout << "Dictionary: All Tests Passed!\n";
  return 0;
}
```

```
//Ben Liepert & Gezim Saciri
//testing file for the movie class
#include <iostream>
#include <cassert>
#include "movie.h"
using namespace std;

void test_default(){
  Movie a;
  Movie b;
  Movie c;
  assert(a.title == "Drew Bapst: The Musical");
  assert(a.cast == "Drew Bapst  Josh Blaz  Brian Tran");
  assert(b.title == "Drew Bapst: The Musical");
  assert(b.cast == "Drew Bapst  Josh Blaz  Brian Tran");
  assert(c.title == "Drew Bapst: The Musical");
  assert(c.cast == "Drew Bapst  Josh Blaz  Brian Tran");
}

void test_copy(){
  Movie a;
  a.title = "Alphabet Soup 4000";
  a.cast = "Teddy Bear";
  Movie b(a);
  assert(b.title == "Alphabet Soup 4000");
  assert(b.cast == "Teddy Bear");
}

void test_init(){
  string t = "Skert: Episode III: Revenge of Young Thug";
  string c = "Young Thug  Yo Yo Ma";
  Movie a(t,c);
  assert(a.title == "Skert: Episode III: Revenge of Young Thug");
  assert(a.cast == "Young Thug  Yo Yo Ma");
  t = "Bro it’s a prank";
  assert(a.title == "Skert: Episode III: Revenge of Young Thug");
  c = "So Flo Antonio";
  assert(a.cast == "Young Thug  Yo Yo Ma");
}

void test_lessThan(){
  Movie a;
  Movie b;
  a.title = "Hello Its Me, Drew Bapst";
  b.title = "Zebra Invasion IV: Revenge of the Serengeti Stripes";
  assert(a < b);
  assert(!(b < a));
}

void test_toString(){
  Movie a;
  Movie b;
  a.title = "Mr. Blaz’s Day on the Tundra";
  a.cast = "Josh Blaz";
  b.title = "Need for speed: skert skert";
  b.cast = "Drew Bapst";
  assert(a.toString() == "(Mr. Blaz’s Day on the Tundra:: Josh Blaz)");
  assert(b.toString() == "(Need for speed: skert skert:: Drew Bapst)");
}

int main(){
  test_default();
  test_init();
```

```
  test_copy();
  test_lessThan();
  test_toString();
  cout << "Movie: All Tests Passed!\n";
  return 0;
}
```

```
//Gezim Saciri & Ben Liepert
//test file for node class
#include <iostream>
#include <string>
#include <cassert>
#include "node.h"
using namespace std;

void test_constructor()
{
  Node<int> A;
  cout << A << endl;
  //assert(A.toString() == "( )");
  cout << "Constructor passed!\n";
}

void test_copyConstructor()
{
  Node<int> A;
  int x = 5;
  A.key = &x;
  assert(A.toString() == "(5)");


  Node<int> B(A);
  assert(B.toString() == "(5)");
  cout << "Copy constructor passed!\n";
}

void test_lessThanOp()
{
  Node<int> A;
  int x = 5;
  A.key = &x;
  assert(A.toString() == "(5)");


  Node<int> B(A);
  assert(B.toString() == "(5)");
  int y = 3;
  B.key = &y;
  assert(B.toString() == "(3)");

  assert(B < A);
  assert(!(A < B));
  cout << "Less than operator passed!\n";
}

int main()
{
    test_constructor();
    //test_copyConstructor();
    //test_lessThanOp();
    cout << "All Tests Passed!\n";
}
```

```
//Gezim Saciri & Ben Liepert
//test file for RedBlackTree class
#include <iostream>
#include <sstream>
#include "rbt.h"
#include <cassert>
using namespace std;

void test_empty(){
  RedBlackTree<int> A;
  assert(A.empty());
  int a = 5;
  A.insert(&a);
  assert(!A.empty());
  int b = 77;
  A.insert(&b);
  assert(!A.empty());
}

void test_copy(){
  RedBlackTree<int> A;
  assert(A.empty());
  int a = 5, b = 3, c = 4, d = 7, e = 6, f = 9, j = 2, k = 1;
  A.insert(&a);
  A.insert(&b);
  A.insert(&c);
  A.insert(&d);
  A.insert(&e);
  A.insert(&f);
  A.insert(&j);
  A.insert(&k);
  RedBlackTree<int> B(A);
  //cout << "A.inOrder() = \n" << A.inOrder() << endl;
  //cout << "A.preOrder() = \n" << A.preOrder() << endl;
  //cout << "A.postOrder() = \n" << A.postOrder() << endl;
  assert(A.inOrder() == B.inOrder());
  assert(A.preOrder() == B.preOrder());
  assert(A.postOrder() == B.postOrder());
}

void test_get(){
  RedBlackTree<float> A;
  assert(A.empty());
  float a = 1.3;
  float b = 2.7;
  float c = 2.959;
  float d = 155.66;
  A.insert(&a);
  A.insert(&d);
  float i = 10.73;
  float j = 10.73;
  A.insert(&i);
  float* k = A.get(j);
  assert(&i == k);
  A.insert(&b);
  A.insert(&c);
  float l = -11.373;
  float m = -11.373;
  A.insert(&l);
  float *o = A.get(m);
  assert(&l == o);
}
```

```
void test_insert(){
  RedBlackTree<int> A;
  assert(A.empty());
  int i = 18;
  A.insert(&i);
  assert(!A.empty());
  int j = 50002;
  A.insert(&j);
  assert(!A.empty());
}

void test_remove(){
  RedBlackTree<int> A;
  int a = 1;
  A.insert(&a);
  A.remove(a);
  assert(A.empty());
  assert(A.inOrder() == "{ }");
  assert(A.preOrder() == "{ }");
  assert(A.postOrder() == "{ }");
  int b = 2;
  A.insert(&b);
  int c = 3;
  A.insert(&c);
  int d = 4;
  A.insert(&d);
  A.remove(d);
  assert(A.inOrder() == "{ (2) (3) }");
  int e = 5;
  A.insert(&e);
  int f = 6;
  A.insert(&f);
  int g = 7;
  A.insert(&g);
  int j = 8;
  A.insert(&j);
  A.remove(e);
  assert(A.get(e) == NULL);
  RedBlackTree<int> B;
  int h = 2;
  try{
    B.remove(h);
    assert(false);
  }catch(EmptyError e){
    //do nothing
  }
  int k = 10;
  int l = 3;
  int m = -8;
  int n = 16;
  int o = 2;
  int p = 4;
  int q = 17;
  B.insert(&p);
  B.insert(&o);
  B.insert(&l);
  B.insert(&m);
  B.insert(&n);
  B.insert(&k);
  B.insert(&q);
  B.remove(q);
  assert(B.get(q) == NULL);
  B.remove(p);
```

```
  assert(B.get(p) == NULL);
  B.remove(m);
  B.remove(l);
  B.remove(k);
  B.remove(n);
  B.remove(o);
  assert(B.empty());
}

void test_maximum(){
  RedBlackTree<int> A;
  assert(A.empty());
  int i,j,k,l,m;
  i = 15;
  A.insert(&i);
  assert(!A.empty());
  assert(*A.maximum() == 15);
  j = -1;
  A.insert(&j);
  assert(*A.maximum() == 15);
  k = 16;
  A.insert(&k);
  assert(*A.maximum() == 16);
  l = -150;
  A.insert(&l);
  assert(*A.maximum() == 16);
  m = 1600;
  A.insert(&m);
  assert(*A.maximum() == 1600);
  assert(!A.empty());
}

void test_minimum(){
  RedBlackTree<int> A;
  assert(A.empty());
  int i,j,k,l,m;
  i = 15;
  A.insert(&i);
  assert(!A.empty());
  assert(*A.minimum() == 15);
  j = -1;
  A.insert(&j);
  assert(*A.minimum() == -1);
  k = 16;
  A.insert(&k);
  assert(*A.minimum() == -1);
  l = -150;
  A.insert(&l);
  assert(*A.minimum() == -150);
  m = 1600;
  A.insert(&m);
  assert(*A.minimum() == -150);
  assert(!A.empty());
}

void test_successor(){
  RedBlackTree<int> A;
  int a = 5;
  A.insert(&a);
  int b = 2;
  A.insert(&b);
  int c = 8;
  A.insert(&c);
```

```
  assert(A.successor(c) == NULL);
  assert(A.successor(a) == &c);
  int d = 4;
  A.insert(&d);
  int e = 1;
  A.insert(&e);
  int f = 6;
  A.insert(&f);
  assert(A.successor(e) == &b);
  assert(A.successor(c) == NULL);
  assert(A.successor(f) == &c);
  assert(A.successor(b) == &d);
  int g = 7;
  A.insert(&g);
  int j = 3;
  A.insert(&j);
  assert(A.successor(d) == &a);
}

void test_predecessor(){
  RedBlackTree<int> A;
  int a = 5;
  A.insert(&a);
  int b = 2;
  A.insert(&b);
  int c = 8;
  A.insert(&c);
  assert(A.predecessor(c) == &a);
  assert(A.predecessor(b) == NULL);
  int d = 4;
  A.insert(&d);
  int e = 1;
  A.insert(&e);
  int f = 6;
  A.insert(&f);
  assert(A.predecessor(f) == &a);
  assert(A.predecessor(e) == NULL);
  int g = 7;
  A.insert(&g);
  int j = 3;
  A.insert(&j);
  assert(A.predecessor(g) == &f);
  assert(A.predecessor(a) == &d);
}

void test_inOrder(){
  RedBlackTree<int> A;
  assert(A.inOrder() == "{ }");
  assert(A.empty());
  int i = 999;
  A.insert(&i);
  assert(A.inOrder() == "{ (999, BLACK) }");
  int j = -15;
  A.insert(&j);
  assert(A.inOrder() == "{ (-15, RED) (999, BLACK) }");
  assert(!A.empty());
  int k = 1055;
  A.insert(&k);
  assert(A.inOrder() == "{ (-15, RED) (999, BLACK) (1055, RED) }");
}

void test_preOrder(){
  RedBlackTree<int> A;
```

```
  assert(A.preOrder() == "{ }");
  assert(A.empty());
  int i = 999;
  A.insert(&i);
  assert(A.preOrder() == "{ (999, BLACK) }");
  int j = -15;
  A.insert(&j);
  assert(A.preOrder() == "{ (999, BLACK) (-15, RED) }");
  assert(!A.empty());
  int k = 1055;
  A.insert(&k);
  assert(A.preOrder() == "{ (999, BLACK) (-15, RED) (1055, RED) }");
}

void test_postOrder(){
  RedBlackTree<int> A;
  assert(A.postOrder() == "{ }");
  assert(A.empty());
  int i = 999;
  A.insert(&i);
  assert(A.postOrder() == "{ (999, BLACK) }");
  int j = -15;
  A.insert(&j);
  assert(A.postOrder() == "{ (-15, RED) (999, BLACK) }");
  assert(!A.empty());
  int k = 1055;
  A.insert(&k);
  assert(A.postOrder() == "{ (-15, RED) (1055, RED) (999, BLACK) }");
}

void test_assignment(){
  RedBlackTree<int> A;
  assert(A.empty());
  int a = 5, b = 3, c = 4, d = 7, e = 6, f = 9, j = 2, k = 1;
  A.insert(&a);
  A.insert(&b);
  A.insert(&c);
  A.insert(&d);
  A.insert(&e);
  A.insert(&f);
  A.insert(&j);
  A.insert(&k);
  RedBlackTree<int> B; //can't just do = A here;
  int l = 150, m = -15, n = 77;
  B.insert(&l);
  B.insert(&m);
  B.insert(&n);
  B = A;
  assert(A.inOrder() == B.inOrder());
  assert(A.preOrder() == B.preOrder());
  assert(A.postOrder() == B.postOrder());
}

void test_question1(){
  RedBlackTree<int> A;
  assert(A.empty());
  int a = 41, b = 38, c = 31, d = 12, e = 19, f = 8;
  A.insert(&a);
  A.insert(&b);
  A.insert(&c);
  A.insert(&d);
  A.insert(&e);
  A.insert(&f);
```

```
  cout << A.preOrder() << endl;
}

int main(){

  test_copy();
  test_assignment();
  test_empty();
  test_get();
  test_insert();
  //test_remove();
  test_maximum();
  test_minimum();
  test_successor();
  test_predecessor();
  test_inOrder();
  test_preOrder();
  test_postOrder();
  test_question1();
  cout << "Red Black Tree: All Tests Passed!\n";

  return 0;
}
```