CS 2100: Data Structures & Algorithms 1

Red-Black Trees (brief) & Tree Applications

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

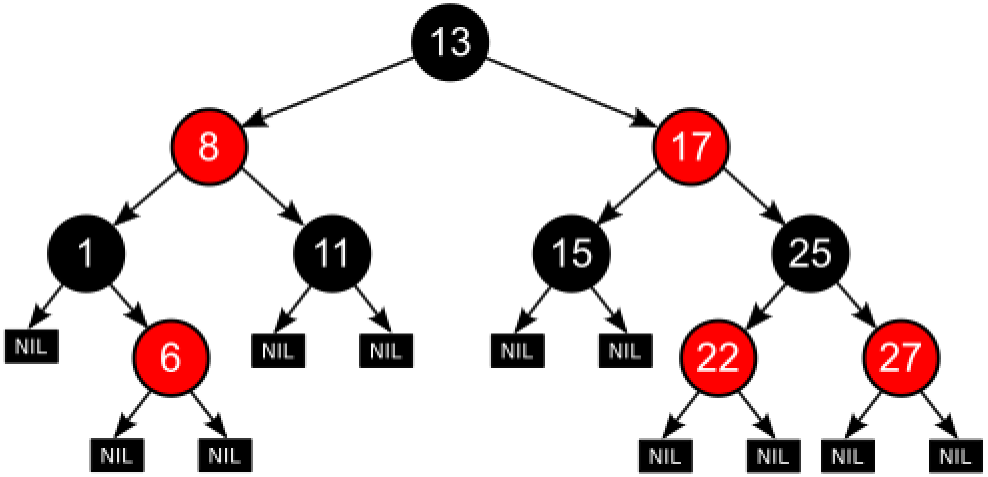
- Masks are **required** at all times during class (University Policy)
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Red-Black Trees

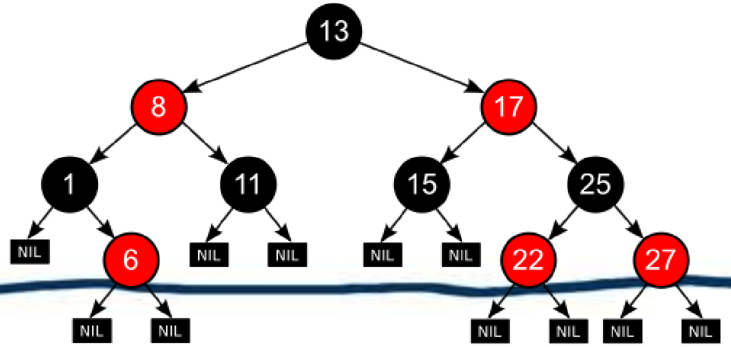
(Brief)

Red-Black Trees

- Each node has a color attribute, which is either (wait for it…) red or black ☺
- Animation site examples are **HERE** and **HERE**. *(All copyright remains with original author(s) as applicable)*. There are more out there, you can find, even one by Daniel Liang.



Red-Black Tree Properties



All of these properties must hold for a red-black tree

- A node is either **red** or **black**
- The root is **black**
- All leaves are **black**
  - The leaves may be the NULL children
- Both children of every **red** node are **black**
  - Therefore, a **black** node is the only possible parent for a **red** node
- Every simple path from a node to any descendant leaf contains the same number of **black** nodes
  - Counting or not counting the NULL **black** nodes; it doesn't make a difference as long as you are *consistent*

Red-Black Tree Operations

Insert

- Insert the node as for a normal BST
  - And color it **red**
- 5 possible cases:
  1. The new node is the root node
  2. The new node's parent is **black**
  3. Both the parent and uncle (aunt?) are **red**
  4. Parent is **red**, uncle/aunt is **black**, new node is the right child of parent
  5. Parent is **red**, uncle/aunt is **black**, new node is the left child of parent

Delete / Remove

- Do a normal BST remove
- Find next highest/lowest value, put its value in the node to be deleted, ***remove*** that highest/lowest node
  - Note that that node won't have 2 children!
- We replace the node to be deleted with its **left** child
  - This child is N, its sibling is S, its parent is P
- There are 6 possible cases! (See next slide)

Red-Black Tree: Removal Cases

- A total of 6 cases!
  1. N is the new root
  2. S is **red**
  3. P, S, and S's children are **black**
  4. S and S's children are **black**, but P is **red**
  5. S is **black**, S's left child is **red**, S's right child is **black**, and N is the left child of its parent
  6. S is **black**, S's right child is **red**, and N is the left child of parent P

- We won't see them in detail, though, but you can find details on the Wiki
  - https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

Why Red-Black Trees vs. AVL Trees?

- **AVL trees** are more rigidly balanced than **red-black trees**
  - Thus, more rotations are required during the operations in the worst case

- Time-critical applications will see a *performance boost*

- Functional programming languages used red-black trees for ***associative arrays (hashes)***
  - The tree can be a persistent data structure
    - A data structure that retains a "memory" of its mutations

- **Main take-away:**
  - Red-Black Trees and BSTs/AVL Trees have the same Big-Theta run time
  - However, Red-Black trees ***perform better in practice*** due to…
    - Generally lower constant factors
    - Doesn't rotate as often

Tree Applications

Some examples and concluding thoughts on Trees

When Are Trees Not Good To Use?

- ***Trees are fast*** -- so when would we not want to use them?
  - When the items do not have a sorted order
    - A list of todo tasks
  - When we want less complexity
    - A stack or a queue
  - When we want an $\Theta(1)$ operation on retrieves
    - Vector get()
  - When we want an $\Theta(1)$ time for all operations
    - **Hash tables** can (almost) achieve that

Application Of Tress: Programs

- Any program can be represented as a tree; consider the following program (no externall source code):

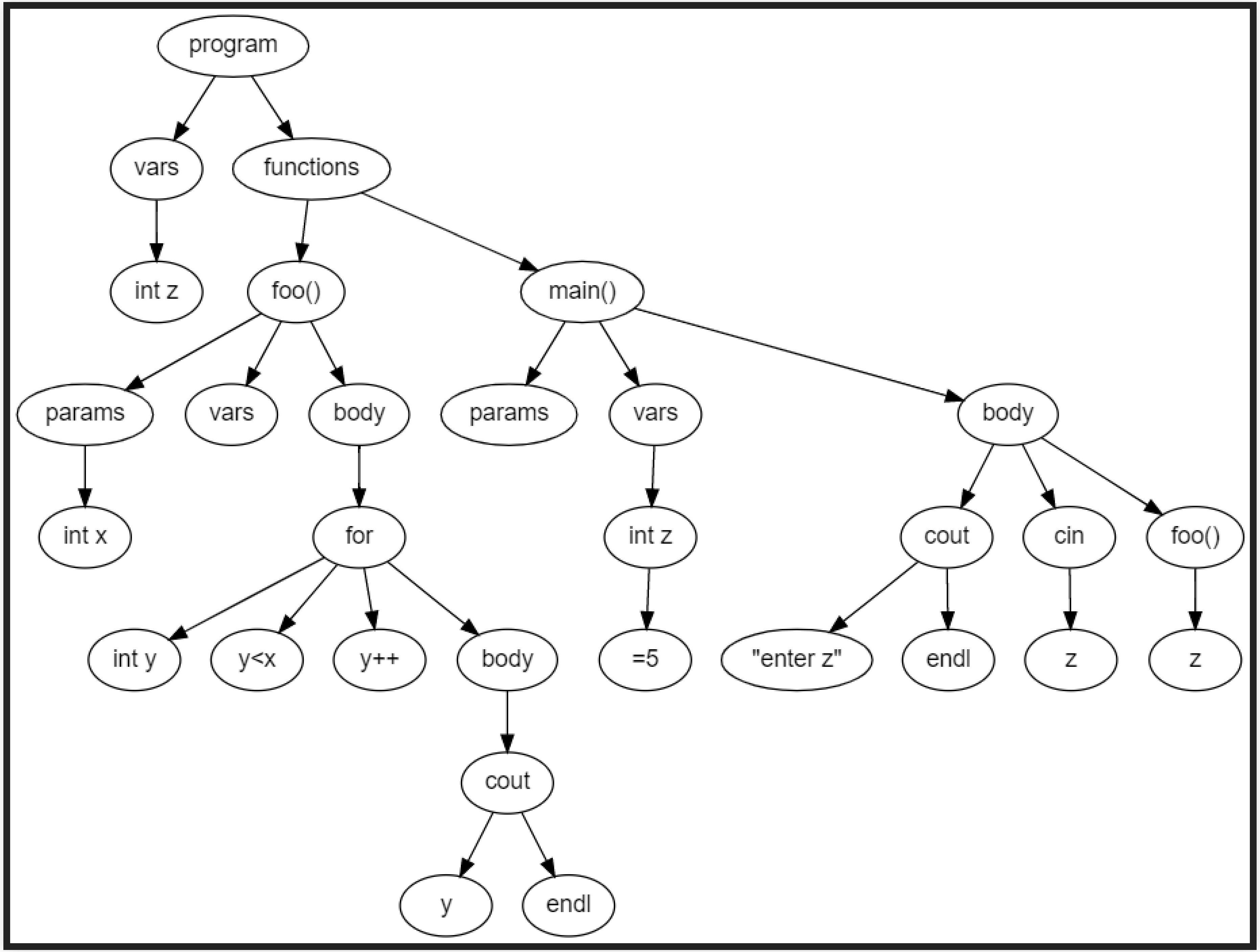
```
int z;

int foo (int x) {
    for ( int y = 0; y < x; y++ )
        cout << y << endl;
}

int main() {
    int z = 5;
    cout << "enter x" << endl;
    cin >> z;
    foo(z);
}
```

- Note that there are two `int z` declarations; this will be relevant shortly

The Program Tree



Notes on The Program Tree

- Called an "abstract syntax tree" or a "parse tree"

- Each **node** can be a different type
  - Having different properties and different number of children
    - A for loop node has four children (for init, for expression, for update, body)
    - A function node has at least three children (parameters, variables, body)
      - (we are ignoring other possible children of a function node here)
  - A body node has a variable number children

- A **compiler** will build such a tree in memory
  - And traverse it many times
  - For example, to figure out which 'z' is used in the main() function
  - Or to do code generation
    - Each node has an overridden method to generate the code for that node
  - Or to do type checking
  - Or to do code optimization

Comparing Two Programs

- What if we read in two programs…
  - … and build parse trees for each
  - … and compare their structure?

- We would be able to compare the two programs while ignoring such things as:
  - Function / method order
  - Variable renaming
  - Different comments

Measure of Structural Similarity

- "*A System for Detecting Software Plagiarism*" (website - http://theory.stanford.edu/~aiken/moss/)
  - The paper the site is based on can be found here (http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf)
- It will load up all the programs for a class
- And do all n^2 comparisons
- And display the most similar programs