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

```plaintext
Cur. Location  Cur. Data  Next

ListNode *

index

(x, y, z)
```
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:
```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
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
std::vector<Animal> zoo;
...
for ( const Animal & animal : zoo ) {
    std::cout << animal.name << " " << animal.food << std::endl;
}
```

```cpp
std::unordered_set<std::string, 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 **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”:
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?

• 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:

- OR

-
Tree Property: height

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

Given a binary tree $T$:

*height($T$) =

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

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

1. 
2. 

![Diagram of a full tree](image-url)
Tree Property: perfect

A **perfect** tree $P$ is:

1.

2.
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”.

```
  C
 / \
S   X
 /|
A 2 2
 /|
Y  Z
 /|
5
```
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**?