Iterators

Suppose we want to look through every element in our data structure:
Iterators encapsulated access to our data:

<table>
<thead>
<tr>
<th>Cur. Location</th>
<th>Cur. Data</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListNode *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>index</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x, y, z)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Iterators

Every class that implements an iterator has two pieces:

1. [Implementing Class]:
Iterators

Every class that implements an iterator has two pieces:

2. [Implementing Class’ Iterator]:
   • Must have the base class: `std::iterator`

   • `std::iterator` requires us to minimally implement:
Iterators encapsulated access to our data:

```
::begin
::end
```
```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) { /* 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
for ( const TYPE & variable : collection ) {
    // ...
}
```

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

```cpp
for ( const TYPE & variable : collection ) {
    // ...
}
```

```cpp
14 std::vector<Animal> zoo;
... ...
20 for ( const Animal & animal : zoo ) {
21    std::cout << animal.name << " " << animal.food << std::endl;
22 }
```

```cpp
... std::multimap<std::string, Animal> zoo;
... ...
20 for ( const Animal & animal : zoo ) {
21    std::cout << animal.name << " " << animal.food << std::endl;
22 }
```
Exam Programming A

• 2 hours
• 1 code reading question; 2 POTD-like coding questions
• Topics: see course website
  • C++, List implementations (linked list, array), Stack/Queue ADT
  • Labs: lab_intro, lab_debug, lab_memory, lab_inheritance
  • MP1 and MP2

Be sure you know how to do POTDs from EWS Linux machines !!!
Honors Section

CS 225 offers a one-credit add on honors section!

What is data science?

- Algorithms
- Data Structures
- Python
- pandas
- Visualizations
- JavaScript
- d3.js
Honors Section

Course Starts: **Tomorrow**, Thursday, February 14, 2019

**Meets:** Thursdays: 5:00 – 5:50pm, 1404 Siebel Center

If you are interested in adding the course, come to the first lecture!

**Taught By:** Wade Fagen-Ulmschneider (CS faculty)

**Open to EVERYONE** – not required to be part of an honors program. Fulfills HCLA, James Scholar, etc.
Trees

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

A tree is:

• 
• 
•
A Rooted Tree

“Mario Family Line”
<http://limitbreak.gameriot.com/blogs/Caveat-Emptor/Mario-Family-Line>
More Specific Trees

We’ll focus on **binary trees**:  
- A binary tree is **rooted** – every node can be reached via a path from the root
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
 / \    |
i   h   j
```
Tree Terminology

• Find an edge that is not on the longest path in the tree. Give that edge a reasonable name.
• One of the vertices is called the root of the tree. Which one?
• Identify the vertices that have a parent but no sibling.
• How many parents does each vertex have?
• Which vertex has the fewest children?
• Which vertex has the most ancestors?
• Which vertex has the most descendants?
• List all the vertices in b’s left subtree.
• List all the leaves in the tree.
Binary Tree – Defined

A *binary tree* $T$ is either:

1. $\texttt{null}$

2. OR

3. $\{\texttt{node}\} \rightarrow \texttt{left} \rightarrow \texttt{right}$
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. 
Tree Property: perfect

A perfect tree $P$ is:

1. 
2. 

\[ \begin{align*}
\text{C} & \quad \text{S} \\
& \quad \text{X} \\
\text{A} & \quad 2 \\
\ & \quad 2 \\
\ & \quad 5
\end{align*} \]
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 any level $k$ in $[0, h-1]$, $k$ has $2^k$ nodes. For level $h$, all nodes are “pushed to the left”.

![Tree Diagram]

- **C**
- **S**
- **X**
- **A**
- **2**
- **2**
- **5**
- **Y**
- **Z**
Tree Property: complete

A complete tree $C$ of height $h$, $C_h$:
1. $C_{-1} = {}$
2. $C_h \ (where \ h>0) = \{r, T_L, T_R\}$ and either:
   - $T_L$ is __________ and $T_R$ is __________
   - OR
   - $T_L$ is __________ and $T_R$ is __________
Tree Property: complete
Is every **full** tree **complete**?

If every **complete** tree **full**?