CSCI-1200 Data Structures — Fall 2025
Lecture 21 – Trees, Part IV

Review from Lecture 20

- Breadth First Traversal

```
template <class T>
void breadth_first_print(TreeNode<T>* root) {
  int counter = 1;
  if (root == NULL) return;
  std::list< TreeNode<T>* > current;  // list of all nodes on a specific level
  current.push_back(root);
  std::list< TreeNode<T>* > next;     // list of all nodes on the next level
  while ( current.size() > 0 ) {      // print everything at this level
    std::cout << "level " << counter << ": ";
    typename std::list<TreeNode<T>*>::iterator itr = current.begin();
    while ( itr != current.end() ) {  // and collect items for next level
      TreeNode<T> *tmp = *itr;
      std::cout << tmp->value << " ";
      if (tmp->left != NULL) { next.push_back(tmp->left); }
      if (tmp->right != NULL) { next.push_back(tmp->right); }
      itr++;
    }
    current = next;                   // move on to the next level!
    next.clear();
    counter++;
    std::cout << std::endl;
  }
}
```

- BST / `ds_set` iterator increment (`operator++`) & decrement (`operator--`)

```
template <class T>
typename ds_set<T>::iterator& ds_set<T>::iterator::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;
}
```

- Every node stores `Node` parent pointer *or*
  iterator stores a vector of `Node` pointers (the path from root Node).

  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
  - revise `copy_tree` to set parent pointers
  - revise `erase` to update with parent pointers
- Overview *discussion* of `erase` from a BST

Today's Lecture

- Insert with parent pointers - *and the importance of pass-by-reference*
- Finish implement `erase` from a `ds_set`
- Tree height & longest/shortest paths from root to leaf node / null pointer
- Limitations of our `ds_set` implementation, brief intro to red-black trees
- BONUS TOPIC: Template Specialization

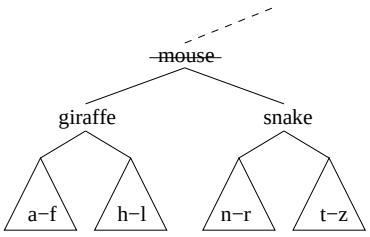
21.1 Re-Implementation of Insert with Parent Pointers

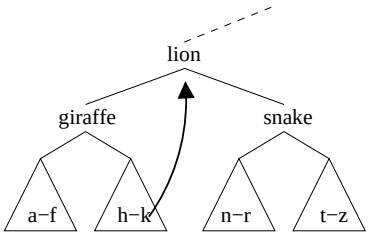
```
public:
  std::pair< iterator, bool > insert(T const& key_value) {
     return insert(key_value, root_, NULL);
  }

private:
  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);
  }
```

21.2 Implementation of 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!

- 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.






```
template <class T>
int ds_set<T>::erase(T const& key_value, Node* &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;
    size_--;
  } else if (!p->left) { // no left child
    Node* q = p;
    p=p->right;
    assert (p->parent == q);
    p->parent = q->parent;
    delete q;
    size_--;
  } else if (!p->right) { // no right child
    Node* q = p;
    p=p->left;
    assert (p->parent == q);
    p->parent = q->parent;
    delete q;
    size_--;
  } else { // Find rightmost node in left subtree
    Node* 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;
}
```

21.3 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.

21.4 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?

21.5 Limitations of Our BST Implementation

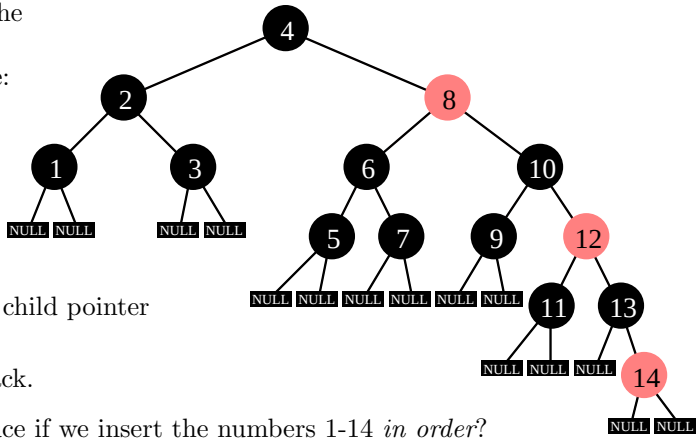
- The efficiency of the main insert, find and erase algorithms depends on the height of the tree.
- The best-case and average-case heights of a binary search tree storing n nodes are both $O(\log n)$. The worst-case, which often can happen in practice, is $O(n)$.
- Developing more sophisticated algorithms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

21.6 Red-Black Trees

- **Intuition:** A binary tree with $n = 2^k - 1$ nodes can be *exactly balanced*. We would draw this tree using only black nodes. However, if the tree needs to store additional nodes, the height will increase in some branches of the tree and it will become somewhat unbalanced. We will monitor and limit where and how unbalanced the tree is by adding red nodes and occasionally rearranging & recoloring the nodes to distribute the extra data.

- 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. *The parent of a red node must also be black. We cannot have 2 red nodes in a row.*
  4. All paths from a particular node to a NULL child pointer contain the same number of black nodes.
  5. *Optional requirement:* The root is always black.



- What tree does our `ds_set` implementation produce if we insert the numbers 1-14 *in order*? The tree above is the result using a red-black tree. Notice how the tree is still quite balanced.

- After each insert or erase we need to check the red-black properties, and perform **rotation**s and/or **recoloring**s to re-establish the red-black properties. Even with these additional implementation steps, insert and erase remain $O(log\ n)$! We won't cover the implementation, proof of the correctness, or full analysis in this course.

- There are lots of internet resources about Red-Black trees – videos & interactive animations demonstrate the automatic rebalancing algorithm (with rotations & recolorings), e.g.:

  https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

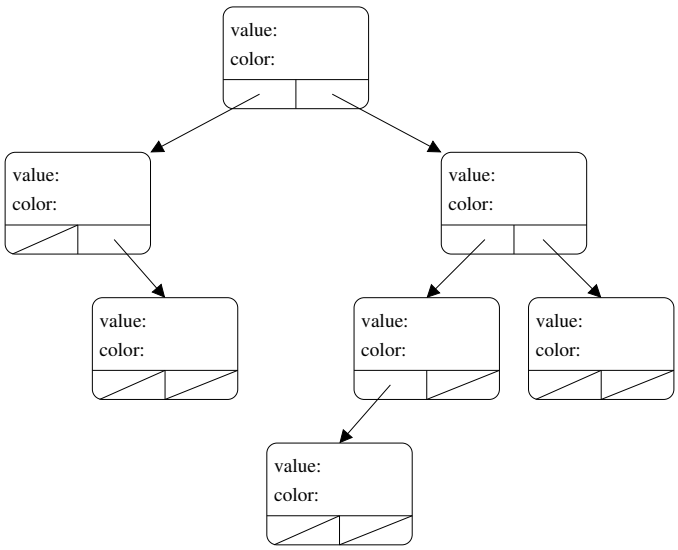
- What is the best/average/worst case height of a red-black tree with n nodes?

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

21.7 Practice Exam Question

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, 2-3-4 trees, (& more!).

21.8 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"
```