CS 225
Data Structures

September 24 – Iterators
Wade Fagen-Ulmschneider
List ADT

• Linked Memory Implementation ("Linked List")
  • O(1) insert/remove at front/back
  • O(1) insert/remove after a given element
  • O(n) lookup by index

• Array Implementation ("Array List")
  • O(1) insert/remove at front/back
  • O(n) insert/remove at any other location
  • O(1) lookup by index
Queue ADT

• [Order]:

• [Implementation]:

• [Runtime]:
Stack ADT

• [Order]:

• [Implementation]:

• [Runtime]:
#pragma once

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

    private:
        T *items_;
        unsigned capacity_;  
        unsigned count_; 
};
What type of implementation is this Queue?

How is the data stored on this Queue?

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);
#pragma once

template <typename T>
class Queue {
    public:
        void enqueue(T e);
        T dequeue();
        bool isEmpty();
    private:
        T *items_;  // T *items_;  // T *items_;
        unsigned capacity_;  // unsigned capacity_;  // unsigned capacity_
        unsigned count_;  // unsigned count_;  // unsigned count_
};

Queue<char> q;
q.enqueue(m);
q.enqueue(o);
q.enqueue(n);
...
q.enqueue(d);
q.enqueue(a);
q.enqueue(y);
q.enqueue(i);
q.enqueue(s);
q.dequeue();
q.enqueue(h);
q.enqueue(a);
Implications of Design

1. class ListNode {
   public:
   T & data;
   ListNode * next;
   ...

2. class ListNode {
   public:
   T * data;    ... 

3. class ListNode {
   public:
   T data;     ...
## Implications of Design

<table>
<thead>
<tr>
<th></th>
<th>Storage by Reference</th>
<th>Storage by Pointer</th>
<th>Storage by Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Who manages the lifecycle of the data?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is it possible for the data structure to store NULL?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If the data is manipulated by user code while in our data structure, is the change reflected in our data structure?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Data Lifecycle

### Storage by reference:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cube c;</td>
</tr>
<tr>
<td>2</td>
<td>myStack.push(c);</td>
</tr>
</tbody>
</table>

### Storage by pointer:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cube c;</td>
</tr>
<tr>
<td>2</td>
<td>myStack.push(&amp;c);</td>
</tr>
</tbody>
</table>

### Storage by value:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cube c;</td>
</tr>
<tr>
<td>2</td>
<td>myStack.push(c);</td>
</tr>
</tbody>
</table>
Possible to store NULL?

Storage by reference:

class ListNode {
    public:
        T & data;
        ListNode * next;
    ListNode(T & data) : data(data), next(NULL) { }
};

Storage by pointer:

T ** arr;

Storage by value:

T * arr;
Data Modifications

```c
1 Cube c(1);
2 myStack.push(c);
3 c.setLength(42);
4 Cube r = myStack.pop();
5 // What is r's length?
```
Speed
Iterators

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

```
8 2 5
```

Ø
Iterators encapsulated access to our data:

- Cur. Location
- Cur. Data
- Next

<table>
<thead>
<tr>
<th>Cur. Location</th>
<th>Cur. Data</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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`
   • Must implement
     `operator*`
     `operator++`
     `operator!=`
#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;
}
```