# Data Structures

# Stacks and Queues

CS 225
Brad Solomon

September 11, 2024

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

# Learning Objectives

Introduce the stack and the queue data structure

Introduce and explore iterators

# List Implementation

| | Singly Linked List | Array |
|---|---|---|
| Look up **arbitrary** location | O(n) | O(1) |
| Insert after **given** element | O(1) | O(n) |
| Remove after **given** element | O(1) | O(n) |
| Insert at **arbitrary** location | O(n) | O(n) |
| Remove at **arbitrary** location | O(n) | O(n) |
| Search for an input **value** | O(n) | O(n) |

Special Cases:

# Thinking critically about lists: tradeoffs

As we progress in the class, we will see that $O(n)$ isn’t very good.

Take searching for a specific list value:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 7 | 5 | 9 | 7 | 14 | 1 | 0 | 8 | 3 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 5 | 7 | 7 | 8 | 9 | 14 |

# Thinking critically about lists: tradeoffs

Can we make a ‘list’ that is O(1) to insert and remove?

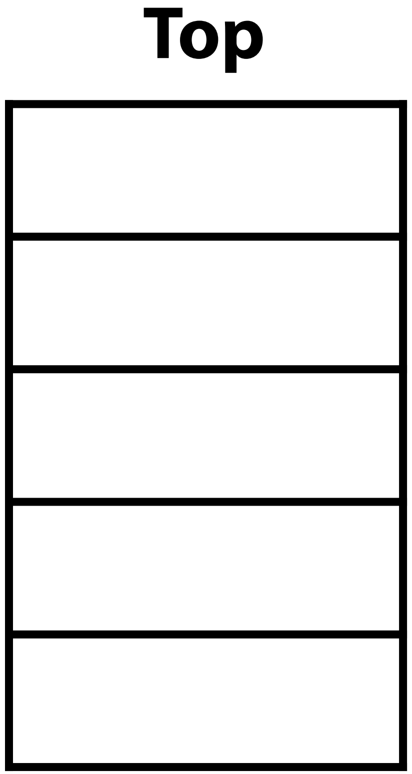
# Stack Data Structure

A **stack** stores an ordered collection of objects (like a list)

However you can only do two* operations:

**Push**: Put an item on top of the stack

**Pop**: Remove the top item of the stack (and return it)



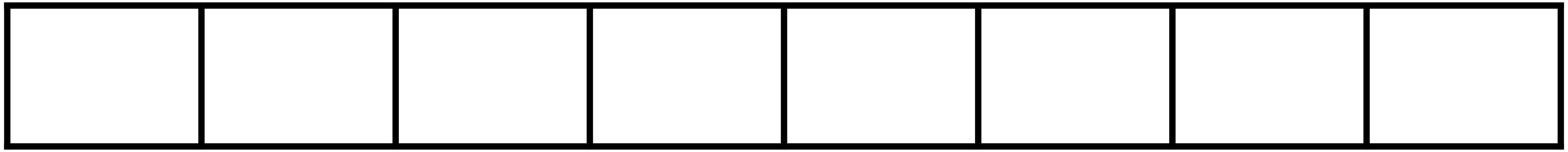

```
push(3); push(5); pop(); push(2)
```

# Stack Data Structure

C++ has a built-in stack

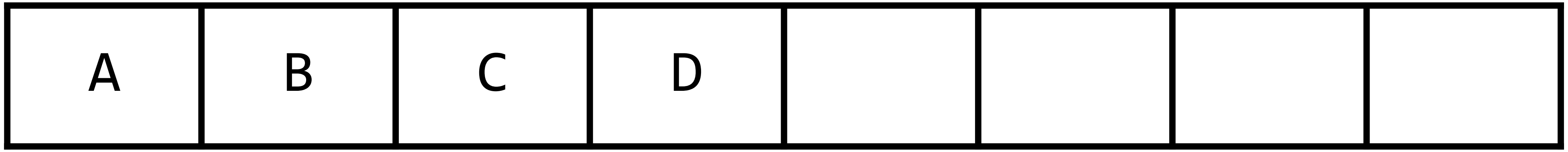
Underlying implementation is vector or deque

```
#include <stack>
int main() {
  stack<int> stack;
  stack.push(3);
  stack.push(8);
  stack.push(4);
  stack.pop();
  stack.push(7);
  stack.pop();
  stack.pop();
}
```



# Stack Data Structure

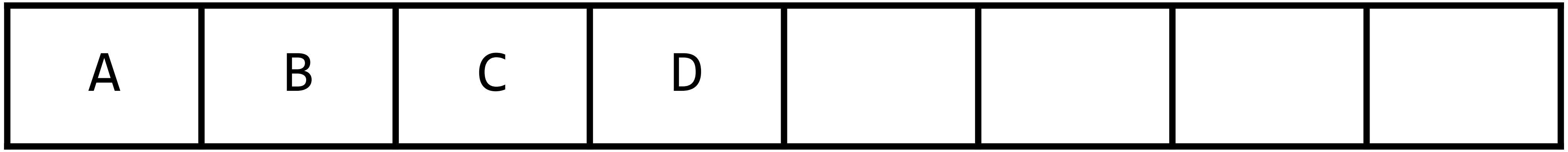
`Push(X)` is equivalent to …



# Stack Data Structure

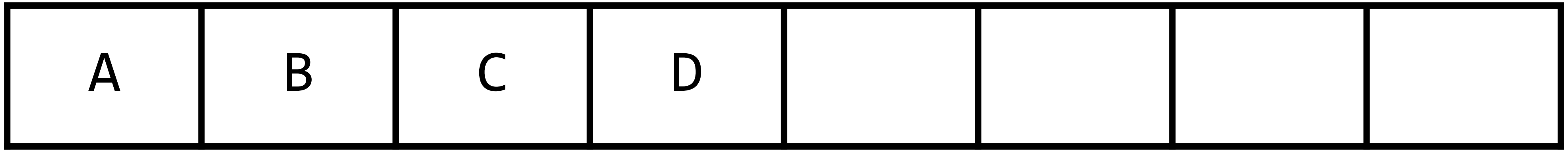
`Push(X)` is equivalent to `insertBack(X)`

```
*size = X;

size++;
```



# Stack Data Structure

Pop( ) is equivalent to…



# Stack Data Structure

`Pop()` is equivalent to `removeBack()`

```
size--;

T tmp = *size;

return tmp;
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | B | C | D | | | | |

# Stack ADT

- [Order]:

- [Implementation]:

- [Runtime]:

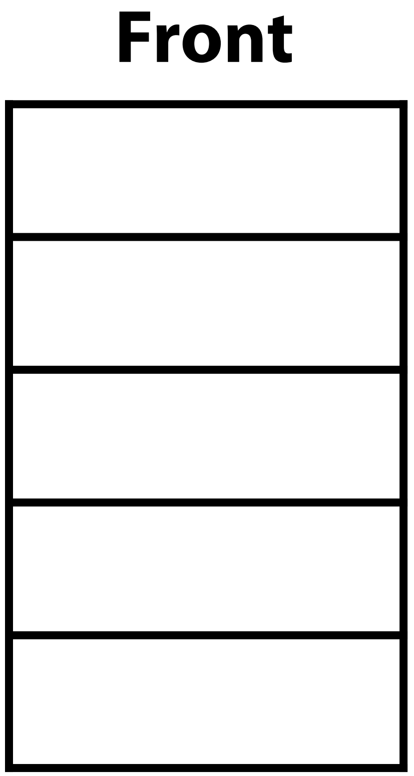
# Queue Data Structure

A **queue** stores an ordered collection of objects (like a list)

However you can only do two* operations:

**Enqueue**: Put an item at the back of the queue

**Dequeue**: Remove and return the front item of the queue




```
enqueue(3); enqueue(5); dequeue(); enqueue(2)
```

# Queue Data Structure

The queue is a **first in — first out** data structure (FIFO)

What data structure excels at removing from the front?

Can we make that same data structure good at inserting at the end?

# Queue Data Structure

The C++ implementation of a queue is also a vector or deque — why?

# Engineering vs Theory Efficiency

| | Time x1 billion | Like |
|---|---|---|
| L1 cache reference | 0.5 seconds | Heartbeat 💓 |
| Branch mispredict | 5 seconds | Yawn 😲 |
| L2 cache reference | 7 seconds | Long yawn 😲 😲 😲 |
| Mutex lock/unlock | 25 seconds | Make coffee ☕ |
| Main memory reference | 100 seconds | Brush teeth |
| Compress 1K bytes | 50 minutes | TV show 📺 |
| Send 2K bytes over 1 Gbps network | 5.5 hours | (Brief) Night's sleep 🛌 |
| SSD random read | 1.7 days | Weekend |
| Read 1 MB sequentially from memory | 2.9 days | Long weekend |
| Read 1 MB sequentially from SSD | 11.6 days | 2 weeks for delivery 📦 |
| Disk seek | 16.5 weeks | Semester |
| Read 1 MB sequentially from disk | 7.8 months | Human gestation 🐣 |
| Above two together | 1 year | 🌍 ☀️ |
| Send packet CA->Netherlands->CA | 4.8 years | Ph.D. 🎓 |

(Care of https://gist.github.com/hellerbarde/2843375)

# Engineering vs Theory Efficiency

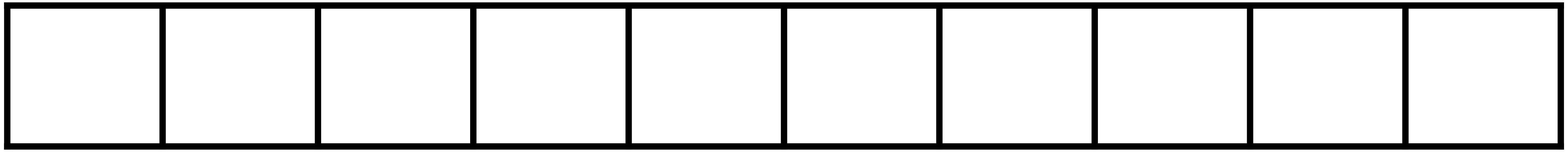
| | Time x1 billion | Like |
|---|---|---|
| L1 cache reference | 0.5 seconds | Heartbeat 💓 |
| Main memory reference | 100 seconds | Brush teeth |
| SSD random read | 1.7 days | Weekend |
| Disk seek | 16.5 weeks | Semester |
| Send packet CA->Netherlands->CA | 4.8 years | Ph.D. 🎓 |

(Care of https://gist.github.com/hellerbarde/2843375)

# Queue Data Structure

```
q.enqueue(8);
q.enqueue(4);
q.dequeue();
```

What do we need to track to maintain a queue with an array list?

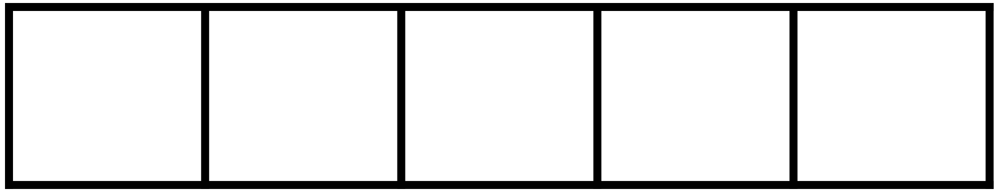


# Queue Data Structure

Unlike the array list, it is easier to implement a Queue using unsigned ints

**Queue.h**

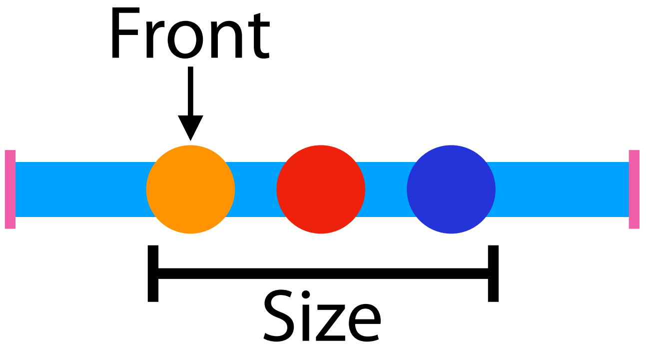
```
#pragma once

template <typename T>
class Queue {
  public:
    void enqueue(T e);
    T dequeue();
    bool isEmpty();

  private:
    T *data_;
    unsigned size_;
    unsigned capacity_;
    unsigned front_;
};
```

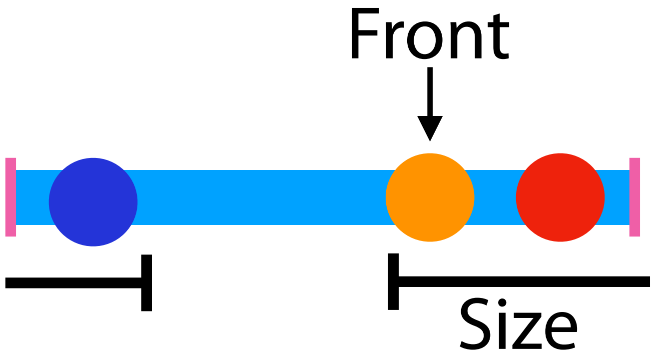


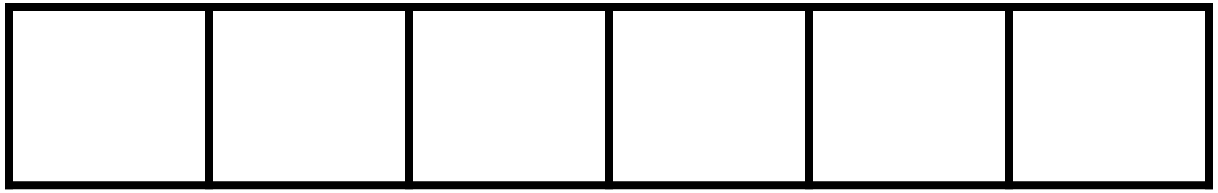
# (Circular) Queue Data Structure

**Queue.h**

```
#pragma once

template <typename T>
class Queue {
  public:
    void enqueue(T e);
    T dequeue();
    bool isEmpty();

  private:
    T *data_;
    unsigned capacity_;
    unsigned size_;
    unsigned front_;
};
```





```
Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue();
q.enqueue(9);
```

Enqueue(D):

Dequeue():

Size:

Front:

Capacity:

# Queue Data Structure: Resizing

| | | | | | |
|---|---|---|---|---|---|
| | | m | o | n | |

```
Queue<char> q;
...
q.enqueue(d);
q.enqueue(a);
q.enqueue(y);
q.enqueue(i);
q.enqueue(s);
```

# Queue Data Structure: Resizing

```
Queue<char> q;
...
q.enqueue(d);
q.enqueue(a);
q.enqueue(y);
q.enqueue(i);
q.enqueue(s);
```

| | | | | | |
|---|---|---|---|---|---|
| | | m | o | n | |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | |

# Queue ADT

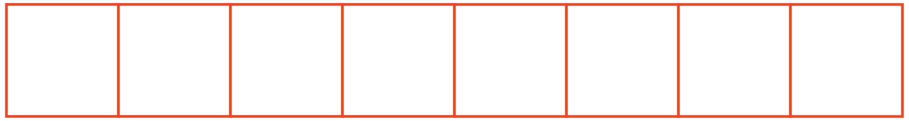
- [Order]:

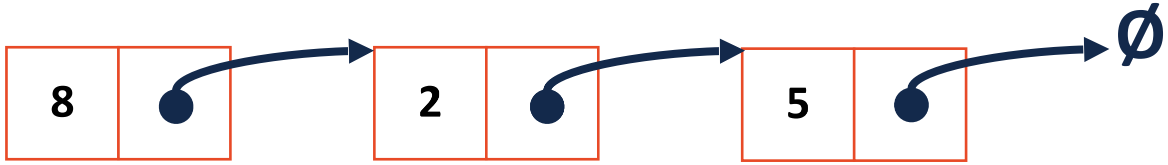
- [Implementation]:

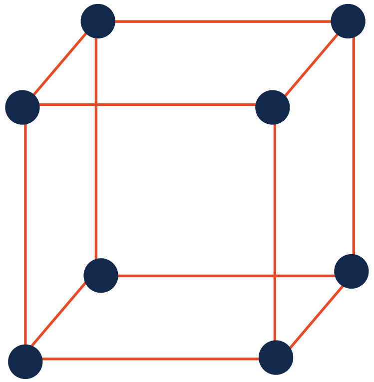
- [Runtime]:

# Iterators

We want to be able to loop through all elements for any underlying implementation in a systematic way

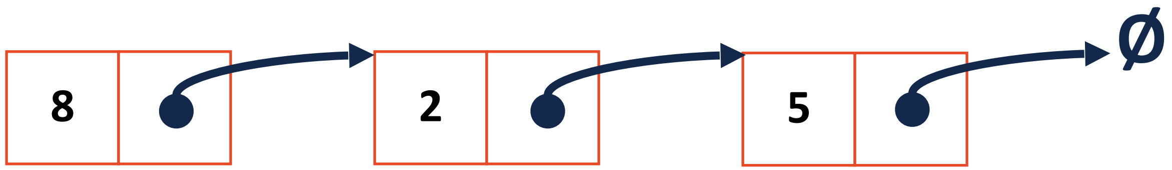


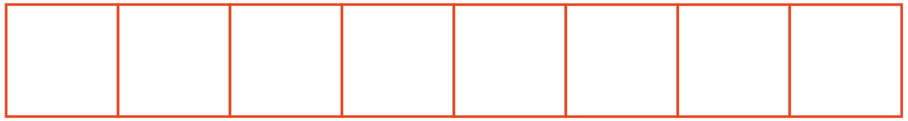


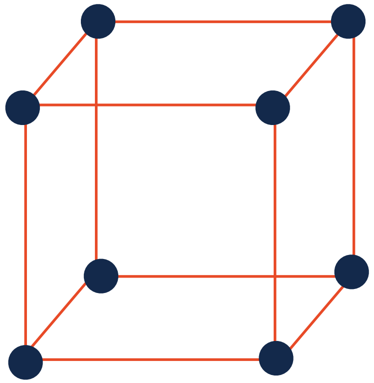



# Iterators

We want to be able to loop through all elements for any underlying implementation in a systematic way



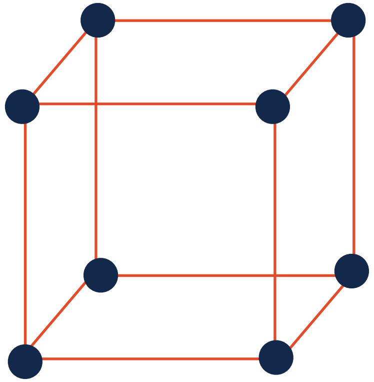





| Cur. Location | Cur. Data | Next |
| --- | --- | --- |
| `ListNode * curr` | | |
| `unsigned index` | | |
| `Some form of (x, y, z)` | | |

# Iterators

Iterators provide a way to access items in a container without exposing the underlying structure of the container



```
Cube::Iterator start = myCube.begin();

while (it != myCube.end()) {
     std::cout << *it << " ";
     it++;
}

```

# Iterators

For a class to implement an iterator, it needs two functions:

```
Iterator begin()
```

```
Iterator end()
```

# Iterators

The actual iterator is defined as a class **inside** the outer class:

1. It must be of base class **`std::iterator`**

2. It must implement at least the following operations:

```
Iterator& operator ++()

const T & operator *()

bool operator !=(const Iterator &)
```

# Iterators

Here is a (truncated) example of an iterator:

```
template <class T>
class List {

    class ListIterator : public std::iterator<std::bidirectional_iterator_tag, T> {
      public:

        ListIterator& operator++();

        ListIterator& operator--()

        bool operator!=(const ListIterator& rhs);

        const T& operator*();
    };

    ListIterator begin() const;

    ListIterator end() const;
};
```

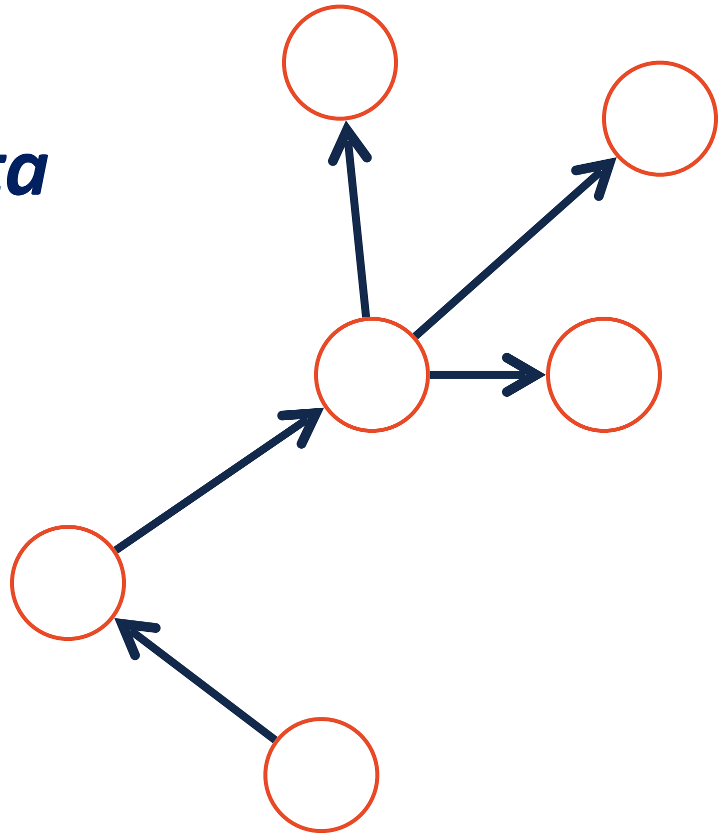
```cpp
#include <list>
#include <string>
#include <iostream>

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;
}
```

```
std::vector<Animal> zoo;


/* Full text snippet */

  for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
    std::cout << (*it).name << " " << (*it).food << std::endl;
  }


/* Auto Snippet */

  for ( auto it = zoo.begin(); it != zoo.end; ++it ) {
    std::cout << animal.name << " " << animal.food << std::endl;
  }

/* For Each Snippet */

  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:**

- 
- 