February 6 – Iterators and Trees
G Carl Evans
Iterators encapsulated access to our data:
Iterators encapsulated access to our data:

<table>
<thead>
<tr>
<th>Get Data</th>
<th>Next Element</th>
<th>Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>*it</td>
<td>++it</td>
<td>it1 != it2</td>
</tr>
</tbody>
</table>
```cpp
#include <list>
#include <string>
#include <iostream>

struct Animal {
    std::string name, food;
    bool big;
    Animal(std::string name = "blob", std::string food = "you", bool big = true) :
        name(name), food(food), big(big) { /* nothing */ }
};

int main() {
    Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
    std::vector<Animal> zoo;
    zoo.push_back(g);
    zoo.push_back(p);  // std::vector's insertAtEnd
    zoo.push_back(b);
    for (std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it) {
        std::cout << (*it).name << " " << (*it).food << std::endl;
    }
    return 0;
}
```
```cpp
#include <list>
#include <string>
#include <iostream>

struct Animal {
    std::string name, food;
    bool big;
    Animal(std::string name = "blob", std::string food = "you", bool big = true) :
        name(name), food(food), big(big) { /* nothing */ }
};

int main() {
    Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
    std::vector<Animal> zoo;
    zoo.push_back(g);
    zoo.push_back(p);   // std::vector's insertAtEnd
    zoo.push_back(b);
    for ( auto it = zoo.begin(); it != zoo.end(); ++it ) {
        std::cout << (*it).name << " " << (*it).food << std::endl;
    }
    return 0;
}
```
```cpp
#include <list>
#include <string>
#include <iostream>

struct Animal {
    std::string name, food;
    bool big;

    Animal(std::string name = "blob", std::string food = "you", bool big = true) :
        name(name), food(food), big(big) { /* none */ }
};

int main() {
    Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
    std::vector<Animal> zoo;

    zoo.push_back(g);
    zoo.push_back(p);  // std::vector's insertAtEnd
    zoo.push_back(b);

    for ( const Animal & animal : zoo ) {
        std::cout << animal.name << " " << animal.food << std::endl;
    }

    return 0;
}
```
For Each and Iterators

```cpp
std::vector<Animal> zoo;
...
for ( const Animal & animal : zoo ) {
    std::cout << animal.name << " " << animal.food << std::endl;
}
```
Trees

“The most important non-linear data structure in computer science.”
- David Knuth, The Art of Programming, Vol. 1

A tree is:

- 
- 
-
More Specific Trees

We’ll focus on **binary trees:**

- A binary tree is **acyclic** – there are no cycles within the graph
More Specific Trees

We’ll focus on **binary trees**: 

- A binary tree contains **two or fewer children** – where one is the “left child” and one is the “right child”:

```
  a
 / \
c   b
 / \ / \
e f d g h
  
```

```
  a
 / \
c   b
 /   
  c   d
 / \  /  
e f g  
  
```

```
  a
 / \  
c   b  
 /  /  
c d f g
/  /  /
e f g h
```
Binary Tree – Defined

A binary tree $T$ is either:

• 

OR

•
Tree Property: height

height(T): length of the longest path from the root to a leaf

Given a binary tree T:

height(T) =
Tree Property: full

A tree $F$ is **full** if and only if:

1. 

2. 

![Diagram of a full tree]
Tree Property: perfect

A **perfect** tree $P$ is defined in terms of the tree’s height.

Let $P_h$ be a perfect tree of height $h$, and:

1. 
2. 

![Diagram of a perfect tree with nodes labeled C, S, X, A, 2, 2, 5.](image)
Tree Property: complete

**Conceptually:** A perfect tree for every level except the last, where the last level is “pushed to the left”.

**Slightly more formal:** For all levels $k$ in $[0, h-1]$, $k$ has $2^k$ nodes. For level $h$, all nodes are “pushed to the left”.
Tree Property: complete

A **complete** tree $C$ of height $h$, $C_h$:
1. $C_{-1} = \{\}$
2. $C_h (\text{where } h>0) = \{r, T_L, T_R\}$ and either:
   
   $$T_L \text{ is } \_\_\_\_\_\_\_\text{ and } T_R \text{ is } \_\_\_\_\_\_\_$$
   
   OR

   $$T_L \text{ is } \_\_\_\_\_\_\_\text{ and } T_R \text{ is } \_\_\_\_\_\_\_$$

![Tree Diagram]

1. $C_{-1} = \{\}$
2. $C_h (\text{where } h>0) = \{r, T_L, T_R\}$ and either:
   
   $$T_L \text{ is } \_\_\text{ and } T_R \text{ is } \_\_$$
   
   OR

   $$T_L \text{ is } \_\_\text{ and } T_R \text{ is } \_\_$$

![Tree Diagram]
Tree Property: complete

Is every **full** tree **complete**?

If every **complete** tree **full**?
Tree ADT
Tree ADT

**insert**, inserts an element to the tree.

**remove**, removes an element from the tree.

**traverse**, 
```cpp
#pragma once

template <class T>
class BinaryTree {
    public:
        /* ... */

    private:

};
```
Trees aren’t new:
Trees aren’t new:
How many NULLs?

**Theorem:** If there are $n$ data items in our representation of a binary tree, then there are ___________ NULL pointers.
How many NULLs?

Base Cases:

n = 0:

n = 1:

n = 2:
How many NULLs?

Induction Hypothesis:
How many NULLs?

Consider an arbitrary tree $T$ containing $n$ data elements:
Traversals
template<class T>
void BinaryTree<T>::__Order(TreeNode * root)
{
  if (root != NULL) {
    ________________________;
    __Order(root->left);
    ________________________;
    ________________________;
    __Order(root->right);
    ________________________;
  }
}
Traversals

```
template<class T>
void BinaryTree<T>::__Order(TreeNode * root)
{
    if (root != NULL) {
        ____________________;
        __Order(root->left);
        ____________________;
        ____________________;
        __Order(root->right);
        ____________________;
    }
}
```
template<class T>
void BinaryTree<T>::__Order(TreeNode * root)
{
    if (root != NULL) {
        __Order(root->left);
        __Order(root->right);
    }
}