Kirsten Szeto 662044026 Mentors: Nia, Samir, David, Xenia, Abby
Typedefs - typedef type t_name → EX: typedef map<string, vector<int> > map_vect;
Lists: sequentially linked structure
Push_front, pop_front, push_back, pop_back, list.sort(opt_func) (vect → std::sort(beg, end, opt))
Cannot add integers to list iterators ex: itr + 4 → can for vector + string itrs
Operators: as member functions → obj.operator=(s_obj)
→ Private variables can be accessed directly, s_obj can be accessed by rhs. || one parameter and const
Operators: as non-member functions → operator=(first, second)
→ Private variables cannot be accessed directly + two parameters
Friend Class: allows for access all the private member variables and functions of class as if public
Friend Function: allows for Friend function to access and work just like member function
Note: Friend class must be declared as public || We cannot create new operators or change # of args
Maps: #include <map> – search, insert, erase = O(log n) || keys are ordered (need op<) no duplicates
std::map<key_type, value_type> var_name - declaration || std::pair <**const** key_type, value_type> - entry
std::map<std::string, int>::const_iterator it - iterator
map.find(key) - find func → returns map itr to position of key else map.end() || .size()
map.insert(std::make_pair(val, val)); → inserts pair, returns pointer to insertion/place, bool of inserted
map.erase(iterator p) → deletes pair referred to by p || map.erase(itr first, itr, last) → deletes from first to last- not last || size_type erase(const key_type& k) → earases pair containing k return bool of deleted
[] operator - Ex: ++count[s] looks for key of s, if none create and increment, if there, just increment
Map Iterators: std::map<type, type>::iterator it, can be incremented with ++, - -. it can access first and second with it->first, it->second
Pairs: #include <utility>
std::pair<int, double> p1(8, 8.8) || std::make_pair(8, 9.0) - declarations || .first or .second - accessing
Sets; ordered containers storing unique "keys" || sorts with < to keys || Insert, erase, find = O(log n)
std::set<const val_type> set_1() - declaration NOTE: no []operator and have .size
pair<iterator, bool> set<key>::insert(const Key& entry) – returns pos of insertion or exist, bool if inserted
Iterator set<Key>::insert(iterator pos, const Key& entry) – return pos of insertion, pos is a "hint"
size_type set<Key>::erase(const Key& x); – returns 1 removed, 0 for not removed
void set<Key>::erase(itr p) – erases at iterator || void set<Key>::erase(iterator first, iterator last) – erase
Const_iterator set<key>::find(const Key& x) const; - find – returns pos or set.end() if not found
Binary Trees: empty or node that has pointers to two binary sub-trees;
top most node = root, left +right node = children + subtrees, node with null left + right = leafs
Binary Search Trees (BST): at each node of the tree, the value stored at the node is >= all the values stored in the left subtree and <= all the values stored in the right subtree
Balanced Tree: the number of nodes on each subtree is approximately the same
given : balanced tree from 1 - 7, 4 root ABCD - bring in print in order code
In-order Traversal - print from smallest to largest (print left, me, print right) - 1234567
Post-order Traversal - print children then parent (print left, print right, me) - 1325764
Pre-order Traversal - parents first then children - (print me, print left, print right) - 4213657
B+ Trees: trees that can have up to B children and B - 1 keys || Good for everything bc flat + wide
Depth-first - follow all the way to leaf then back up to decision point (in, pre, post order trav)

- Quick but takes a while if wrong decision is made early on

Breadth-first - nodes closer to roots visited first || solution closer to root found first || memory intensive

Ds_set - root_, size_ → priv representation

```
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);
}
```

```
template <class T> class TreeNode {
public:
  TreeNode() : left(NULL), right(NULL) {}
  TreeNode(const T& init) : value(init), left(NULL), right(NULL) {}
  T value;
  TreeNode* left;
  TreeNode* right;
};
```

```
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);
}
```

```
std::string s;
std::map<std::string, int> counters; // store each word and a

// read the input, keeping track of each word and how often w
while (std::cin >> s)
  ++counters[s];

// write the words and associated counts
std::map<std::string, int>::const_iterator it;
for (it = counters.begin(); it != counters.end(); ++it) {
  std::cout << it->first << "\t" << it->second << std::endl;
}
```

```
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;
}
```

Height of tree

```
unsigned int find_height(Node* root)
{
   if (root == nullptr) {
      return 0;
   }
   else {
      unsigned int left_height =
find_height(root->left);
      unsigned int right_height =
find_height(root->right);
      return 1 + std::max(left_height,
right_height);
   }
```

Sum of odd map values

```
map<string, vector<int>> m;
unsigned int count = 0;
std::map<string,
vector<int>>::iterator it = m.begin();
while (it != m.end()) {
   for (unsigned int i=0;
i<it->second.size(); i++) {
      if (it->second[i]%2 == 1) {
         count += it->second[i];
      }
   }
   it++;
}
```

Smallest value in tree

```
template <class T>
TreeNode<T>*
FindSmallestInRange(const T& a,
const T& b, TreeNode<T>* root, T&
best_value){
 if(!root){ return NULL; }
   TreeNode<T>* left_subtree =
FindSmallestInRange(a,b,root->left,b
est_value);
   TreeNode<T>* right_subtree =
FindSmallestInRange(a,b,root->right,
best_value);
 if(root->value > a && root->value <
best_value){
    best_value = root->value;
    return root;
 }
 else if(left_subtree &&
left_subtree->value == best_value){
    return left_subtree;
 }
 else if (right_subtree){
    return right_subtree;
 }
 return NULL;
}
```

```
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);
  }
}
```