Recap

- Abstract Data Types
  - Specification (What?)
  - Implementation (How?)
- Unsorted lists
  - Array-based implementation
  - Pointer-based implementation
- Questions
  - Reference vs. value type
  - Comparing enumerated types
- Exceptions (TODAY)

Lecture Plan

- Linked list implementation of unsorted list
- Time complexity of algorithms
  - Big Oh
ADT Unsorted List

• Transformers
  – MakeEmpty
  – InsertItem
  – DeleteItem

• Observers
  – IsFull
  – GetLength
  – RetrieveItem

• Iterators
  – ResetList
  – GetNextItem

---

Specification

// SPECIFICATION FILE ( unsortedType.h )
#include "ItemType.h"
struct NodeType;
class UnsortedType // declares a class data type
{
  public :
    // 8 public member functions
    UnsortedType(); // constructors
    void MakeEmpty();
    bool IsFull() const; // returns length of list
    int GetLength() const; // returns length of list
    void RetrieveItem( ItemType& item, bool& found );
    void InsertItem( ItemType item );
    void DeleteItem( ItemType item );
    void ResetList();
    void GetNextItem( ItemType& item );

---

Linked List Implementation (private part)

private:
  NodeType* listData;
  list length;
  NodeType* currentPos;
struct NodeType
{
  NodeType* next;
};
list
  length
  currentPos
  listData

List with two items

---
Linked List Implementation

How do you know that a linked list is empty?
listData is NULL

What should the constructor do?
Set length to 0
Set listData to NULL

What about currentPos?
We let ResetList take care of initializing currentPos

Linked List Implementation

What about the observers IsFull and GetLength?
GetLength just returns length

Can a linked list ever be full?
Yes, if you run out of memory
Ask for a new node within a try/catch

Linked List Implementation of IsFull

```cpp
bool UnsortedType::IsFull() const
{
    NodeType* location;
    try
    {
        location = new NodeType;
        delete location;
        return false;
    }
    catch (std::bad_alloc exception)
    {
        return true;
    }
}
```
Linked List Implementation of MakeEmpty

```cpp
void UnsortedType::MakeEmpty()
{
    NodeType* tempPtr;
    while (listData != NULL)
    {
        tempPtr = listData;
        listData = listData->next;
        delete tempPtr;
    }
    length = 0;
}
```

Why can’t we just set `listData` to `NULL`?

C++ concepts to come

- Exceptions
- Deep vs. shallow copying
- Constructor, destructor, copy constructor
- Dynamically allocated arrays

Order of Magnitude of a Function

- The order of magnitude, or Big-O, of a function expresses an upper bound to the growth of a function relative to its parameters.
- Used to analyze the space and time complexity of algorithms/programs.
Asymptotic Analysis

- Ignoring constants in $T(n)$
- Analyzing $T(n)$ as $n$ "gets large"

Example: $T(n) = 13n^3 + 42n^2 + 2n \log n + 4n$

As $n$ grows larger, $n^3$ is MUCH larger than $n^2$, $n \log n$, and $n$, so it dominates $T(n)$.

The running time grows "roughly on the order of $n^3$"

$T(n) = O(n^3)$

Big-Oh Defined

$T(n) = O(f(n))$ if there are constants $c$ and $n_0$ such that $T(n) \leq c \cdot f(n)$ when $n > n_0$

Big-O Notation

- $T(n) = O(f(n))$ if there are constants $c$ and $n_0$ such that $T(n) \leq c \cdot f(n)$ when $n > n_0$
- If $f(n) = 1000n$ and $g(n) = n^2$, $n_0 = 999$ and $c = 1$, then $f(n) \leq 1 \cdot g(n)$ where $n > n_0$ and we say that $f(n) = O(g(n))$
- The $O$ notation indicates bounded above by a constant multiple of: 
Big-Oh Properties

- Fastest growing function dominates a sum
  \( O(f(n)+g(n)) = O(\max\{f(n), g(n)\}) \)
- Product of upper bounds is upper bound for the product
  \( f(n) \cdot g(n) = O(f(n) \cdot g(n)) \)
- If \( f \) is \( O(g) \) and \( h \) is \( O(r) \) then \( f \cdot h \) is \( O(gr) \)
- \( f(n) \) is transitive
  \( O(1), O(\log n), O(n^{1/2}), O(n \log n), O(n^2), O(2^n), O(n!) \)

Some Big-Oh’s are not reasonable

- Polynomial Time algorithms
  \( O(n^c), c > 1 \)
  Polynomial algorithms are said to be reasonable
  - They solve problems in reasonable times!
  - Coefficients, constants or low-order terms are ignored e.g. if \( f(n) = 2n^2 \) then \( f(n) = O(n^2) \).

- Exponential Time algorithms
  \( O(r^n), r > 1 \)
  Exponential algorithms are said to be unreasonable

Can we justify Big O notation?

- Big O notation is a huge simplification; can we justify it?
- It only makes sense for large problem sizes
- For sufficiently large problem sizes, the highest-order term swamps all the rest!
Classifying Algorithms based on Big-Oh

- A function \( f(n) \) is said to be of at most logarithmic growth if \( f(n) = O(\log n) \).
- A function \( f(n) \) is said to be of at most quadratic growth if \( f(n) = O(n^2) \).
- A function \( f(n) \) is said to be of at most polynomial growth if \( f(n) = O(n^k) \) for some natural number \( k > 1 \).
- A function \( f(n) \) is said to be of at most exponential growth if there is a constant \( c \), such that \( f(n) = O(c^n) \), and \( c > 1 \).
- A function \( f(n) \) is said to have constant running time if the size of the input \( n \) has no effect on the running time of the algorithm, e.g., assignment of a value to a variable. The equation for this algorithm is \( f(n) = c \).
- Other logarithmic classifications: \( f(n) = O(n \log n) \), \( f(n) = O(\log \log n) \).

Names of Orders of Magnitude

- \( O(1) \) bounded (by a constant) time
- \( O(\log N) \) logarithmic time
- \( O(N) \) linear time
- \( O(N \log N) \) \( N \log N \) time
- \( O(N^2) \) quadratic time
- \( O(2^n) \) exponential time

How do various functions grow?

<table>
<thead>
<tr>
<th>( N )</th>
<th>( \log N )</th>
<th>( N \log N )</th>
<th>( N^2 )</th>
<th>( 2^N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>24</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>64</td>
<td>256</td>
<td>65,536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>160</td>
<td>1024</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>384</td>
<td>4,096</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>896</td>
<td>16,384</td>
<td></td>
</tr>
</tbody>
</table>
## Big-O Comparison of List Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array-based</th>
<th>Pointer-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>IsFull</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>GetLength</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>RetrieveItem</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>MakeEmpty</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>InsertItem</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DeleteItem</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>ResetList</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>GetNextItem</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>