Announcements


Queues:

Queue ADT:
- enqueue
- dequeue
- isEmpty

<table>
<thead>
<tr>
<th>enqueue</th>
<th>dequeue</th>
<th>isEmpty</th>
</tr>
</thead>
</table>

Diagram of a queue ADT with enqueue, dequeue, and isEmpty operations.
Queue—linked memory based implementation:

```
template<class SIT>
class Queue {
public:
    // ctors dtor
    bool empty() const;
    void enqueue(const SIT & e);
    SIT dequeue();
private:
    struct queueNode {
        SIT data;
        queueNode * next;
    };
    queueNode * entry;
    queueNode * exit;
    int size;
};
```

Which pointer is “entry” and which is “exit”?

What is running time of enqueue?

What is running time of dequeue?
Queue array based implementation:

```cpp
template<class SIT>
class Queue {
public:
    Queue();
    ~Queue(); // etc.
    bool empty() const;
    void enqueue(const SIT & e);
    SIT dequeue();
private:
    int capacity;
    int size;
    SIT * items;
    // maybe some other stuff...
};
```

```cpp
template<class SIT>
Queue<SIT>::Queue(){
    capacity = 8;
    size = 0;
    items = new SIT[capacity];
}
```
Queue array based implementation:

template<class SIT>
class Queue {
public:
    Queue();
    ~Queue(); // etc.
    bool empty() const;
    void enqueue(const SIT & e);
    SIT dequeue();
private:
    int capacity;
    int size;
    SIT * items;
};
Queue array based implementation:

```cpp
template<class SIT>
class Queue {
public:
    Queue();
    ~Queue(); // etc.
    bool empty() const;
    void enqueue(const SIT & e);
    SIT dequeue();
private:
    int capacity;
    int size;
    SIT * items;
    int entry;
    int exit;
    // some other stuff...
};
```
What if array fills?:

entry

exit

d

\[\text{a y i s h o n d}\]

...
# include <list>
# include <iostream>
# include <string>
using namespace std;

struct animal {
  string name;
  string food;
  bool big;
  animal(string n="blob", string f="you", bool b=true):name(n),food(f),big(b) {} 
};

int main() {
  animal g("giraffe","leaves"), p("penguin","fish",false), b("bear");
  list<animal> zoo;
  zoo.push_back(g); zoo.push_back(p); zoo.push_back(b); //STL list insertAtEnd

  for(list<animal>::iterator it = zoo.begin(); it != zoo.end(); it++)
    cout << (*it).name << "  " << (*it).food << endl;

  return 0;
}
Suppose these familiar structures were encapsulated.

Iterators give client the access it needs to traverse them anyway!

Objects of type “iterator” promise to have at least the following defined:

- `operator++`
- `operator*`
- `operator!=`
- `operator==`
- `operator=`

“Container classes” typically have a variety of iterators defined within:

- `Forward`
- `Reverse`
- `Bidirectional`