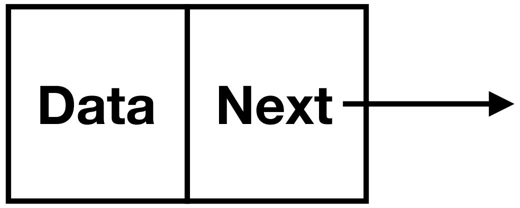
Lab 05: Singly linked list implementation of the List ADT

Disadvantage of array implementation

- There is a dead space in the array.
- Using an array will impose an upper limit on the size of the List.
- Deleting an element pointed by cursor takes O(n) time.
- Inserting an element after cursor takes O(n) time.

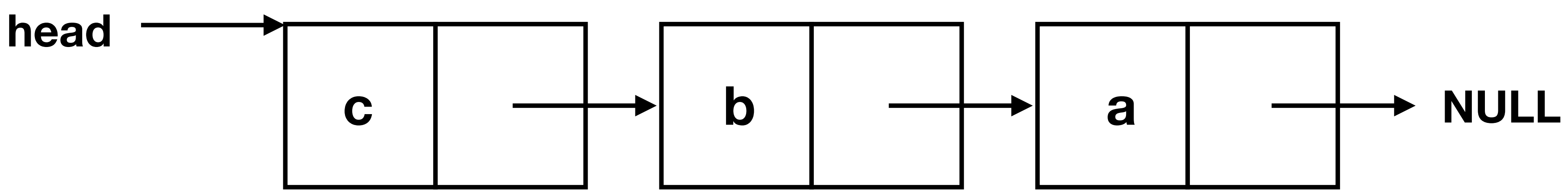
Singly linked list

- A linked list is a linear data structure where each element is a separate object.
- Memory is allocated just when an element is added to the list.
- Destructor must destroy each of the nodes, one-by-one.
- Each element (node) has two items: data, and a reference to the next node.



Singly linked list

- The last node has a reference to null.
- The entry point into a linked list is called the **head** of the list.
- **Cursor** iterates through the list.



Advantage of singly linked array

- Inexpensive deletion/insertion operation.
- Extra memory space for a pointer is required with each element of the list.
- Arrays have better cache locality that can make a pretty big difference in performance.

Data members

```
class ListNode {

      public:

         ListNode(const DataType& nodeData, ListNode* nextPtr);


         DataType dataItem;

         ListNode* next;

    };


    ListNode* head;

    ListNode* cursor;

};
```

Default constructor

```
template <typename DataType>

List<DataType>::ListNode::ListNode(const DataType& nodeData,

                                   ListNode* nextPtr)

// Creates a initialized ListNode by setting the ListNode's data item to the value
nodeData and the ListNode's next pointer to the value of nextPtr

  : dataItem(nodeData), next(nextPtr)

{

}
```

Note: For inner class ListNode, it is required to include both the outer and inner class names.

Creating new ListNode objects: ***head = new ListNode<DataType>(newDataItem, 0);***

Default constructor

```
List(int ignored = 0);

// Creates an empty list. The argument is included for
compatibility with the array implementation (maxSize
specifier) and is ignored.
```

Copy constructor

```
List(const List& other);

//Creates a list which is equivalent in content to the other
list.
```

Assignment operator

```
List& operator=(const List& other);

//Reinitializes the list to be equivalent in content to the
"other" list. Return a reference to this object.

// Note: we include self-assignment protection by checking
whether "this" object is identical to the "other" object.
```

Destructor

```
~List()
{
    clear();
}
// Free the memory used to store the nodes in the list.
```

clear

```
void clear();

// Removes all the items from a list. Sets head and cursor to 0.
```

showStructure

```
void showStructure() const;

// Outputs the items in a list. If the list is empty, outputs

// "Empty list". This operation is intended for testing and

// debugging purposes only.
```

Can be found in show5.cpp

Insert

void insert(const DataType& newDataItem) throw (logic_error);

// Inserts *newDataItem* after the cursor. If the list is empty, then newDataItem is inserted as the first (and only) item in the list.

// In either case, moves the cursor to *newDataItem*.

remove

```
void remove() throw (logic_error);

// Removes the item marked by the cursor from a list.

// Moves the cursor to the next item in the list.

// If the deleted data item was at the end of the list, then
moves the cursor to the beginning of the list.
```

replace

```
void replace(const DataType& newDataItem) throw (logic_error);
```

// Replaces the item marked by the cursor with *newDataItem* and leaves the cursor at *newDataItem*.

isEmpty

```
bool isEmpty() const;

// Returns true if a list is empty. Otherwise, returns false.
```

isFull

```
bool isFull() const
{
    return false;
}
// Returns true if a list is full. Otherwise, returns false.
```

gotoBeginning

```
void gotoBeginning() throw (logic_error);

// If a list is not empty, then moves the cursor to the
beginning of the list. If list is empty, throws logic error.
```

gotoEnd

```
void gotoEnd() throw (logic_error);

// If a list is not empty, then moves the cursor to the end of
the list. Otherwise, throws logic_error.
```

gotoNext

```
bool gotoNext() throw (logic_error);

// If the cursor is not at the end of a list, then moves the
cursor to the next item in the list and returns true.

// Otherwise, leaves cursor unmoved and returns false.
```

gotoPrior

```
bool gotoPrior() throw (logic_error);

// If the cursor is not at the beginning of a list, then moves
the cursor to the preceding item in the list and returns true.

// Otherwise, returns false.
```

getCursor

```
DataType getCursor() const throw (logic_error);

// Returns the item marked by the cursor. Requires that list
is not empty.
```

moveToBeginning

```
void moveToBeginning () throw (logic_error);

// Removes the item marked by the cursor from a list and

// reinserts it at the beginning of the list. Moves the cursor to
the beginning of the list.
```

insertBefore

void insertBefore(const DataType& newDataItem) throw (logic_error);

// Inserts *newDataItem* before the cursor. If the list is empty, then *newDataItem* is inserted as the first (and only) item in the list.

// In either case, moves the cursor to newDataItem.