CMPT 225

Lecture 5 – linked list
Last Lecture

- Designed List as an ADT
  1. Design the visible section of our List ADT class -&gt; its public interface – public section
  2. Design the invisible section of our List ADT class -&gt; what is hidden behind the wall – private section
     - Data structure -&gt; array
     - Size of array
     - Number of elements

- Implemented List as an ADT
  - Array-based implementation
  - Stack-allocated array and heap-allocated array
  - Introduced test cases and a test driver -&gt; more in Lab 2
Summary: Stack versus heap memory allocation

Memory layout

- **Stack**
  - Code
  - Stack-allocated List object
  - OR
  - Stack-allocated List object

- **Heap**
  - Heap-allocated List object
  - OR
  - Heap-allocated List object

- **Static**
  - Patient elements[INITIAL_SIZE];
  - OR
  - Patient* elements;

- **Public**
  - Destructor
  - ~List();
  - OR
  - insert(...)

```cpp
/* Header Comment Block */
...
int main() {
  /* Variables declaration */
  // Heap-allocated List object
  List* patients = new List();
  OR
  // Stack-allocated List object
  List patients;
  ...
  // Heap-allocated List object
  delete patients;
  ...
  return 0;
}
```

```cpp
/* Header Comment Block */
...
class List {
  private:
    static const int INITIAL_SIZE = 5;
    Patient elements[INITIAL_SIZE];
    OR
    Patient* elements;
    int elementCount;
    int capacity;
  public:
    // Destructor
    ~List();
    OR
    insert(...)
    }
};
```

```cpp
List_Test_Driver.cpp
/* Header Comment Block */
...
int main() {
  /* Variables declaration */
  // Heap-allocated List object
  List* patients = new List();
  OR
  // Stack-allocated List object
  List patients;
  ...
  // Heap-allocated List object
  delete patients;
  ...
  return 0;
}
```

```cpp
List.h
/* Header Comment Block */
...
class List {
  private:
    static const int INITIAL_SIZE = 5;
    Patient elements[INITIAL_SIZE];
    OR
    Patient* elements;
    int elementCount;
    int capacity;
  public:
    // Destructor
    ~List();
    OR
    insert(...)
    }
};
```

```cpp
List_Test_Driver.cpp
/* Header Comment Block */
...
int main() {
  /* Variables declaration */
  // Heap-allocated List object
  List* patients = new List();
  OR
  // Stack-allocated List object
  List patients;
  ...
  // Heap-allocated List object
  delete patients;
  ...
  return 0;
}
```

```cpp
List.h
/* Header Comment Block */
...
class List {
  private:
    static const int INITIAL_SIZE = 5;
    Patient elements[INITIAL_SIZE];
    OR
    Patient* elements;
    int elementCount;
    int capacity;
  public:
    // Destructor
    ~List();
    OR
    insert(...)
    }
};
```

```cpp
List_Test_Driver.cpp
/* Header Comment Block */
...
int main() {
  /* Variables declaration */
  // Heap-allocated List object
  List* patients = new List();
  OR
  // Stack-allocated List object
  List patients;
  ...
  // Heap-allocated List object
  delete patients;
  ...
  return 0;
}
```

```cpp
List.h
/* Header Comment Block */
...
class List {
  private:
    static const int INITIAL_SIZE = 5;
    Patient elements[INITIAL_SIZE];
    OR
    Patient* elements;
    int elementCount;
    int capacity;
  public:
    // Destructor
    ~List();
    OR
    insert(...)
    }
};
```
Activity - Test cases for our List ADT class?

- Inserting into an empty List
- Inserting into a full List
- Inserting into a List that has some elements
Learning Outcomes

At the end of this lecture, a student will be able to:

- define one of the concrete data types, namely *linked list*, and demonstrate, simulate, and trace its operations
- write C++ code
- manipulate pointers
- manage memory in C++
- design test drivers and use test cases
Today’s menu

- Introducing class documentation
- Introducing 2\textsuperscript{nd} data structure: \textit{linked list}
  - Pointers and linked lists
Class Documentation

- Header comment block
  - File name
  - Class description
  - Class invariant (if any)
  - Author
  - Creation (or modification) date

- For each public and private methods
  - Description
  - Precondition
  - Postcondition
Introducing 2nd data structure: linked list

- Made of **pointers** and **nodes**

- Characteristics:
  - Flexible/unbounded size: it grows or shrinks as needed
  - Sequential access
    - Elements must be accessed in some specific order dictated by the links
What can we do with a linked list?

1. **Insert** element into it
2. **Remove** element from it
   - No shifting required
3. **Traverse (iterate through)** a linked list for various purposes such as displaying each element
4. **Search** for a particular element
5. **Concatenate** linked lists
Insert an element into a linked list

@ front

```
... insert(int newElement) // insert of List ADT class
    Node *newNode = new Node(newElement);
    newNode->next = head; // Head NULL or not
    head = newNode;
    elementCount++;
```
Learning Check

- We can now ...
  - Understand and create class documentation
  - Create linked lists and perform some operations
Next Lecture

- Finish introducing linked list
- A second implementation of our List ADT class:
  - Link-based implementation of List ADT class