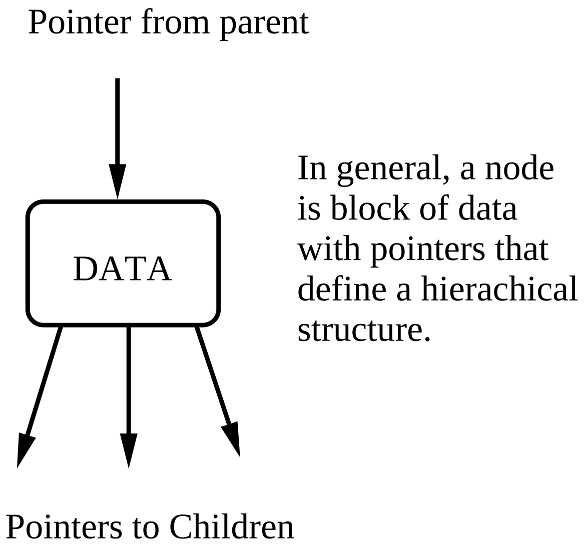
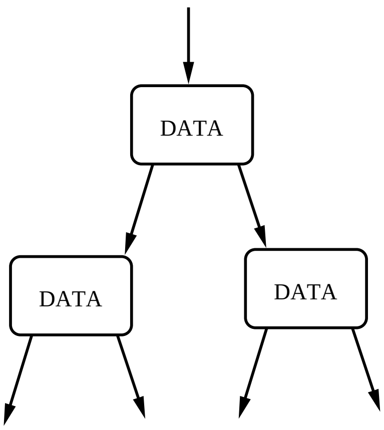
1 Trees and Huffman Coding

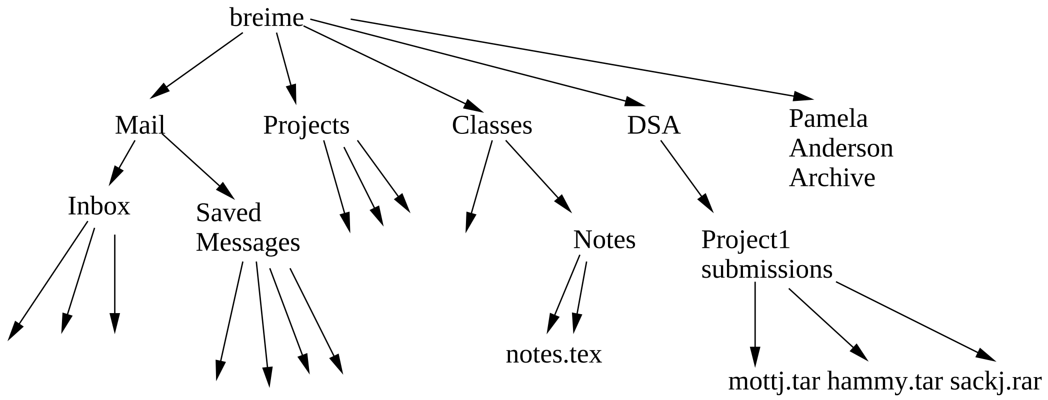
Trees are defined recursively. They are a collection of nodes.



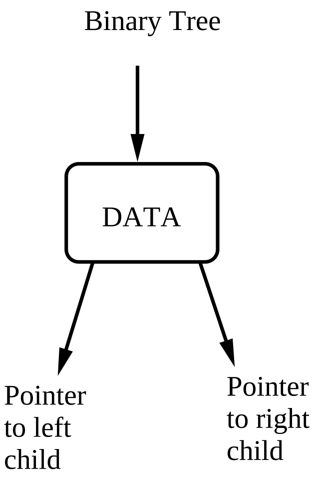
Nodes are combined to form a hierachical structure. Trees are structures for representing hierachical relationships.



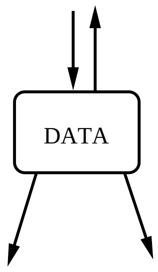
File/directory structure is a good example of a tree-based hierachical structure.



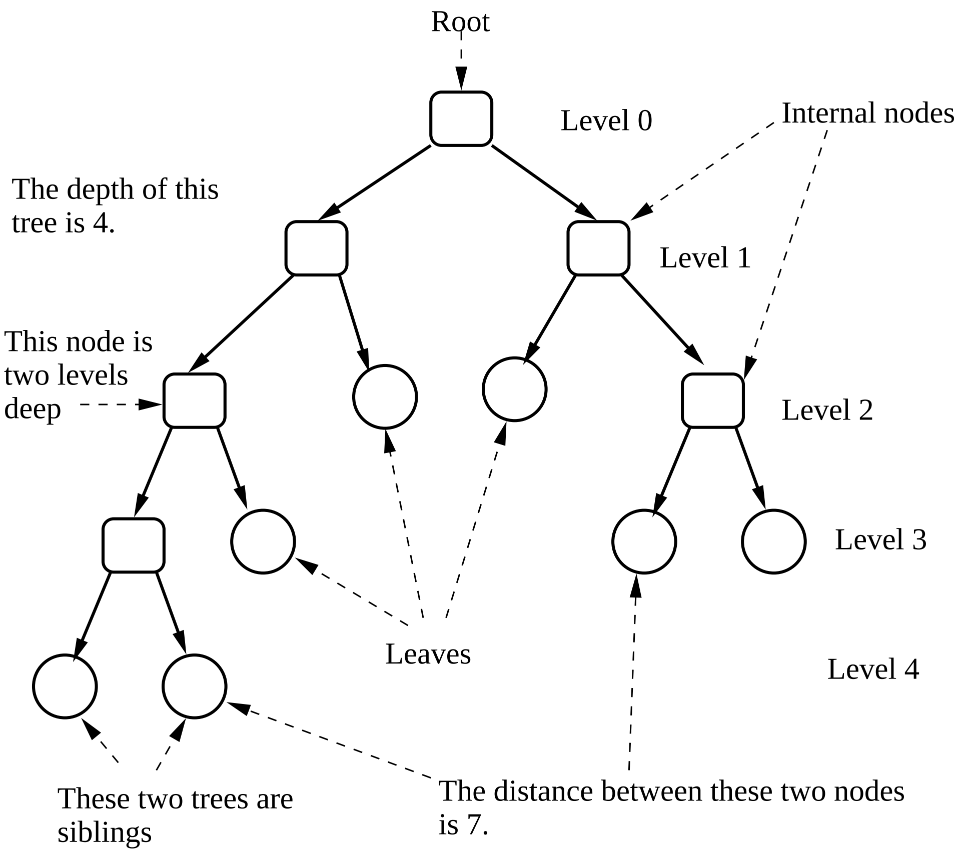
Binary Trees are restricted trees where each node has exactly two pointers to children. (left child and right child)



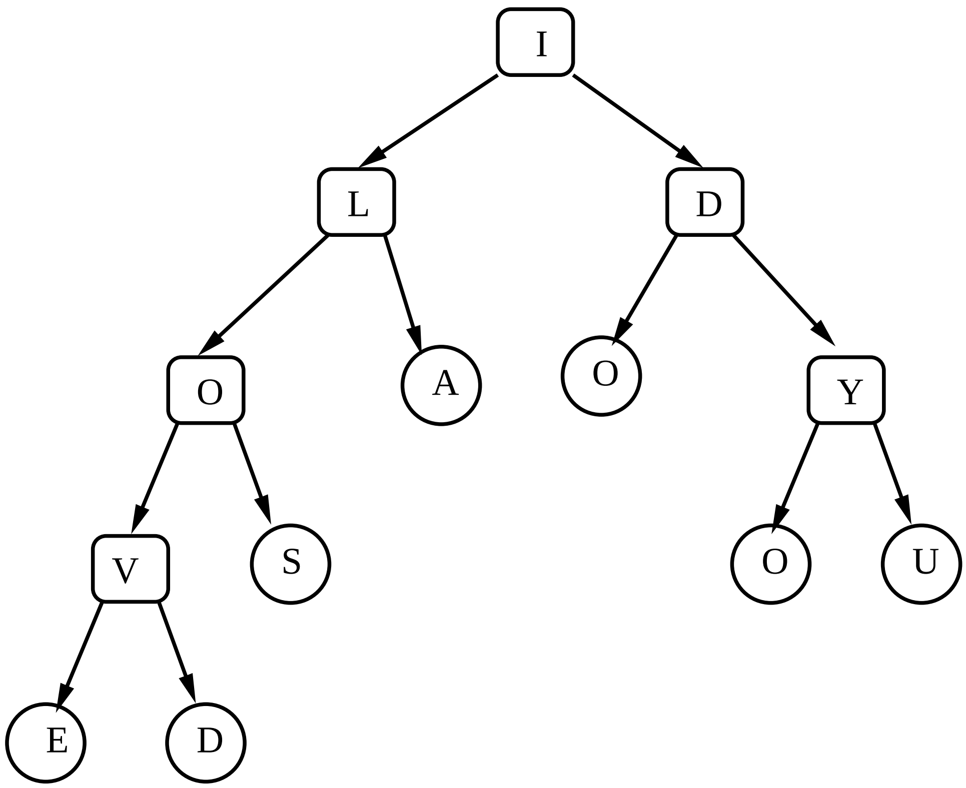
Trees can also have back pointers to parents.



Tree terminology.



Tree traversal:



Pre-order Depth-first Traversal:
I L O V E D S A D O Y O U

Post-order Depth-first Traversal:
E D V S O A L O O U Y D I

Breath-first Traversal:
I L D O A O Y V S O U E D

For project 2, you'll need to implement a binary tree to do huffman coding. Here are some of the specific details and hints to help you implement your binary tree:

```
// T is the data type stored in a node.
template <class T>
class binary_tree {
public:
  binary_tree();           // Default constructor

  // constructs a tree with one node.
  binary_tree(const T &);

  class iterator;  // Tree iterator

  T & root();  // Returns the data in the root

  // Combines two binary tree
  void combine(binary_tree &, binary_tree &);

  iterator begin() const;
  iterator end() const;

  // I've left out some stuff so don't freak out about it!
```

Why is root important?
Why do we need combine?
A great challenge is to write the combine tree.

The data type is a template parameter.

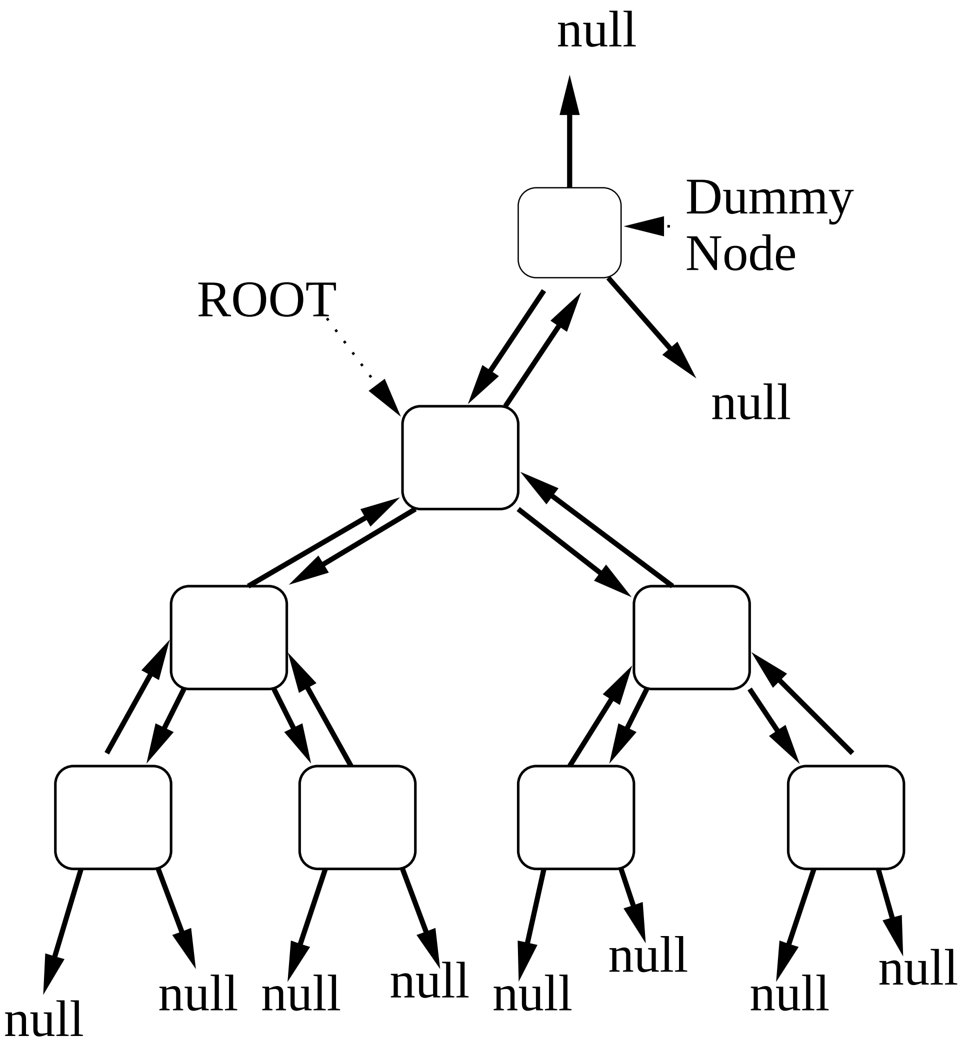
For project 2, your data type can be a pair (frequency (int) and letter (unsigned char)). Or you could make your own class.

```
private:
  struct node
  {
    T data;
    node * lt, * rt, * pt;

  };
  node * dummy_node;
```

Why do I have a dummy node?

I need a pointer to the root of the tree. Dummy node serves as that pointer. Also, dummy node is used to help determine if a tree is empty.


null
Dummy
Node
ROOT
null
null
null
null
null
null
null
null
null

If the tree is empty, both begin() and end() will point to the dummy node. Otherwise, begin() will point to the root. end() always points to the dummy node. Besides indicating whether the tree is empty or full, end() and begin() are useless. The reason we have them is because we want to preserve STL conventions. The binary tree is a container, and all containers have a begin() and an end().

```
template <class T>
inline binary_tree<T>::iterator
binary_tree<T>::begin() const
{
  if (dummy_node->lt)
    return iterator(dummy_node->lt);
  else
    return iterator(dummy_node);
}

template <class T>
inline binary_tree<T>::iterator
binary_tree<T>::end() const
{
  return iterator(dummy_node);
}
```

You are going to need an iterator for the tree. It should be nested inside of the tree class (as given in the description). Notice that this iterator is merely a "wrapper" for a node pointer. Hopefully, you can see how iterators can be used to define strict control over a pointer and how iterators can add functionality that is specific to the container (for example, left, right and parent define a kind of iteration that is designed specifically for binary tree).

```
public:
  friend class iterator;
  class iterator
  {
  public:
    friend class binary_tree;
    iterator() : ptr(0) {}
    T & operator*() const { return ptr->data; }
    iterator left() const { return iterator(ptr->lt); }
    iterator right() const { return iterator(ptr->rt); }
    iterator parent() const { return iterator(ptr->pt); }
    bool operator!=(const iterator & i) const {}
    bool operator==(const iterator & i) const {}
    bool is_root() const { return ptr->pt->pt == 0; }
    bool is_leaf() const {}
  private:
    iterator(node * n) : ptr(n) {}
    node * ptr;
  };
};
```

Finally, lets take a look at root and the constructors. Note that everything depends on the fact that I'm using the dummy node as the pointer to the root. Notice that the root is always the left child of the dummy node. This is a perfect example of the kind of implementation details that we want hide/encapsulate.

```
template <class T>
inline T & binary_tree<T, A>::root() {
  return dummy_node->lt->data;
}

template <class T>
binary_tree<T>::binary_tree() : n_elems(0) {
  dummy_node = new node;
  dummy_node->lt = dummy_node->rt = dummy_node->pt = 0;
}

template <class T>
binary_tree<T>::binary_tree(const T & x) {
  dummy_node = new node;
  dummy_node->rt = dummy_node->pt = 0;
  node * n = new node;
  n->data = x;
  n->lt = n->rt = 0;
  n->pt = dummy_node;
  dummy_node->lt = n;
}
```

Traversing the tree using the iterators:

```
void
huffman_codec::write_tree(std::ostream & o,
                          tree_type::iterator i,
  tree_type::iterator stop)
{
  if (i.is_leaf())
  {
    o << ".";
    o.put((*i).second);
    return;
  }
  else
  {
    o << "(";
    write_tree(o, i.left(), stop);
    o << ")(";
    write_tree(o, i.right(), stop);
    o << ")";
  }
}
```

How do you compare two trees inside the priority queue? You need to pass the comparator function into the priority queue. You can't just define a comparison operator for your tree type and expect the priority queue to use it.

First, the way you want to order your trees is specific to huffman coding and you don't want your generic tree to depend on the data type and application usage.

Second, the priority queue contains pointers to trees (not the trees themselves). Essentially, the priority queue is a queue of pointers. The only thing the queue cares about is the rule for ordering these pointers. You must supply this rule; If you don't supply the rule properly, then the queue will use the comparison operator of the data type. However, the data type of the queue is a pointer. So how would you expect the data to be ordered?

```
// We want small frequencies to be higher
// priority so use greater than operator
// in the comparision function object.
struct pq_comparator
{
  bool operator()(tree_type * t1, tree_type * t2)
  {
    return t1->root().first > t2->root().first;
  }
};
```