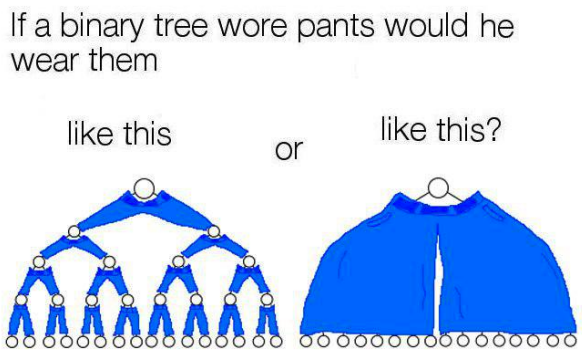
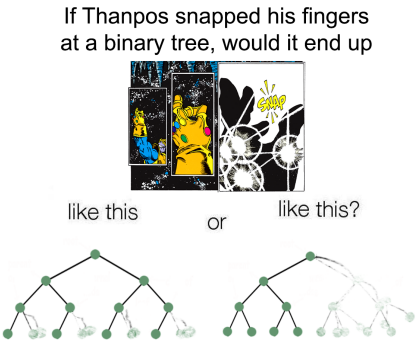
Week 8





Announcements

- Good job with project 3!
- HW 4
  - due Tuesday (5/25)

Questions?

- anything?
  - code
    - traversal
    - creating a tree
  - inserting to the middle
  - deleting

Trees

- like a linked list, but nodes point to multiple children
  - if each node has at most 2 children, its a **binary** tree
- instead of a head node we have a root node (and root pointer)
- very efficient data structure for certain algorithms (we'll get to these towards the end of today)

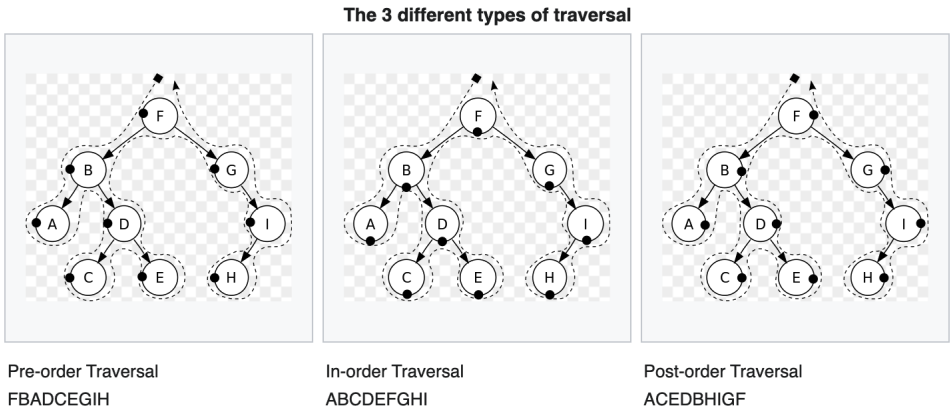
```
// root (the node at the top level)
// nodes (0-2 children if a binary tree)
// edges connect nodes
  // a path is the sequence of edges between two nodes
  // must have a unique path from the root to any child
// leaf nodes (0 children)

     1                    // root
    / \
   /   \
  2     \
 / \     3                // 2, 3, and 8 are (internal) nodes
```

```
4   5   / \
       9   \
            8
           / \
          6   7             // 4, 5, 9, 6, and 7 are leaves

// this is for binary trees
struct BTNODE {
  int value;
  BTNODE *left;
  BTNODE *right;
}

// traversals

// pre-order
// in-order
// post-order
// level order
```

```
// tree algorithms usually lend themselves to recursive structure

struct Node {
  string name;
  vector<Node*> children;
}

// todo
int countNodes(const Node* p) {

  if (p == nullptr)
    return 0;

  if ( (p->children).size() == 0)
    return 1;

  int sum = 1;

  for ( vector<Node*>::iterator it = (p->children).begin();
    it != (p->children).end(); it++ ) {

    sum += countNodes( *it );

  }

  return sum;

}
```

Traversals

- image is from this link: https://en.wikibooks.org/wiki/A-level_Computing_2009/AQA/Problem_Solving,_Programming,_Operating_Systems,_Databases_and_Networking/Programı



Pre-Order (current node, left, then right)

```
void PreOrder(Node* cur) {

  if (cur == nullptr)      // if empty, return
    return;

  cout << cur-> value;    // * process CURRENT node * //

  PreOrder(cur->left);    // LEFT sub-tree

  PreOrder(cur->right);   // RIGHT sub-tree
}
```

In-Order (left, current node, then right)

```
// todo: change order of the statements
void InOrder(Node* cur) {

  if (cur == nullptr)      // if empty, return
    return;

  InOrder(cur->left);    // LEFT sub-tree

  cout << cur-> value;    // * process CURRENT node * //

  InOrder(cur->right);   // RIGHT sub-tree
}
```

Post-Order (left, right, then current node)

```
// todo: change order of the statements
void PostOrder(Node* cur) {

  if (cur == nullptr)      // if empty, return
    return;

  PostOrder(cur->left);    // LEFT sub-tree

  PostOrder(cur->right);   // RIGHT sub-tree

  cout << cur-> value;    // * process CURRENT node * //
}
```

Level order (left to right at each level)

```
// use a queue
add root to your queue

while queue not empty:
  dequeue, print the node value
  add children to queue (if not NULL)

    b
   / \
  a   c
 / \
d   e

first 2 steps of recursion:
queue start, printed value, queue end
     [b]           b        [a c]
    [a c]          a        [c d e ]
```

Some Q's

- With which algorithm can we...
  - We can print BST in alphabetical order?
    - in-order
  - Free nodes?
    - post-order

Binary Search Trees (BST)

- like a tree, but with some rules on where nodes are placed
  - each LEFT child must be <= cur node
  - each RIGHT child must be > than cur node
- fast search on average
- animation site: http://btv.melezinek.cz/binary-search-tree.html

Search

- (think about recursive implementation)
  - time complexity for a good tree: O(log(n))
  - for an imbalanced tree: O(n)

```
// pseudocode

start at root pointer

{
  if v == cur node val , return

  if v < cur node, go left

  if v > cur node, go right

  if null, not found
}
```

Insert

- time complexity
  - balanced O(log(n))
  - unbalanced

```
// pseudocode

if the tree is empty
  allocate new node and point root pointer to our new node

else
  start at the root

  while we haven't inserted our node:

    if val == cur node
      return (already in our tree, nothing to do)

    if val < cur node
      if left child exists
        go left
      else
        allocate new node to the left child of cur node

    if val > cur node
      if right child exists
        go right
      else
        allocate new node to the right child of cur node


// inserting sorted data is... fast or slow?

1 2 3 4 5

1
 \
  2
   \
    3
```

Delete

- time complexity
  - balanced O(log(n))
  - unbalanced

```
// find V with 2 pointers
  // one to find node (binary search), one to track parent
  // if you found val, 3 cases
    // 1. delete a leaf
    // 2. target has 1 child
    // 3. target has 2 children


1. parent pointer set to NULL
   delete the node

    if root node, set root to NULL and delete child

2. link parent pointer to the target node's only child
   delete the node

   if root, re-link root to the child then delete child

3. replace node with left's largest child or right's smallest child
   these can be leaves or nodes with one child

   replace the value then use the algorithms above to delete the
   old node
```

BST applications

```
// STL map and set (among other containers) use balanced BST
// for the keys and values, respectively

// Huffman encoding (not covered/tested)
```

Balanced BST

```
// unbalanced, search is O( n )
5
 \
 10
 /
7
 \
  9
 /
8

// add some more values
    8
   / \
  7  15
 /   /
5   9
     \
      10
       \
       12

// ideal balancing of the above tree would look like this
      9
   7     12
  5 8   10 15

// want balance so you have O(log(n)) search, insertion, and deletion
// make sure heights of subtrees don't differ by more than 1
    // balancing is O(log(n))

// if asked about a BST in an interview, ask if its balanced!
```