Trees

The linked list we have worked with is a linear structure:
• can have pointers forward and backward (doubly linked), but still a linear structure
• what about hierarchical structures?

Trees vs Trees

• an arrangement of nodes
  • again, think of payload and pointers
  • the degree of the tree is the maximum number of "children" that any one node may have
    • binary trees, two children (max) per node
For the most part, if you understand \textit{binary} trees (degree==2, at most two children to a node), then you get most of the idea of a tree

Without getting into too many logic details, a lot of CS uses trees, especially full-complete (or nearly so) trees

Bad for many things we want to do
Much algorithmic work to change the trees to be "fuller"
Lots of applications

Trees (as an abstract data type, ADT), are very useful for lots of things.

Most useful when they are "fullish", because they have some nice properties

There is no STL tree 😊

Fun facts, full complete binary trees

- \( I \) internal nodes, \( L \) leaves is \( L = I + 1 \).
- \( I \) internal nodes, the total nodes \( N = 2I + 1 \).
- total of \( N \) nodes, total internal nodes is \( I = (N - 1)/2 \).
- full complete tree with \( N \) nodes has height \( < \log_2 N \)
  - thus, to traverse a tree from root to a leaf is \( O(\log n) \), a great property!

Simple principle.

- From any node, if target is smaller, go left. If larger, go right. Otherwise done.
- \( O(\log n) \) to search for a value. Why?
Abstract Syntax Tree
Evaluate bottom up to find the answer

Many ways, but the simple, obvious way related to our linked list is to use dynamic memory, payloads and pointers

```
template<typename T>
struct BNode{
    BNode* left_;  
    BNode* right_; 
    T data_; 
    ...
};
```

Let's build a binary search tree
- 5,3,7,2,1,6,4,8
• how to find where to insert 10?

Could go either way on what a tree class is:
• could just be the node
• could be a separate class

We'll go with a separate class

```cpp
template<typename T>
class BinaryTree{
private:
    BNode<T>* root_;

public:
    BinaryTree():root_(nullptr) {};
    BinaryTree(const BinaryTree&); ~BinaryTree();
    BinaryTree& operator=(BinaryTree);
    void insert(T);
    void insert(BNode<T>*&, BNode<T>*&);
    bool del_node(T);
    void del_node(BNode<T>*&);
    operator string();
    BNode<T>*& find(T);
    BNode<T>*& find(BNode<T>*&, BNode<T>*&);
    void BFS_print();

    template<typename T>
void BinaryTree<T>::insert(T d){
    BNode<T>* node = new BNode<T>(d);
    insert(root_,node);
}
```
recursive insert

template<typename T>
void BinaryTree<T>::insert(BNode<T>** ptr, 
BNode<T>** node){
    if (ptr == nullptr)
        ptr = node;
    else if (*ptr != *node){
        if (*node < *ptr)
            insert(ptr->left_, node);
        else if (*node > *ptr)
            insert(ptr->right_, node);
    }
}

what type?

why does this work?

what case is this?

Depth First Search

- Depth first search looks down a branch until it hits bottom
- then it comes up a node, and looks down the other branch, until it hits bottom.
- does this recursively!!!

operator string

Like the other operators we have seen, this is also an operator string
- it is the cast operator, converting the tree to a string
- interestingly, no return on these operators

DFS tree traversal, setup

template<typename T>
BinaryTree<T>::operator string(){
    string result="";
    return recurse_str(root_, result);
}
template<
type T>
string recurse_str(BNode<T>* ptr,
    string result)
{
    if (ptr == nullptr)
        return result;
    else
    {
        result = recurse_str(ptr->left_, result);
        result = result + to_string(ptr->data_);
        result = recurse_str(ptr->right_, result);
    }
    return result;
}

Not really right
Why??

could we change the order?
Yes, we could. We could visit in:
• infix
• prefix
• postfix
order depending on the order of the statements

Infix: left, node, right
result = recurse_str(ptr->left_, result);
result = result + to_string(ptr->data_);
result = recurse_str(ptr->right_, result);

Prefix: node, left, right
result = result + to_string(ptr->data_);
result = recurse_str(ptr->left_, result);
result = recurse_str(ptr->right_, result);

Postfix: left, right, node
result = recurse_str(ptr->left_, result);
result = recurse_str(ptr->right_, result);
result = result + to_string(ptr->data_);
Many times it is easiest to have two functions:
• setup the recursive start
• the actual recursive function

```cpp
template<typename T>
BNode<T>*& BinaryTree<T>::find(T d){
    return find_recurse(d, root_);
}

// function, recursive find value
template<typename T>
BNode<T>*& BinaryTree<T>::find(T val, BNode<T>*& node_ptr){
    if (node_ptr == nullptr)
        return node_ptr;
    if (node_ptr->data_ == val)
        return node_ptr;
    else if (val < node_ptr->data_)
        return find_recurse(val, node_ptr->left_);
    else
        return find_recurse(val, node_ptr->right_);
}
```

```cpp
template<typename T>
BinaryTree<T>::BinaryTree(const BinaryTree& bt){
    copy_recurse(root_, bt.root_);
}

// function for recursive copy. BNode is public so not
// required to be part of BinaryTree
template<typename T>
void copy_recurse(BNode<T>*& new_ptr,
    BNode<T>* to_copy_ptr){
     if (to_copy_ptr == nullptr)
        new_ptr = nullptr;
     else{
        new_ptr = new BNode<T>({to_copy_ptr->data_});
        copy_recurse(new_ptr->left_, to_copy_ptr->left_);
        copy_recurse(new_ptr->right_, to_copy_ptr->right_);
     }
}
```

```cpp
template<typename T>
BinaryTree<T>::~BinaryTree(){
    destruct_recurse(root_);
}

// function for recursive destruction
template<typename T>
void destruct_recurse(BNode<T>* bn_ptr){
    if (bn_ptr != nullptr){
        destruct_recurse(bn_ptr->left_);
        destruct_recurse(bn_ptr->right_);
        delete bn_ptr;
    }
}
```
template<typename T>
BinaryTree<T> &
BinaryTree<T>::operator=(BinaryTree bt) {
    swap(root_, bt.root_);
    return *this;
}

Breadth first search

Also called level search. Can't really do it recursively. (WHY?)

Basic idea is to use a deque (queue) and simulate in that fashion

Basic algorithm
• take a node from the front of the queue
• process that node
• push all the children of that node onto the back of the queue

Queue assures that you go through the nodes in level order!
We are going to use this for printing, so we want to remember the level and the node. We create a pair for this using directive allows for templates. Nice!

template <typename T>
void BinaryTree<T>::BFS_print()
{
    using NodePair = pair< BNode<T>*, int>;
    NodePair nd_pr;
    BNode<T>* node_ptr;
    T dat;
    int level, previous_level=-1;
    if (root_ == nullptr)
        return;
    deque < NodePair > bfs_queue;
    bfs_queue.push_back({root_, 0});
    while (!bfs_queue.empty())
    {
        nd_pr = bfs_queue.front();
        node_ptr = nd_pr.first;
        level = nd_pr.second;
        bfs_queue.pop_front();
        // if to get a newline for a new level
        // have to track previous_level, see if it is different
        if (level != previous_level)
        {
            cout << endl;
            cout << "Level " << level << " : ";
            previous_level = level;
            cout << " ";
        }
        // print the data
        cout << node_ptr->data_ << ", ";
        // update the queue
        if (node_ptr->left_ != nullptr)
            bfs_queue.push_back({node_ptr->left_, level+1});
        if (node_ptr->right_ != nullptr)
            bfs_queue.push_back({node_ptr->right_, level+1});
    }
    cout << endl;
}

Focus on Bnode<T>* & type
The reference type for a pointer means what?

That this ptr is an alias for the pointer that was sent in. Change that pointer and you change the original pointer

We will need that

deleting is a bit complicated
Three cases (really two):
• delete a node with no children
• delete a node with only 1 child
• delete a node with two children
```cpp
template<typename T>
void BinaryTree<T>::del_node(BNode<T>*& to_del)
{
    BNode<T>* copy_to_del = to_del;
    // 0 or 1 child cases
    if (to_del->right_ == nullptr)
    {
        to_del = to_del->left_;  
        if (to_del->left_ == nullptr)
        {
            to_del = nullptr;
            return;
        }  
        to_del = copy_to_del;
    }
    else
    {
        to_del = copy_to_del;
    }
    return;
}
```

As it turns out, if what 1’s left_ points to is the `nullptr`, then we still skip the node.
1. Move the right subtree up, skipping the node to be deleted
2. To fix the left subtree, we must reattach it somewhere
   1. We reattach it to the **leftmost node** of the **right subtree**

We need a picture!

---

**left subtree is the issue**

As we said, we can just move the right subtree up one and reattach, but where to put that left guy?

What do I know about the right tree’s leftmost node?

It is the smallest in that right subtree
// 2 child case
else{
  BNode<T>* right_ptr = to_del->right_;  
  BNode<T>* leftmost_ptr = right_ptr;
  while (leftmost_ptr->left_ != nullptr)  
    leftmost_ptr = leftmost_ptr->left_;  
  // repair left subtree
  leftmost_ptr->left_ = to_del->left_;  
  // repair right subtree
  to_del = to_del->right_;  
}  
delete copy_to_del;

Remember that a tree that is at least "kind of" full is a good tree. Its depth is about log₂(n) and that is nice

But not all trees end up nicely "balanced"

How can we make this tree more balanced?
We can do some relinking!

Note the search relations are preserved!!

Another relink

Note the search relations are preserved!!

This looks more complicated, but still unbalanced. With what we know, what might we do?

How about a combo rotation

left rotation

right rotation

How can we make this tree more balanced?
How about a combo rotation

right rotation

left rotation