CSCI-1200 Data Structures — Spring 2019
Test 3 — Thursday, April 4th 6-7:50pm

Xinhao Luo	luox6@rpi.edu lab section: 11
room: Darrin 337 zone: BROWN row: 4 seat: 1	6-7:50pm

Write the names of the undergraduate mentors for your lab sorted by height, from tallest to shortest.

Trevor Eli George Anshul

- This exam has 5 problems worth a total of 100 points (including the cover sheet).
- This packet contains 11 pages of problems numbered 1-11. Please count the pages of your exam and raise your hand if you are missing a page.
- The packet contains 1 blank pages. If you use a blank page to solve a problem, make a note in the original box and clearly label which problem you are solving on the blank page.
- This test is closed-book and closed-notes except for the .pdf notes you (optionally) uploaded to Submitty by last night. These notes are the last 2 pages of your exam packet.
- **DO NOT REMOVE THE STAPLE OR SEPARATE THE PAGES OF YOUR EXAM. DOING SO WILL RESULT IN A -10 POINT PENALTY!**
- You may have pencils, eraser, pen, tissues, water, and your RPI ID card on your desk. Place everything else on the floor under your chair. Electronic equipment, including computers, cell phones, calculators, music players, smart watches, cameras, etc. is not permitted and must be turned "off" (not just vibrate).
- Please read each question carefully. Raise your hand if you have a question.
- Please state clearly any assumptions that you made in interpreting a question. Unless otherwise stated you may use any technique that we have discussed in lecture, lab, or on the homework.
- Please write neatly. If we can't read your solution, we can't give you full credit for your work.
- You do not need to write `#include` statements for STL libraries. Writing `std::` is optional.

Xinhao Luo luox6@rpi.edu

1 Short Recursion Problems [/ 18]

1.1 Donut Boxes [/ 10]

In this problem you are given a vector of strings representing donut flavors. You can assume they are unique and that the vector is not empty. Fill in the call in the driver function, and then implement your function, which should be recursive and returns a vector of all possible ways to arrange a box with one of each flavor of donut.

```
typedef std::vector<std::string> DonutBox;
DonutBox donuts;
std::vector<DonutBox> boxes;
donuts.push_back("strawberry"); donuts.push_back("chocolate"); donuts.push_back("maple");
findBoxes(donuts, boxes);
printDonutBoxes(boxes);
```

Gives the output:

```
Printing 6 boxes:
Box has: strawberry chocolate maple
Box has: strawberry maple chocolate
Box has: chocolate strawberry maple
Box has: chocolate maple strawberry
Box has: maple strawberry chocolate
Box has: maple chocolate strawberry
```

```
void findBoxes(const DonutBox& box, DonutBox& current_box, std::vector<DonutBox>& boxes){
```

```
if (box.empty()) {
    boxes.push_back(current_box); return;
}
for (int i = 0; i < box.size(); i++) {
    DonutBox c = current_box;
    DonutBox b = box;
    c.push_back(b[i]);
    b.erase(i);
    findBoxes(b, c, boxes);
}
```

sample solution: 11 line(s) of code

```
}
void findBoxes(const DonutBox& box, std::vector<DonutBox>& boxes){
    DonutBox tmp;
    findBoxes( box, tmp, boxes );
}
```

1.2 String Reversal [/ 8]

In this problem you will write *reverse()*, which should do an in-place reversal of a string. You cannot use any for/while loops and you can only use `#include<iostream>` and `#include<string>`. You cannot declare any additional strings, i.e. the reverse operation must happen in-place. You may write helper functions as long as they follow the same rules.

```
std::string a = "ThisIsEven";
std::string b = "ThisIsOdd";
std::string c(a);
std::string d(b);

reverse(c);
reverse(d);

std::cout << a << " reversed is " << c << std::endl;
std::cout << b << " reversed is " << d << std::endl;
```

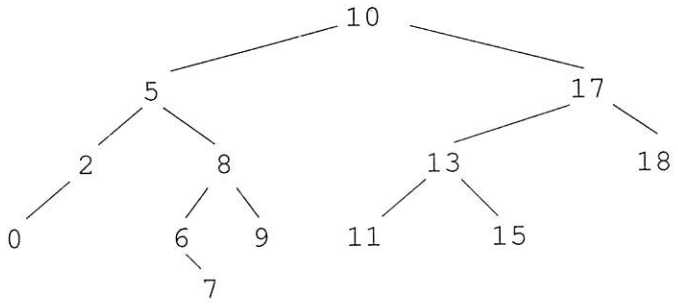
Outputs:

```
ThisIsEven reversed is nevEsIsihT
ThisIsOdd reversed is ddOsIsihT
```

```
void reverse(string& str, int start, int end)
    if (start >= end) {
        return;
    }
    str += str[start];
    str[start] = str[end];
    str[end] = str[str.size()-1];
    str = str.substring(0, str.size()-1);
    reverse(str, start+1, end-1);
}
void reverse(string& str) {
    reverse(str, 0, str.size()-1);
}
```

sample solution: 15 line(s) of code

2 Tree Drawings [/ 17]



2.1 Tree Construction [/5]

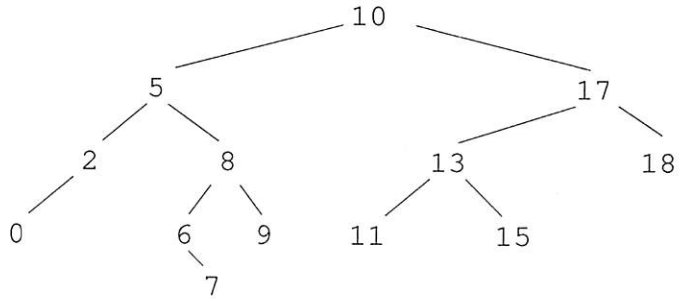
Write a list of calls to *insert()* in order that would create the binary search tree shown above.

insert(10);
insert(17);
insert(5);
insert(2);
insert(0);
insert(8);
insert(6);
insert(9);
insert(7);
insert(13);
insert(18);
insert(11);
insert(15);

2.2 Post-order Traversal [/6]

Next, write the post-order traversal of this tree:

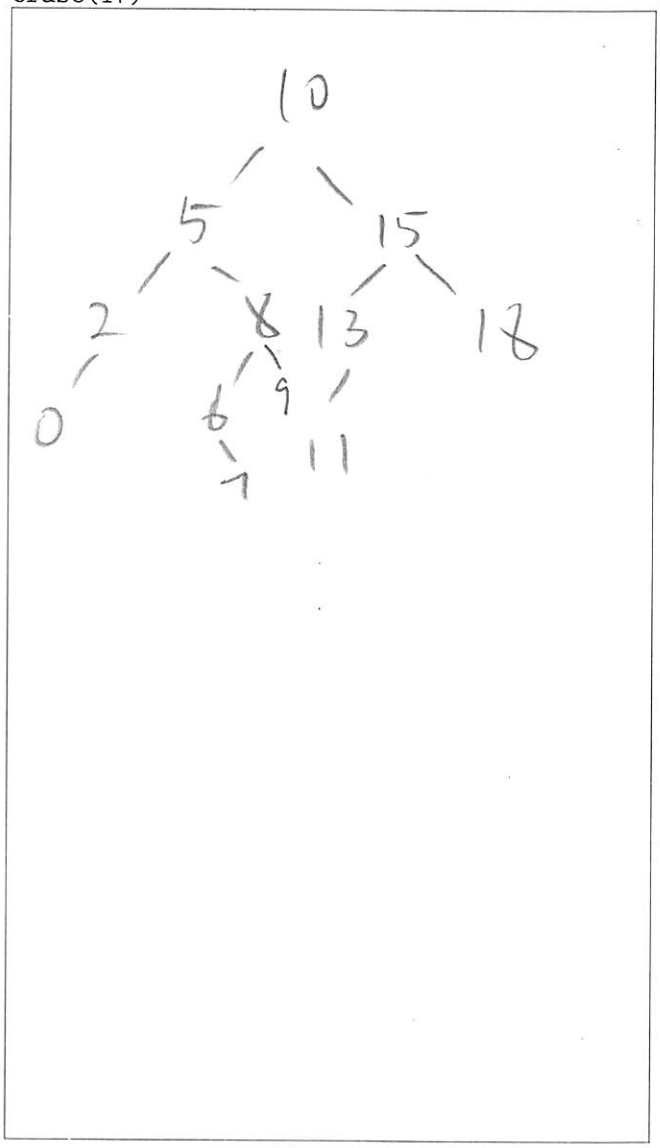
0 2 7 6 9 8 5 10 11 15 13 18 17



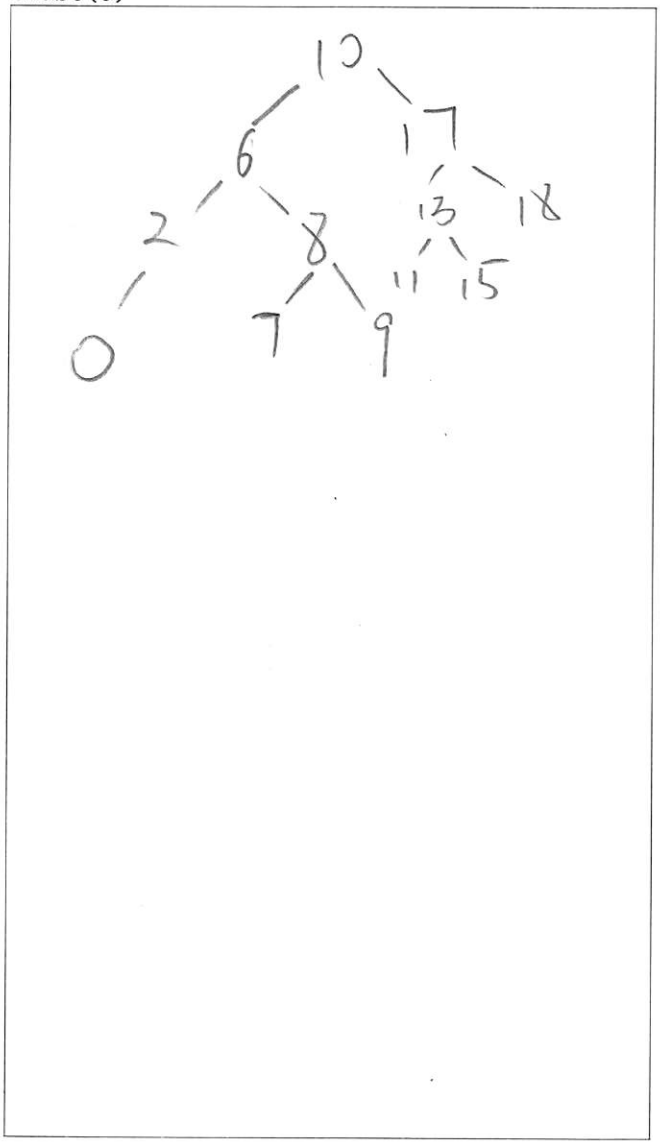
2.3 Erasing From Trees [/6]

Draw the resulting trees from the following `erase` calls. If you need to choose between the left and right subtree, choose the right subtree. Assume this is two independent calls to the binary search tree shown above, in other words each solution should have 12 nodes.

`erase(17)`



`erase(5)`



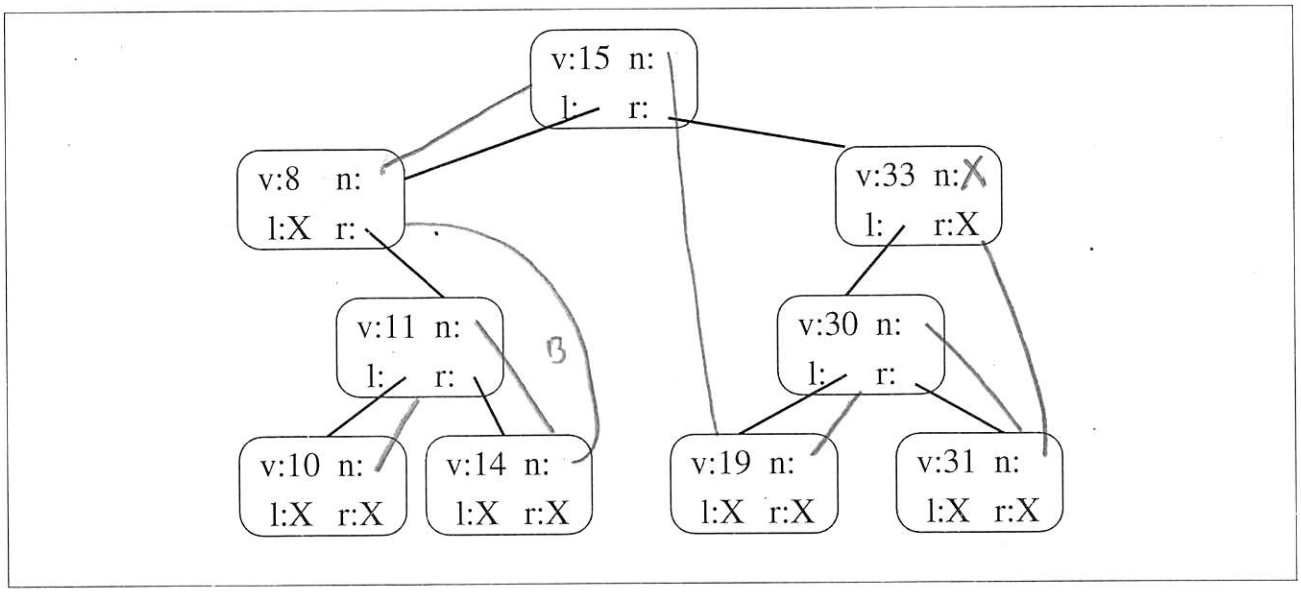
3 Linked Trees [/ 25]

Ben Bitdiddle, Data Structures extraordinaire, is concerned about the running time of his binary search tree iterator's increment operation. Fearing that it might have to go through many nodes, he wants a data structure that makes his operator++() easy to write, an invention he calls "Linked Trees." The modified implementation of TreeNode is below. The new *next* member variable points to the next in-order node for the tree.

```
class TreeNode{
    TreeNode(int init): value(init), left(NULL), right(NULL),
    next(NULL) {}
    int value;
    TreeNode* left;
    TreeNode* right;
    TreeNode* next;
};
```

3.1 Visualizing the Tree [/ 5]

First, for the BST below, draw in all the *next* pointers to make it a "linked tree". Use a **X** to denote a NULL pointer, and try to avoid crossing lines:



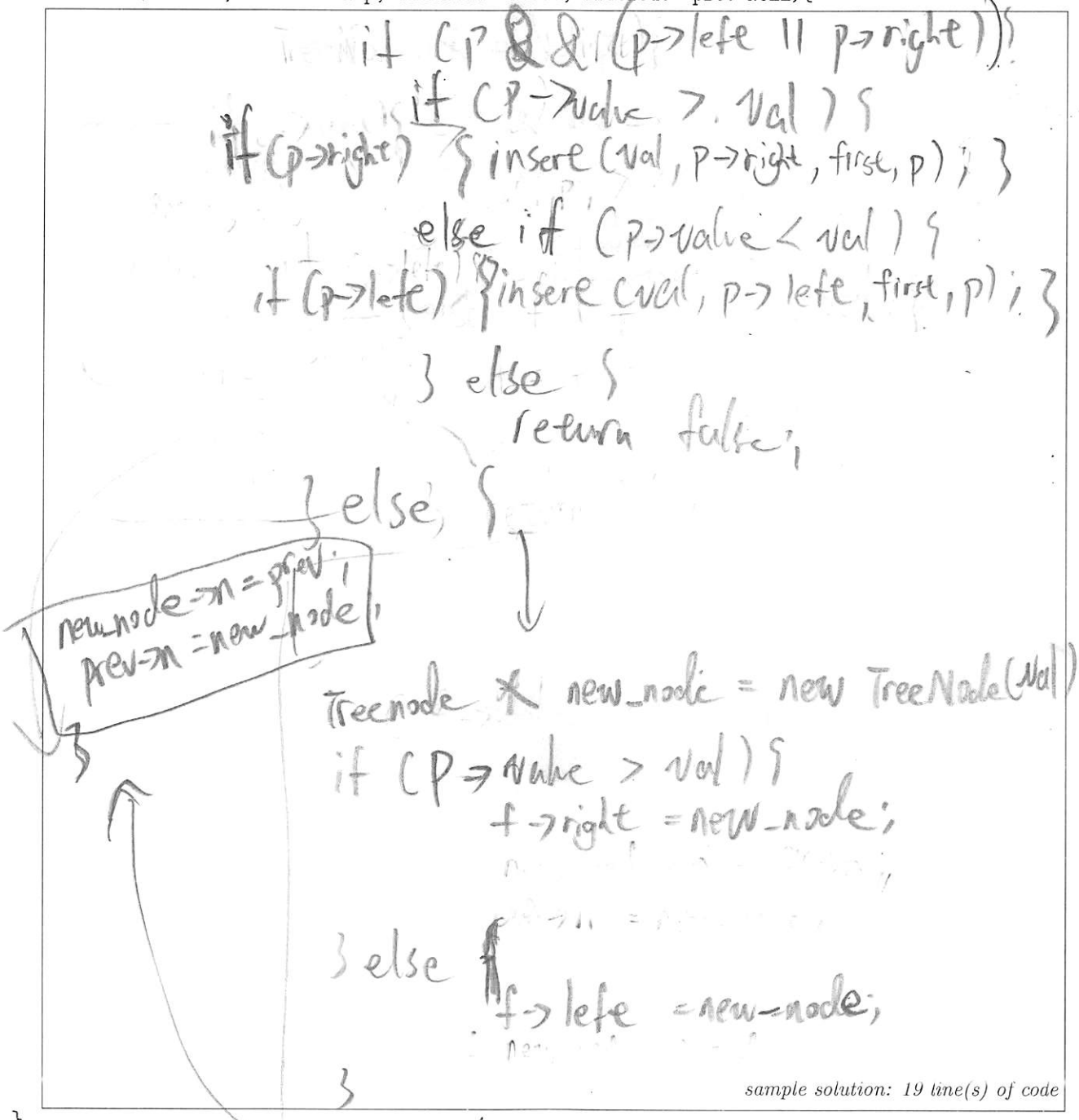
With Ben's addition, what will be *linked_tree_iterator::operator++()*'s worst case running time, assuming n nodes in the tree and a maximum height of h?

O(1)

3.2 Writing *insert()* [/ 20]

Implement *insert()* for the linked tree. Similarly to the insert we've done in labs, your function should take in a value to insert and a root pointer that represents the root of a binary search tree. To simplify things, your function should simply return a boolean indicating if the element was added to the tree. You cannot use *linked_tree_iterator::operator--*. Make sure that *insert()* still runs in ~~$O(h)$~~ time, where h is the height of the tree.

```
bool insert(int val, TreeNode*&p){
    TreeNode* start = p;
    while(start && start->left) { start = start->left; }
    return insert(val,p,start);
}

//prev will have a default value of NULL if we only pass in 3 args
bool insert(int val, TreeNode*& p, TreeNode* first, TreeNode* prev=NULL){
```



sample solution: 19 line(s) of code

```
}
```

4 Actor Voting [/ 22]

In this problem you will fill in the blanks in a function *bestActors()*, which takes in an STL `set` of strings which are actor names, and an STL `vector` of strings, where every entry in the vector is one vote for an actor. The three actors with the highest numbers of votes should be printed, ordered by the number of votes. The worst actor (the one with the smallest number of votes) should also be printed. Ties should be resolved by choosing the name that comes first alphabetically. Assume that there will be at least 4 actors. **See the next page for instructions on answering the question.**

As an example, if there were 2 votes for "KevinBacon", 3 votes for "OwenWilson", 2 votes for "LukeWilson", 1 vote for "JackBlack", and 5 votes for "MarilynMonroe", the output should be:

```
Actor #1: MarilynMonroe
Actor #2: OwenWilson
Actor #3: KevinBacon
Worst actor: JackBlack
```

```
typedef /***FILL IN #1***/ TallyType;
typedef /***FILL IN #2***/ DataType;

void bestActors(/***FILL IN #3***/ actors, /***FILL IN #4***/ votes) {
    std::set<std::string>::const_iterator a_itr;
    TallyType tally;
    for (a_itr = actors.begin(); a_itr != actors.end(); a_itr++) {
        /***FILL IN #5***/ ;
    }

    for (int i=0; i<votes.size(); i++) {
        /***FILL IN #6***/ ;
    }

    DataType data;
    TallyType::iterator t_itr;

    for (t_itr = tally.begin(); t_itr != tally.end(); t_itr++) {
        data[ /***FILL IN #7***/ ].insert( /***FILL IN #8***/ );
    }

    DataType::iterator d_itr = /***FILL IN #9***/;
    std::set<std::string>::const_iterator s_itr = /***FILL IN #10***/;
    for(int i=0; i<3; i++){
        std::cout << "Actor #" << i+1 << ": " << *s_itr << std::endl;
        s_itr++;
        if(s_itr == /***FILL IN #11***/){
            d_itr--;
            s_itr = /***FILL IN #12***/;
        }
    }

    d_itr = /***FILL IN #13***/; s_itr = /***FILL IN #14***/;
    std::cout << "Worst actor: " << *s_itr << std::endl;
}
```

4.1 Fill In the Blanks [/ 22]

Using the code bank and empty boxes below, write which letter corresponds to the correct code fragment for each "FILL IN" comment. You cannot use a letter more than once.

A) `map<int, set<string> >`

B) `map<string, int>`

C) `const set<string>&`

D) `const vector<string>&`

E) `tally[votes[i]]++`

F) `tally[*a_itr]++`

G) `t_itr->first`

H) `--data.end()`

I) `--data.end()`

J) `map<int, set<string> >`

K) `map<string, int>`

L) `set<string>`

M) `vector<string>`

N) `tally[votes[i]] = 0`

O) `tally[*a_itr] = 0`

P) `t_itr->second`

Q) `data.end()`

R) `data.end()--`

S) `d_itr->second.begin()`

T) `d_itr->second.begin()`

U) `d_itr->second.begin()`

V) `d_itr->second.end()`

W) `d_itr->second.end()`

X) `d_itr->second.end()`

Y) `data.end()--`

Z) `data.begin()`

FILL IN #1: K

FILL IN #2: A

FILL IN #3: C

FILL IN #4: D

FILL IN #5: O

FILL IN #6: E

FILL IN #7: P

FILL IN #8: G

FILL IN #9: H

FILL IN #10: S

FILL IN #11: V

FILL IN #12: T

FILL IN #13: Z

FILL IN #14: U

5 More About Ropes [/ 15]

For the following problems, we will be working with the Rope from HW8. The answers will not depend on your HW8 solution. As a reminder, the rope `Node` class is defined as follows:

```
class Node{
public:
    Node() : left{NULL}, right{NULL}, parent{NULL}, weight{0} {}
    Node* left;
    Node* right;
    Node* parent;
    int weight;
    std::string value;
};
```

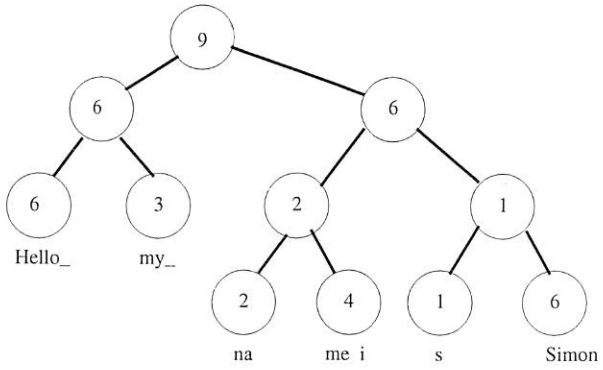
For Problems 5.2 and 5.3, assume that there is a function *Rope::split(i,rhs)* that will leave the first i characters of the string in the current rope, and move the rest to a new Rope *rhs*. Also assume there is a function *Rope::concat_no_copy(rhs)* that will add the nodes from rhs to the current rope, set rhs's root to NULL, and recalculate weights.

5.1 Calculating Rope Length [/ 5]

Assuming we don't have a `Rope` object, write a function called *rope_size()* that takes in a `Node*` called *root* that points to the root of a rope's tree, and returns an `int` which is the length of string the rope represents. Your function should run in $O(h)$ time where h is the height of the tree. There is an example rope on the next page.

```
int rope_size (Node *n) {
    int sum = 0;
    while (n) {
        sum += n->weight;
        n = n->right;
    }
    return sum;
}
```

sample solution: 12 line(s) of code



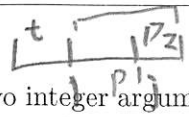
5.2 Writing *insert()* [/ 5]

Write *Rope::insert()*, a void function that takes two arguments. The first is an integer i that specifies which index to insert at, and the second is a `Rope` s to insert. If the example above is in `r1`, calling `r1.insert(13," without a doubt")` would change `r1` to have the string ``Hello my name without a doubt is Simon''. *Hint: You can do this using exactly one call to* split() *and two calls to* concat().

```
void Rope::insert(int i, Rope& s) {
    Rope rhs;                          if (i<0 || i>=size_) {
    this->split(i,rhs);                    return;
    concat(s);                         }
    concat(rhs);
}
```

sample solution: 6 line(s) of code

5.3 Writing *remove()* [/ 5]



Write *Rope::remove()*, a void function that takes two integer arguments i and j which specify the first and last index of a sequence of characters to remove from the rope. If the example pictured above is in `r1`, calling `r1.remove(6,17)` would change `r1` to have the string "Hello Simon". *Hint: You can do this using exactly two calls to* split() *and one call to* concat().

```
void Rope::remove(int i, int j) {
    Rope p1, p2;                       if (i>=size_ ||
    split(i, p1);                          i>j) {
    p1.split(j-i, p2);                     return;
    concat(p2);                        }
}
```

sample solution: 7 line(s) of code

(blank page)

val p first prev.

```
if ( ->value==val ) {
    return false;
}
if ( ->val )
```

String.length()

Xinhao Luo luox6@rpi.edu

Lab Section 11	棕色头发眼镜高个子Trevor
noon-1:50pm	棕色卷发讲话超快Eli
JROWL 2C30	自带谷歌印度小哥Anshul
TA: Yingyi	中国小哥George
mentors: Anshul, Eli, Trevor, & George	

Map in C++ Standard Template Library

begin() - Returns an iterator to the first element in the map
end() - Returns an iterator to the theoretical element that follows last element in the map
size() - Returns the number of elements in the map
max_size() - Returns the maximum number of elements that the map can hold
empty() - Returns whether the map is empty
pair insert(keyvalue, mapvalue) - Adds a new element to the map
erase(iterator position) - Removes the element at the position pointed by the iterator
erase(const g) - Removes the key value 'g' from the map
clear() - Removes all the elements from the map

Sample:

```
// empty map container
map<int, int> gquiz1;

// insert elements in random order
gquiz1.insert(pair<int, int>(1, 40));
gquiz1.insert(pair<int, int>(2, 30));
gquiz1.insert(pair<int, int>(3, 60));
gquiz1.insert(pair<int, int>(4, 20));
gquiz1.insert(pair<int, int>(5, 50));
gquiz1.insert(pair<int, int>(6, 50));
gquiz1.insert(pair<int, int>(7, 10));
```

```
OUTPUTS
A:
1 40
2 30
3 60
4 20
5 50
6 50
7 10

B:
3 60
5 50
6 50
7 10

C:
3 60
4 20
5 50
6 50
7 10
```

```
// A - printing map gquiz1
map<int, int>::iterator itr;
for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr) {
    cout << itr->first << '\t' << itr->second << '\n';
}
cout << endl;

// B - assigning the elements from gquiz1 to gquiz2
map<int, int> gquiz2(gquiz1.begin(), gquiz1.end());

// C - remove all elements up to element with key=3 in gquiz2
gquiz2.erase(gquiz2.begin(), gquiz2.find(3));

// D - remove all elements with key = 4
gquiz2.erase(4);
```

Set in C++ Standard Template Library

begin() - Returns an iterator to the first element in the set.
end() - Returns an iterator to the theoretical element that follows last element in the set.
size() - Returns the number of elements in the set.
max_size() - Returns the maximum number of elements that can hold.
empty() - Returns whether the set is empty.

Sample:

```
// empty set container
set <int, greater <int> > gquiz1;

// insert elements in random order
gquiz1.insert(40);
gquiz1.insert(30);
gquiz1.insert(60);
gquiz1.insert(20);
gquiz1.insert(50);
gquiz1.insert(50); // only one 50 will be added to the set
gquiz1.insert(10);
```

```
OUTPUTS
A: 60 50 40 30 20 10

C: 30 40 50 60

D: 30 40 60
```

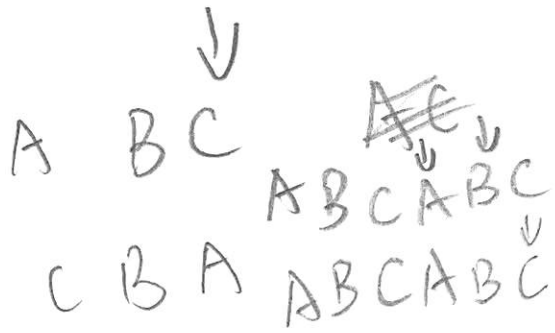
```
// A - printing set gquiz1
set <int, greater <int> > :: iterator itr;
for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr) {
    cout << ' ' << *itr;
}
cout << endl;

// B - assigning the elements from gquiz1 to gquiz2
set <int> gquiz2(gquiz1.begin(), gquiz1.end());

// C - remove all elements up to 30 in gquiz2
gquiz2.erase(gquiz2.begin(), gquiz2.find(30));

// D - remove element with value 50 in gquiz2
gquiz2.erase (50);
```

Big-O Notation

	Average			
	Access	Search	Insert	Delete
Array	1	N	N	N
List	N	N	1	1
HashTable		1	1	1
BinaryST	Log(N)	Log(N)	Log(N)	Log(N)
Maps	Log(N)	Log(N)	Log(N)	Log(N)

	Worst			
	Access	Search	Insert	Delete
Array	1	N	N	N
List	N	N	1	1
HashTable		N	N	N
BinaryST	N	N	N	N
Maps	Log(N)	Log(N)	Log(N)	Log(N)

O(1) - a.k.a. CONSTANT: Number of operations is independent of the size of the problem. e.g., compute quadratic root.
O(log n) - a.k.a. LOGARITHMIC. e.g., dictionary lookup, binary search.
O(n) - a.k.a. LINEAR. e.g., sum up a list.
O(n log n) - e.g., sorting.
O(n^2), O(n^3), O(n^k) - a.k.a. POLYNOMIAL. e.g., find closest pair of points.
O(2^n), O(k^n) - a.k.a. EXPONENTIAL. e.g., Fibonacci, playing chess.



```
#ifndef ds_set_h_
#define ds_set_h_

// ------- ----------------------------------------------------
// TREE NODE CLASS
template < class T>
class TreeNode {
public:
  TreeNode() : left(NULL), right(NULL), parent(NULL) {}
  TreeNode(const T& init) : value(init), left(NULL), right(NULL),
parent(NULL) {}
  T value;
  TreeNode* left;
  TreeNode* right;
  TreeNode* parent;
};

template < class T> class ds_set;

// ------- ----------------------------------------------------
// TREE NODE ITERATOR CLASS
template < class T>
class tree_iterator {
public:
  tree_iterator() : ptr_(NULL), set_(NULL) {}
  tree_iterator(TreeNode<T>* p, const ds_set<T> * s) : ptr_(p), set_(s)
{}
  tree_iterator(const tree_iterator& old) : ptr_(old.ptr_),
set_(old.set_) {}
  ~tree_iterator() {}
  tree_iterator& operator=(const tree_iterator& old) { ptr_ = old.ptr_;
set_ = old.set_; return *this; }
  // operator* gives constant access to the value at the pointer
  const T& operator*() const { return ptr_->value; }
  // compacions operators are straightforward
  friend bool operator== (const tree_iterator& lft, const
tree_iterator& rgt)
  { return (lft.set_ == rgt.set_ && lft.ptr_ == rgt.ptr_); }
  friend bool operator!= (const tree_iterator& lft, const
tree_iterator& rgt)
  { return (lft.set_ != rgt.set_ || lft.ptr_ != rgt.ptr_); }

  // increment & decrement operators
  tree_iterator<T> & operator++() {
    if (ptr_->right != NULL) { // find the leftmost child of the right
node
      ptr_ = ptr_->right;
      while (ptr_->left != NULL) { ptr_ = ptr_->left; }
    } else { // go upwards along right branches... stop after the
first left
      while (ptr_->parent != NULL && ptr_->parent->right == ptr_)
{ ptr_=ptr_->parent; }
      ptr_ = ptr_->parent;
    }
    return *this;
  }
  tree_iterator<T> operator++(int) {
    tree_iterator<T> temp(*this);
    ++(*this);
    return temp;
  }
  tree_iterator<T> & operator--() {
    if (ptr_ == NULL) { // so that it works for end()
      assert (set_ != NULL);
      ptr_ = set_->root_;
      while (ptr_->right != NULL) { ptr_ = ptr_->right; }
    } else if (ptr_->left != NULL) { // find the rightmost child of the
left node
      ptr_ = ptr_->left;
      while (ptr_->right != NULL) { ptr_ = ptr_->right; }
    } else { // go upwards along left branches... stop after the first
right
      while (ptr_->parent != NULL && ptr_->parent->left == ptr_) { ptr_
= ptr_->parent; }
      ptr_ = ptr_->parent;
    }
    return *this;
  }
  tree_iterator<T> operator--(int) {
    tree_iterator<T> temp(*this);
    --(*this);
    return temp;
  }

private:
  // representation
  TreeNode<T>* ptr_;
  const ds_set<T>* set_;
};

// ------- ----------------------------------------------------
// DS_SET CLASS
template < class T>
class ds_set {
public:
  ds_set() : root_(NULL), size_(0) {}
  ds_set(const ds_set<T>& old) : size_(old.size_) {
    root_ = this->copy_tree(old.root_,NULL); }
  ~ds_set() {
    this->destroy_tree(root_);
    root_ = NULL;
  }
  ds_set& operator=(const ds_set<T>& old) {
    if (&old != this) {
      this->destroy_tree(root_);
      root_ = this->copy_tree(old.root_,NULL);
      size_ = old.size_;
    }
    return *this;
  }

  typedef tree_iterator<T> iterator;
  friend class tree_iterator<T>;

  int size() const { return size_; }
  bool operator==(const ds_set<T>& old) const { return (old.root_ ==
this->root_); }

  // FIND, INSERT & ERASE
  iterator find(const T& key_value) { return find(key_value, root_); }
  std::pair< iterator, bool > insert(T const& key_value) { return
insert(key_value, root_, NULL); }
  int erase(T const& key_value) { return erase(key_value, root_); }

  // OUTPUT & PRINTING
  friend std::ostream& operator<< (std::ostream& ostr, const ds_set<T>&
s) {
    s.print_in_order(ostr, s.root_);
    return ostr;
  }
  void print_as_sideways_tree(std::ostream& ostr) const {
    print_as_sideways_tree(ostr, root_, 0);
  }

  // ITERATORS
  iterator begin() const {
    if (!root_) return iterator(NULL,this);
    TreeNode<T>* p = root_;
    while (p->left) p = p->left;
    return iterator(p,this);
  }
  iterator end() const { return iterator(NULL,this); }

  bool sanity_check() const {
    if (root_ == NULL) return true;
    if (root_->parent != NULL) {
      return false;
    }
    return sanity_check(root_);
  }

private:
  // REPRESENTATION
  TreeNode<T>* root_;
  int size_;

  // PRIVATE HELPER FUNCTIONS
  TreeNode<T>*  copy_tree(TreeNode<T>* old_root, TreeNode<T>*
the_parent) {
    if (old_root == NULL)
      return NULL;
    TreeNode<T> *answer = new TreeNode<T>();
    answer->value = old_root->value;
    answer->left = copy_tree(old_root->left,answer);
    answer->right = copy_tree(old_root->right,answer);
    answer->parent = the_parent;
    return answer;
  }

  void destroy_tree(TreeNode<T>* p) {
    if (!p) return;
    destroy_tree(p->right);
    destroy_tree(p->left);
    delete p;
  }

  iterator find(const T& key_value, TreeNode<T>* p) {
    if (!p) return end();
    if (p->value > key_value)
      return find(key_value, p->left);
    else if (p->value < key_value)
      return find(key_value, p->right);
    else
      return iterator(p,this);
  }

  std::pair<iterator,bool> insert(const T& key_value, TreeNode<T>*& p,
TreeNode<T>* the_parent) {
    if (!p) {
      p = new TreeNode<T>(key_value);
      p->parent = the_parent;
      this->size_++;
      return std::pair<iterator,bool>(iterator(p,this), true);
    }
    else if (key_value < p->value)
      return insert(key_value, p->left, p);
    else if (key_value > p->value)
      return insert(key_value, p->right, p);
    else
      return std::pair<iterator,bool>(iterator(p,this), false);
  }

  int erase(T const& key_value, TreeNode<T>* &p) {
    if (!p) return 0;

    // look left & right
    if (p->value < key_value)
      return erase(key_value, p->right);
    else if (p->value > key_value)
      return erase(key_value, p->left);

    // Found the node.  Let's delete it
    assert (p->value == key_value);
    if (!p->left && !p->right) { // leaf
      delete p;
      p=NULL;
      this->size_--;
    } else if (!p->left) { // no left child
      TreeNode<T>* q = p;
      p=p->right;
      assert (p->parent == q);
      p->parent = q->parent;
      delete q;
      this->size_--;
    } else if (!p->right) { // no right child
      TreeNode<T>* q = p;
      p=p->left;
      assert (p->parent == q);
      p->parent = q->parent;
      delete q;
      this->size_--;
    } else { // find rightmost node in left subtree
      TreeNode<T>* q = p->left;
      while (q->right) q = q->right;
      p->value = q->value;
      // recursively remove the value from the left subtree
      int check = erase(q->value, p->left);
      assert (check == 1);
    }
    return 1;
  }

  void print_in_order(std::ostream& ostr, const TreeNode<T>* p) const {
    if (p) {
      print_in_order(ostr, p->left);
      ostr << p->value << "\n";
      print_in_order(ostr, p->right);
    }
  }

  void print_as_sideways_tree(std::ostream& ostr, const TreeNode<T>* p,
int depth) const {
    if (p) {
      print_as_sideways_tree(ostr, p->right, depth+1);
      for (int i=0; i<depth; ++i) ostr << "    ";
      ostr << p->value << "\n";
      print_as_sideways_tree(ostr, p->left, depth+1);
    }
  }

  bool sanity_check(TreeNode<T>* p) const {
    if (p == NULL) return true;
    if (p->left != NULL && p->left->parent != p) {
      std::cout << "Error: this node's left child's parent should be
this node!" << std::endl;
      return false;
    }
    if (p->right != NULL && p->right->parent != p) {
      std::cout << "Error: this node's right child's parent should be
this node!" << std::endl;
      return false;
    }
    return sanity_check(p->left) && sanity_check(p->right);
  }

};

#endif
```

Xinhao Luo luox6@rpi.edu