CSCI-1200 Data Structures — Fall 2024
Lecture 19 – Trees, Part IV

Test 3 Information

- Test 3 will be held **Thursday, November 14th, 2024 from 6:00-7:50pm**.
  - Student's assigned test room, row, and seat assignments will be re-randomized. Test 3 seating assignments will be posted and emailed Tuesday, November 12th.
  - No make-ups will be given except for pre-approved absence or emergency or illness, and a written excuse from the Dean of Students or the Student Experience office or the RPI Health Center will be required.
  - If you have a letter from Disability Services for Students and you have not already emailed it to `ds_instructors@cs.rpi.edu`, please do so IMMEDIATELY. Meredith Widman will be in contact to make arrangements for your test accommodations.
- Coverage: Lectures 1-20, Labs 1-12, HW 1-8.
  - Practice problems from previous tests are available on the course website.
  - Sample solutions to the practice problems will be posted on Wednesday morning.
  - The best way to prepare is to completely work through and write out your solution to each problem, *before* looking at the answers.
  - You should practice timing yourself as well. The test will be 110 minutes and there will be 100 points. If a problem is worth 25 points, budgeting 25 minutes for yourself to solve the problem is a good time management technique.
  - The exam will be handwritten on paper. You're also encouraged to practice *legibly* handwriting your answers to the practice problems on paper.
- OPTIONAL: Prepare a 2 page, black & white, 8.5x11", portrait orientation .pdf of notes you would like to have during the test. This may be digitally prepared or handwritten and scanned or photographed. The file may be no bigger than 2MB. You will upload this file to Submitty gradeable "Test 3 Notes Page (Optional)" before Wednesday, November 13th @11:59pm. We will print this and attach it to your test. No other notes may be used during the test.
- Going in to the test, you should know what big topics will be covered on the test. As you skim through the problems, see if you can match up those big topics to each question. Even if you are stumped about how to solve the whole problem, or some of the details of the problem, make sure you demonstrate your understanding of the big topic that is covered in that question.
- Re-read the problem statement carefully. Make sure you didn't miss anything.
- Additional Notes:
  - Please use the restroom before entering the exam room. Except for emergencies, students must remain in their seats until they are ready to turn in their exam. You may leave early if you finish the exam early.
  - Bring your Rensselaer photo ID card. We will be checking IDs when you turn in your exam.
  - Bring your own pencil(s) & eraser (pens are ok, but not recommended). The test *will* involve handwriting code on paper (and other short answer problem solving). Neat, legible handwriting is appreciated. We will be somewhat forgiving to minor syntax errors – it will be graded by humans not computers :)
  - Do not bring your own scratch paper. The exam packet will include sufficient scratch paper.
  - Computers, cell-phones, smart watches, calculators, music players, headphones, etc. are not permitted. Please do not bring your laptop, books, backpack, etc. to the test room – leave everything in your dorm room. *Unless you are coming directly from another class or sports/club meeting.*

Review from Lecture 18 & Lab 11

- BST / `ds_set` iterator increment (`operator++`) & decrement (`operator--`)
- Every node stores `Node` parent pointer *or*
  iterator stores a vector of `Node` pointers (the path from root Node).
- Overview *discussion* of `erase` from a BST

Today's Lecture

- Implement `erase` from a `ds_set`
- Some more exercises with trees & Big O Notation
- Limitations of our `ds_set` implementation, brief intro to red-black trees
- BONUS TOPIC: Template Specialization

19.1 Erase

First we need to find the node to remove. Once it is found, the actual removal is easy if the node has no children or only one child. *Draw picture of each case!*

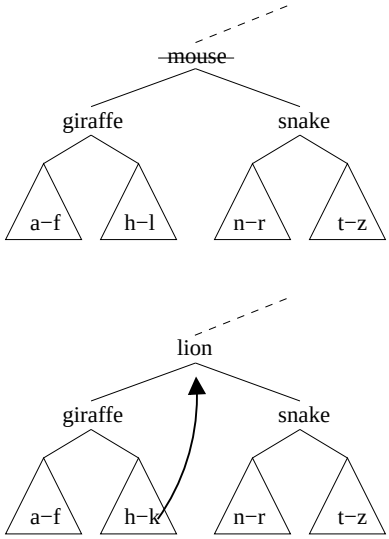
no children

only a left child
(with potentially a big subtree)

only a right child
(with potentially a big subtree)

It is harder if there are two children:

- Find the node with the greatest value in the left subtree or the node with the smallest value in the right subtree.
- The value in this node may be safely moved into the current node because of the tree ordering.
- Then we recursively apply erase to remove that node — which is guaranteed to have at most one child.



Exercise: Write a recursive version of erase.
Note: ignore parent pointers initially!

Exercise: How does the order that nodes are deleted affect the tree structure? Starting with a mostly balanced tree, give an erase ordering that yields an unbalanced tree.

19.2 A Note about Parent Pointers...

- If we choose to implement the iterators using parent pointers, we will need to:
  - add the parent to the Node representation
  - revise `insert` to set parent pointers (see attached code)
  - revise `copy_tree` to set parent pointers (see attached code)
  - revise `erase` to update with parent pointers
- **Exercise:** Rewrite `erase`, now with parent pointers.

19.3 Height and Height Calculation Algorithm

- The *height* of a node in a tree is the length of the longest path down the tree from that node to a leaf node. The height of a leaf is 1. We will think of the height of a null pointer as 0.
- The height of the tree is the height of the root node, and therefore if the tree is empty the height will be 0.

  Exercise: Write a simple recursive algorithm to calculate the height of a tree.

- What is the best/average/worst-case running time of this algorithm? What is the best/average/worst-case memory usage of this algorithm? Give a specific example tree that illustrates each case.

19.4 Shortest Paths to Leaf Node

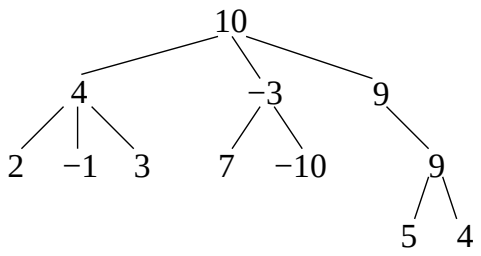
- Now let's write a function to instead calculate the *shortest* path to a NULL child pointer.

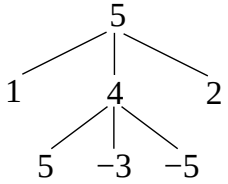
- What is the running time of this algorithm? Can we do better? *Hint: How does a breadth-first vs. depth-first algorithm for this problem compare?*

19.5 A Practice Test Tree Problem

A *trinary tree* is similar to a binary tree except that each node has at most 3 children. Write a *recursive* function named `EqualsChildrenSum` that takes one argument, a pointer to the root of a trinary tree, and returns true if the value at each non-leaf node is the sum of the values of all of its children and false otherwise. In the examples below, the tree on the left will return true and the tree on the right will return false.

```
class Node {
public:
  int value;
  Node* left;
  Node* middle;
  Node* right;
};
```





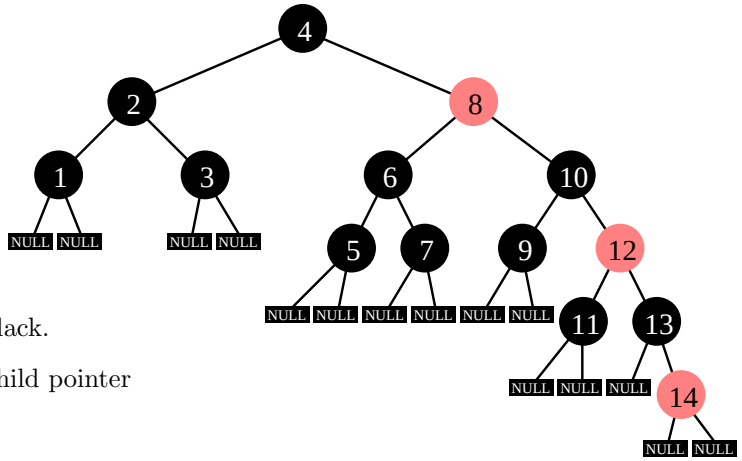
19.6 Limitations of Our BST Implementation

- The efficiency of the main insert, find and erase algorithms depends on the height of the tree.
- The best-case and average-case heights of a binary search tree storing n nodes are both $O(\log n)$. The worst-case, which often can happen in practice, is $O(n)$.
- Developing more sophisticated algorithms to avoid the worst-case behavior will be covered in Introduction to Algorithms. One elegant extension to binary search tree is described below...

19.7 Red-Black Trees

In addition to the binary search tree properties, the following red-black tree properties are maintained throughout all modifications to the data structure:

1. Each node is either red or black.
2. The NULL child pointers are black.
3. Both children of every red node are black.
   Thus, the parent of a red node must also be black.
4. All paths from a particular node to a NULL child pointer contain the same number of black nodes.



What tree does our `ds_set` implementation produce if we insert the numbers 1-14 *in order*?
The tree at the right is the result using a red-black tree. Notice how the tree is still quite balanced.
Visit these links for an animation of the sequential insertion and re-balancing:

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
http://www.youtube.com/watch?v=vDHFF4wjWYU&noredirect=1

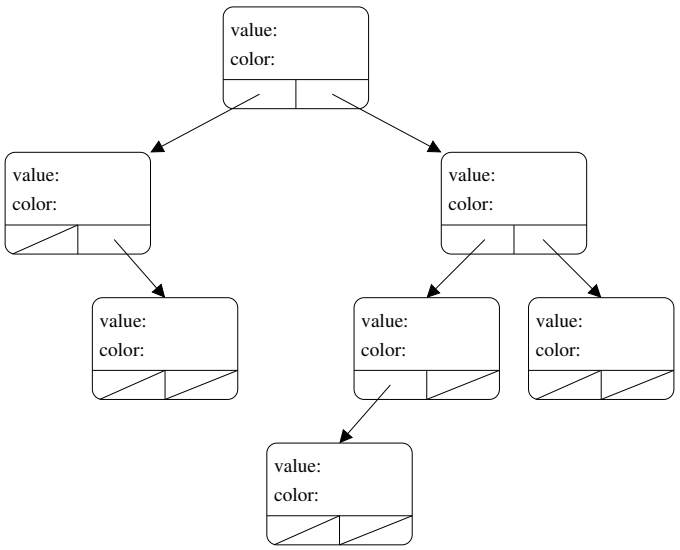
- What is the best/average/worst case height of a red-black tree with n nodes?

- What is the best/average/worst case shortest-path from root to leaf node in a red-black tree with n nodes?

19.8 Exercise [/6]

Fill in the tree on the right with the integers 1-7 to make a binary search tree. Also, color each node "red" or "black" so that the tree also fulfills the requirements of a Red-Black tree.

Draw two other red-black binary search trees with the values 1-7.



Note: Red-Black Trees are just one algorithm for *self-balancing binary search tree*.
Others include: AVL trees, Splay Trees, (& more!).

19.9 BONUS TOPIC: Template Specialization Example

Writing templated functions is elegant and powerful, but sometimes we do not want to handle all types in exactly the same way. Sometimes we want to write different versions of the function depending on the type:

- Let's study and discussion the following code:

```
// We'll use this templated function (unless we find a specialized
// implementation for our type)
template <class T>
void print_vec (const std::vector<T> &v) {
  std::cout << "count=" << v.size() << "  data=";
  for (unsigned int i = 0; i < v.size(); i++) {
    std::cout << " " << v[i]; }
  std::cout << std::endl;
}

// This will match doubles (but not floats)
void print_vec (const std::vector<double> &v) {
  std::cout << "count=" << v.size() << "  data=";
  for (unsigned int i = 0; i < v.size(); i++) {
    std::cout << std::setprecision(1) << std::fixed << " " << v[i]; }
  // unset the formatting
  std::cout << std::defaultfloat << std::endl;
}

int main() {
  // note: this syntax for initialization of vector contents is available with C++11
  std::vector<int> int_v = { 1, 2, 3, 4, 5 };
  std::vector<double> double_v = { 1, 2, 3, 4, 5 };
  std::vector<float> float_v = { 1, 2, 3, 4, 5 };
  std::vector<std::string> string_v = { "1", "2", "3", "4", "5" };
  print_vec(int_v);
  print_vec(double_v);
  print_vec(float_v);
  print_vec(string_v);
}

// This would match strings...  but because it's placed after the
// usage in main it's not used!?!?!
void print_vec (const std::vector<std::string> &v) {
  std::cout << "count=" << v.size() << "  data=";
  for (unsigned int i = 0; i < v.size(); i++) {
    std::cout << " \"" << v[i] << "\""; }
  std::cout << std::endl;
}
```

- If we commented out the specialized implementations of `print_vec` for the double and string types:

```
count=5  data= 1 2 3 4 5
count=5  data= 1 2 3 4 5
count=5  data= 1 2 3 4 5
count=5  data= 1 2 3 4 5
```

- If we run the original code:

```
count=5  data= 1 2 3 4 5
count=5  data= 1.0 2.0 3.0 4.0 5.0
count=5  data= 1 2 3 4 5
count=5  data= 1 2 3 4 5
```

- If we swap the order of the main function and the string version of `print_vec`:

```
count=5  data= 1 2 3 4 5
count=5  data= 1.0 2.0 3.0 4.0 5.0
count=5  data= 1 2 3 4 5
count=5  data= "1" "2" "3" "4" "5"
```

```
#ifndef ds_set_h_
#define ds_set_h_
#include <iostream>
#include <utility>
// ----------------------------------------------------------------------
// DS_SET CLASS -- WITH NESTED NODE & ITERATOR CLASSES (ALTERNATE STYLE)
template <class T>
class ds_set {
public:
  // NODE CLASS
  class Node {
  public:
    Node() : left(NULL), right(NULL), parent(NULL) {}
    Node(const T& init) : value(init), left(NULL), right(NULL), parent(NULL) {}
    T value;
    Node* left;
    Node* right;
    Node* parent; // to allow implementation of iterator increment & decrement
  };
  // ITERATOR CLASS
  class iterator {
    public:
    iterator() : ptr_(NULL) {}
    iterator(Node* p) : ptr_(p) {}
    // operator* gives constant access to the value at the pointer
    const T& operator*() const { return ptr_->value; }
    // comparions operators are straightforward
    bool operator== (const iterator& rgt) { return ptr_ == rgt.ptr_; }
    bool operator!= (const iterator& rgt) { return ptr_ != rgt.ptr_; }
    // pre & post increment & decrement operators
    iterator & operator++(); { /* implementation omitted */ }                  // ++itr
    iterator operator++(int) { iterator temp(*this); ++(*this); return temp; } // itr++
    iterator & operator--() { /* implementation omitted */ }                   // --itr
    iterator operator--(int) { iterator temp(*this); --(*this); return temp; } // itr--
  private:
    // representation
    Node* ptr_;
  };

  // DS_SET CONSTRUCTORS, ASSIGNMENT OPERATOR, & DESTRUCTOR
  ds_set() : root_(NULL), size_(0) {}
  ds_set(const ds_set& old) : size_(old.size_) { root_ = copy_tree(old.root_,NULL); }
  ~ds_set() { destroy_tree(root_); root_ = NULL; }
  ds_set& operator=(const ds_set<T>& old) { /* implementation omitted */ }
  // SET FUNCTIONALITY
  int size() const { return size_; }
  iterator begin() const { /* implementation omitted */ }
  iterator end() const { return iterator(NULL); }
  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_, N
ULL); }
  int erase(T const& key_value) { return erase(key_value, root_); }
private:
  // REPRESENTATION
  Node* root_;
  int size_;
  // PRIVATE HELPER FUNCTIONS
  Node* copy_tree(Node* old_root, Node* the_parent);
  void destroy_tree(Node* p) { /* implementation omitted */ }
  iterator find(const T& key_value, Node* p) { /* implementation omitted */ }
  std::pair<iterator,bool> insert(const T& key_value, Node*& p, Node* the_parent);
  int erase(T const& key_value, Node* &p);
};

// DS_SET FUNCTIONS
template <class T>
typename ds_set<T>::Node* ds_set<T>::copy_tree(Node* old_root, Node* the_parent) {
  if (old_root == NULL)
    return NULL;
  Node *answer = new Node();
  answer->value = old_root->value;
  answer->left = copy_tree(old_root->left,answer);
  answer->right = copy_tree(old_root->right,answer);

  // extra argument is necessary to allow us to set the parent pointers
  answer->parent = the_parent;

  return answer;
}

template <class T>
std::pair<typename ds_set<T>::iterator,bool>
ds_set<T>::insert(const T& key_value, Node*& p, Node* the_parent) {
  if (!p) {
    p = new Node(key_value);

    // extra argument is necessary to allow us to set the parent pointers
    p->parent = the_parent;

    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);
}

template <class T>
int ds_set<T>::erase(T const& key_value, Node* &p) {
  /* Implemented in Lecture 18 */









}

#endif
```