CS 225

Data Structures

Feb. 12 – Iterators

Wade Fagen-Ulmschneider

CS 225 So Far…

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

CS 225 So Far…

Queue ADT

- FIFO: First in, first out – like a line/queue at a shop
- Implemented with a list, O(1) enqueue/dequeue

Stack ADT

- LIFO: Last in, first out – list a stack of papers
- Implemented with a list, O(1) push/pop

Queue.h

```
 1 #ifndef QUEUE_H
 2 #define QUEUE_H
 3
 4 template <class QE>
 5 class Queue {
 6   public:
 7     void enqueue(QE e);
 8     QE dequeue();
 9     bool isEmpty();
10
11   private:
12     QE *items_;
13     unsigned capacity_;
14     unsigned count_;
15
16
17
18 };
19
20 #endif
21
22
```

What type of implementation is this Queue?

How is the data stored on this Queue?

Queue.h

```
 1  #ifndef QUEUE_H
 2  #define QUEUE_H
 3
 4  template <class QE>
 5  class Queue {
 6    public:
 7      void enqueue(QE e);
 8      QE dequeue();
 9      bool isEmpty();
10
11    private:
12      QE *items_;
13      unsigned capacity_;
14      unsigned count_;
15
16
17
18  };
19
20  #endif
21
22
```

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);
```

Queue.h

```
 1  #ifndef QUEUE_H
 2  #define QUEUE_H
 3
 4  template <class QE>
 5  class Queue {
 6    public:
 7      Queue(); // …etc…
 8      void enqueue(QE e);
 9      QE dequeue();
10      bool isEmpty();
11
12    private:
13      QE *items_;
14      unsigned capacity_;
15      unsigned count_;
16      unsigned entry_;
17      unsigned exit_;
18
19  };
20
21  #endif
22
```


```
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.

```
struct ListNode {
  T & data;
  ListNode * next;
  ListNode(T & data) : data(data), next(NULL) { }
};
```

2.

```
struct ListNode {
  T * data;
  …
```

3.

```
struct ListNode {
  T data;
  …
```

Implications of Design

	Storage by Reference	Storage by Pointer	Storage by Value
Who manages the lifecycle of the data?			
Is it possible for the data structure to store NULL?			
If the data is manipulated by user code while in our data structure, is the change reflected in our data structure?			
Speed			

Data Lifecycle

Storage by reference:

```
1 Sphere s;
2 myStack.push(s);
```

Storage by pointer:

```
1 Sphere s;
2 myStack.push(&s);
```

Storage by value:

```
1 Sphere s;
2 myStack.push(s);
```

Possible to store NULL?

Storage by reference:

```
struct ListNode {
  T & data;
  ListNode * next;
  ListNode(T & data) : data(data), next(NULL) { }
};
```

Storage by pointer:

```
T ** arr;
```

Storage by value:

```
T * arr;
```

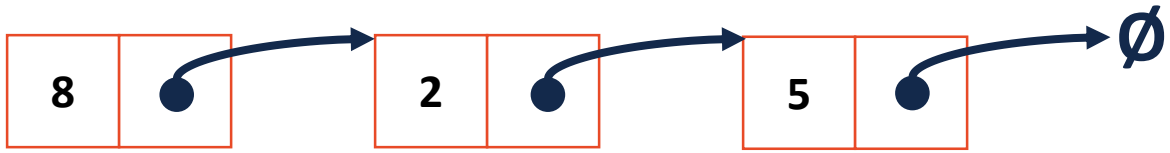
Data Modifications

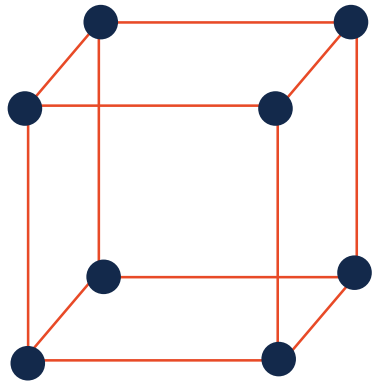
```
1  Sphere s(1);
2  myStack.push(s);
3
4  s.setRadius(42);
5
6  Sphere r = myStack.pop();
7  // What is r's radius?
```

Speed

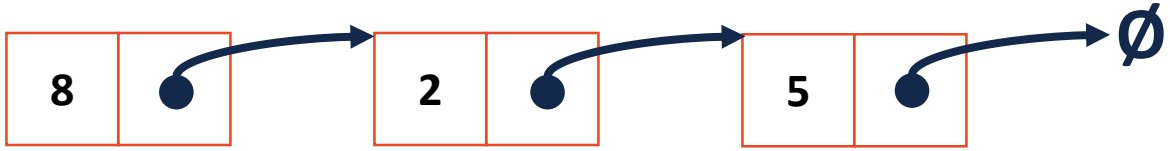
Iterators

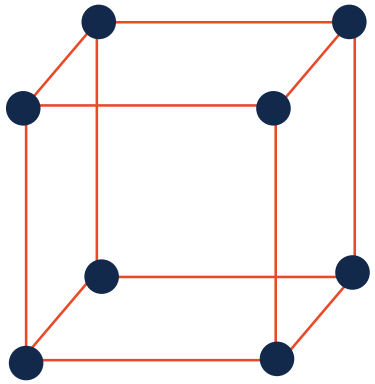
Suppose we want to look through every element in our data structure:





Iterators encapsulated access to our data:





Cur. Location	Cur. Data	Next

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!=

stlList.cpp

```
 1 #include <list>
 2 #include <string>
 3 #include <iostream>
 4
 5 struct Animal {
 6   std::string name, food;
 7   bool big;
 8   Animal(std::string name = "blob", std::string food = "you", bool big = true) :
 9     name(name), food(food), big(big) { /* none */ }
10 }
11
12 int main() {
13   Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
14   std::list<Animal> zoo;
15
16   zoo.push_back(g);
17   zoo.push_back(p);   // std::list's insertAtEnd
18   zoo.push_back(b);
19
20   for ( std::list<Animal>::iterator it = zoo.begin(); it != zoo.end(); it++ ) {
21     std::cout << (*it).name << " " << (*it).food << std::endl;
22   }
23
24   return 0;
25 }
```

stlList.cpp

```
 1 #include <list>
 2 #include <string>
 3 #include <iostream>
 4
 5 struct Animal {
 6   std::string name, food;
 7   bool big;
 8   Animal(std::string name = "blob", std::string food = "you", bool big = true) :
 9     name(name), food(food), big(big) { /* none */ }
10 }
11
12 int main() {
13   Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
14   std::list<Animal> zoo;
15
16   zoo.push_back(g);
17   zoo.push_back(p);   // std::list's insertAtEnd
18   zoo.push_back(b);
19
20   for ( const Animal & animal : zoo ) {
21     std::cout << animal.name << " " << animal.food << std::endl;
22   }
23
24   return 0;
25 }
```