why templated nodes?

- common code “abstracted out”
  - same code written several times: don’t!
  - write a function, call it twice, rewrite code, why?

```cpp
struct STree
{
    string info;
    STree * left;
    STree * right;
};

struct ITree
{
    int info;
    ITree * left;
    ITree * right;
};

template <class T>
struct Tree
{
    T info;
    Tree * left;
    Tree * right;
};

Tree<string> * t = new Tree<string>;
int numNodes(Tree<string> * t);

template <class T>
int numNodes(Tree<T> * t); // hard to count?
```
Tree functions, vocabulary

- trees have roots, nodes, leaves, branches/edges
- the depth of a node is the number of edges on the path from the root to the node
  - single node tree?
  - empty tree?
  - height of a node is depth of deepest leaf below
- what’s good about trees? when are trees used? what are practical examples?
  - files/directories (almost)
  - expression trees
  - search structures
Tree traversals

```cpp
void Inorder(Tree * t)
{
    if (t != 0)
    {
    Inorder(t->left);
    cout << t->info << endl;
    Inorder(t->right);
    }
}

void Preorder(Tree * t)
{
    if (t != 0)
    {
    cout << t->info << endl;
    Preorder(t->left);
    Preorder(t->right);
    }
}

void Postorder(Tree * t)
{
    if (t != 0)
    {
    Postorder(t->left);
    Postorder(t->right);
    cout << t->info << endl;
    }
}
```
Tree shapes

- “good” trees are roughly balanced

- “bad” trees are long, skewed, skinny

- what is complexity of inserting n nodes into a “good” tree? why? a bad tree? why?
sidebar: solving recurrence

\[ T(n) = 2T(n/2) + O(n) \]
\[ T(0) = 1 \]

\[ T(n) = 2 \left[ 2T(n/4) + n/2 \right] + n \]
\[ = 4 \left[ T(n/4) + n \right] + n \]
\[ = 4 \left[ 2T(n/8) + n/4 \right] + 2n \]
\[ = 8T(n/8) + 3n \]
\[ = \ldots \text{eureka!} \]
\[ = 2^k T(n/2^k) + kn \]

let \( 2^k = n \)

\[ k = \log n, \text{ this yields } 2^{\log n} T(n/2^{\log n}) + n(\log n) \]
\[ n T(1) + n(\log n) \]
\[ O(n \log n) \]
sidebar: why observers?

- separate simulation from observation
  - in general, separation yields more flexibility
  - plug-in new observers later, or no observers

- each class should do one thing, not several
  - avoid “kitchen-sink” classes
  - make [it] as simple as possible, but no simpler

- inheritance facilitates re-use of an interface
  - implement in different ways, use in same way