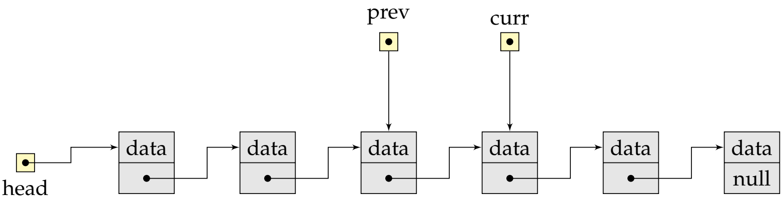
CSCI 2270: Data Structures

Lecture 07: Implementing "Lists" Using Arrays

Ashutosh Trivedi



Department of Computer Science
UNIVERSITY OF COLORADO BOULDER

Abstract Data Type: List

Abstract Data Type: Lists

Abstraction in programming world

– Procedural Abstraction *lets us think about a series of computational steps as an abstract unit (procedure)*.

Abstraction in programming world

- Procedural Abstraction *lets us think about a series of computational steps as an abstract unit (procedure)*.
- Data Abstraction *lets us think about collections of data as abstract entities (abstract data types)*.

What do we mean by a “type”?

- A type is an attribute of data which tells the compiler how the programmer intends to use the data.

What do we mean by a “type”?

- A type is an attribute of data which tells the compiler how the programmer intends to use the data.
- A *type* is characterized by the *values* that data can have and more importantly, the *operations* that one can perform on it:
  - An *integer* is something that you can add, and multiply;

What do we mean by a “type”?

- A type is an attribute of data which tells the compiler how the programmer intends to use the data.
- A *type* is characterized by the *values* that data can have and more importantly, the *operations* that one can perform on it:
  - An *integer* is something that you can add, and multiply;
  - A *string* is something that you can concatenate and find substrings of;

What do we mean by a "type"?

- A type is an attribute of data which tells the compiler how the programmer intends to use the data.
- A *type* is characterized by the *values* that data can have and more importantly, the *operations* that one can perform on it:
  - An *integer* is something that you can add, and multiply;
  - A *string* is something that you can concatenate and find substrings of;
  - A *pointer* is something that stores the address of other variables and supports operations to get the values stored at the address;

What do we mean by a "type"?

- A type is an attribute of data which tells the compiler how the programmer intends to use the data.
- A *type* is characterized by the *values* that data can have and more importantly, the *operations* that one can perform on it:
  - An *integer* is something that you can add, and multiply;
  - A *string* is something that you can concatenate and find substrings of;
  - A *pointer* is something that stores the address of other variables and supports operations to get the values stored at the address;
  - A *boolean* is

What do we mean by a “type”?

- A type is an attribute of data which tells the compiler how the programmer intends to use the data.
- A *type* is characterized by the *values* that data can have and more importantly, the *operations* that one can perform on it:
  - An *integer* is something that you can add, and multiply;
  - A *string* is something that you can concatenate and find substrings of;
  - A *pointer* is something that stores the address of other variables and supports operations to get the values stored at the address;
  - A *boolean* is something that you can negate.

What do we mean by a “type”?

- A type is an attribute of data which tells the compiler how the programmer intends to use the data.
- A *type* is characterized by the *values* that data can have and more importantly, the *operations* that one can perform on it:
  - An *integer* is something that you can add, and multiply;
  - A *string* is something that you can concatenate and find substrings of;
  - A *pointer* is something that stores the address of other variables and supports operations to get the values stored at the address;
  - A *boolean* is something that you can negate.
  - A *list* is

What do we mean by a “type”?

- A type is an attribute of data which tells the compiler how the programmer intends to use the data.
- A *type* is characterized by the *values* that data can have and more importantly, the *operations* that one can perform on it:
  - An *integer* is something that you can add, and multiply;
  - A *string* is something that you can concatenate and find substrings of;
  - A *pointer* is something that stores the address of other variables and supports operations to get the values stored at the address;
  - A *boolean* is something that you can negate.
  - A *list* is linearly-ordered sequence of elements where you can *insert*, *delete*, or *search* elements.

What do we mean by a “type”?

- A type is an attribute of data which tells the compiler how the programmer intends to use the data.
- A *type* is characterized by the *values* that data can have and more importantly, the *operations* that one can perform on it:
  - An *integer* is something that you can add, and multiply;
  - A *string* is something that you can concatenate and find substrings of;
  - A *pointer* is something that stores the address of other variables and supports operations to get the values stored at the address;
  - A *boolean* is something that you can negate.
  - A *list* is linearly-ordered sequence of elements where you can *insert*, *delete*, or *search* elements.
  - A *stack* is

What do we mean by a “type”?

- A type is an attribute of data which tells the compiler how the programmer intends to use the data.
- A *type* is characterized by the *values* that data can have and more importantly, the *operations* that one can perform on it:
  - An *integer* is something that you can add, and multiply;
  - A *string* is something that you can concatenate and find substrings of;
  - A *pointer* is something that stores the address of other variables and supports operations to get the values stored at the address;
  - A *boolean* is something that you can negate.
  - A *list* is linearly-ordered sequence of elements where you can *insert*, *delete*, or *search* elements.
  - A *stack* is something that you can *push* a new element on its top, *pop* an element from the top, *check its emptiness*, and so on.

What do we mean by a "type"?

- A type is an attribute of data which tells the compiler how the programmer intends to use the data.
- A *type* is characterized by the *values* that data can have and more importantly, the *operations* that one can perform on it:
  - An *integer* is something that you can add, and multiply;
  - A *string* is something that you can concatenate and find substrings of;
  - A *pointer* is something that stores the address of other variables and supports operations to get the values stored at the address;
  - A *boolean* is something that you can negate.
  - A *list* is linearly-ordered sequence of elements where you can *insert*, *delete*, or *search* elements.
  - A *stack* is something that you can *push* a new element on its top, *pop* an element from the top, *check its emptiness*, and so on.
- Early programming languages had fixed data types (`int`, `char`) and operations over these types (`+, -, *, /`).

What do we mean by a "type"?

- A type is an attribute of data which tells the compiler how the programmer intends to use the data.
- A *type* is characterized by the *values* that data can have and more importantly, the *operations* that one can perform on it:
  - An *integer* is something that you can add, and multiply;
  - A *string* is something that you can concatenate and find substrings of;
  - A *pointer* is something that stores the address of other variables and supports operations to get the values stored at the address;
  - A *boolean* is something that you can negate.
  - A *list* is linearly-ordered sequence of elements where you can *insert*, *delete*, or *search* elements.
  - A *stack* is something that you can *push* a new element on its top, *pop* an element from the top, *check its emptiness*, and so on.
- Early programming languages had fixed data types (`int`, `char`) and operations over these types (`+, -, *, /`).
- *Abstract data types* (Dahl, Hoare, Liskov among others) marked a major advance in *programming languages*: it allowed a programmer to define their own data types (along with operations over these types).

Data Abstraction

What are some typical operations on data?

Data Abstraction

What are some typical operations on data?

1. Add data to a data collection.

Data Abstraction

What are some typical operations on data?

1. Add data to a data collection.
2. Remove data from data collection.

Data Abstraction

What are some typical operations on data?

1. Add data to a data collection.
2. Remove data from data collection.
3. Ask queries about the data in the collection.

Data Abstraction

What are some typical operations on data?

1. Add data to a data collection.
2. Remove data from data collection.
3. Ask queries about the data in the collection.

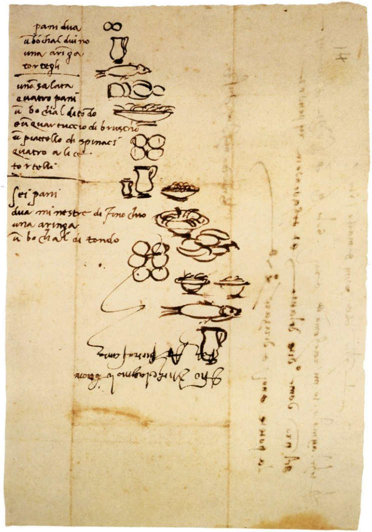
Data abstraction:

1. requires you to think about <u>what</u> (interface) you would like to do with your data, and not focus on <u>how</u> (implementation) you will do it.
2. allows "separation of concerns" by letting you develop different components of your software in isolation

Abstract Data Type: List

Abstract Data Type: Lists

A List by Michelangelo



Abstract Data Types: Lists

1. A list is an ordered sequence of items (with potential duplication), e.g.

$$\langle \text{Denver, Boulder, Erie, Louisville, Denver, Erie} \rangle$$

2. Lists are everywhere:
   - lists of students/addresses/patients/appointments,
   - list of processes/files/interrupts
3. Operations required on a list data type:
   - *construct* an empty list
   - *destruct* a list (return the space to the free store.)
   - *size()*: return size of the list.
   - *insert* an element at a given index on the list.
   - *remove* an element at a given index on the list.
   - *search* for an element in the list.
   - *retrieve* the element at a given index on the list.
   - *capacity()*: return size of allocated storage capacity.
   - *is_empty()*: check if the list is empty.
4. Let's implement such a list in C++.

Lists: Implementation using Fixed-Size Arrays

```
#define MAX_LIST_SIZE 100
/* Class Declaration */
class List {
private:
  std::string m_list[MAX_LIST_SIZE]; //List of items
  int m_size; // number of items in the list
  int m_capacity; // total capacity of the list

public:
  List();   // Default Constructor
  ~List();  // Default Destructor

  int getSize();  // returns number of items in the list
  int getCapacity(); // returns the capacity of the list
  bool isEmpty(); // return true if the list is empty; false otherwise
  bool isFull(); // return true if the list is full; false otherwise

  void insert(std::string data); // insert at item to the end of the list
  void insert(std::string data, int index); // insert an item at a given index
  void remove(int index); // remove the item at a given index
  int search(std::string data); // return the index of the first occurence of the given data, -1 otherwise.
  std::string retrieve(int index); //retrieve an item at a given index
  void print(); // print the list
};
```

- Access level: public, private (default), and protected.
- Public members can be accessed out of the class (the interface).
- Private members can only be accessed from within the class (data-hiding).

Insert operation (Destructive)

Operation. Insert element 99 at index 3:

list $\longrightarrow$

1	1	3	5	6	8	$\perp$	$\perp$

Insert operation (Destructive)

Operation. Insert element 99 at index 3:

list $\longrightarrow$

1	1	3	99	6	8	$\perp$	$\perp$

Insert operation (Non-destructive: Shuffling-Up)

Operation. Insert element 99 at index 3 (while item at the original index must be moved up, and all the items after that index must be shuffled up):

list $\longrightarrow$

1	1	3	5	6	8	$\perp$	$\perp$

Insert operation (Non-destructive: Shuffling-Up)

Operation. Insert element 99 at index 3 (while item at the original index must be moved up, and all the items after that index must be shuffled up):

list →								
	1	1	3	5	6	8	⊥	⊥

```
void List::insert(std::string data, int index) {
  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";
  if (m_size < m_capacity) {
    for (int i = m_size; i > index; i--) {
      m_list[i] = m_list[i-1];
    }
    m_list[index] = data;
    m_size = m_size + 1;
  }
  else {
    throw "List capacity reached";
  }
}
```

Insert operation (Non-destructive: Shuffling-Up)

Operation. Insert element 99 at index 3 (while item at the original index must be moved up, and all the items after that index must be shuffled up):

list $\longrightarrow$ | 1 | 1 | 3 | 5 | 6 | 8 | 8 | $\perp$ |

```
void List::insert(std::string data, int index) {
  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";
  if (m_size < m_capacity) {
    for (int i = m_size; i > index; i--) {
      m_list[i] = m_list[i-1];
    }
    m_list[index] = data;
    m_size = m_size + 1;
  }
  else {
    throw "List capacity reached";
  }
}
```

Insert operation (Non-destructive: Shuffling-Up)

Operation. Insert element 99 at index 3 (while item at the original index must be moved up, and all the items after that index must be shuffled up):

list $\longrightarrow$ | 1 | 1 | 3 | 5 | 6 | 6 | 8 | $\perp$ |

```
void List::insert(std::string data, int index) {
  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";
  if (m_size < m_capacity) {
    for (int i = m_size; i > index; i--) {
      m_list[i] = m_list[i-1];
    }
    m_list[index] = data;
    m_size = m_size + 1;
  }
  else {
    throw "List capacity reached";
  }
}
```

Insert operation (Non-destructive: Shuffling-Up)

Operation. Insert element 99 at index 3 (while item at the original index must be moved up, and all the items after that index must be shuffled up):

list $\longrightarrow$ | 1 | 1 | 3 | 5 | 5 | 6 | 8 | $\perp$ |

```
void List::insert(std::string data, int index) {
  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";
  if (m_size < m_capacity) {
    for (int i = m_size; i > index; i--) {
      m_list[i] = m_list[i-1];
    }
    m_list[index] = data;
    m_size = m_size + 1;
  }
  else {
    throw "List capacity reached";
  }
}
```

Insert operation (Non-destructive: Shuffling-Up)

Operation. Insert element 99 at index 3 (while item at the original index must be moved up, and all the items after that index must be shuffled up):

list $\longrightarrow$ | 1 | 1 | 3 | 99 | 5 | 6 | 8 | $\perp$ |

```
void List::insert(std::string data, int index) {
  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";
  if (m_size < m_capacity) {
    for (int i = m_size; i > index; i--) {
      m_list[i] = m_list[i-1];
    }
    m_list[index] = data;
    m_size = m_size + 1;
  }
  else {
    throw "List capacity reached";
  }
}
```

Remove operation (leave “gaps” in its place)

Operation. Remove the element at index 3.

list $\longrightarrow$

1	1	3	5	6	8	$\perp$	$\perp$

Remove operation (leave "gaps" in its place)

Operation. Remove the element at index 3.

list $\longrightarrow$

1	1	3	$\top$	6	8	$\perp$	$\perp$

Remove operation (Keep array gap-free)

Operation. Remove the element at the index 3 (all the items after the removed item's index must be shuffled down):

$$\text{list} \longrightarrow \boxed{1}\boxed{1}\boxed{3}\boxed{5}\boxed{6}\boxed{8}\boxed{\perp}\boxed{\perp}$$

Remove operation (Keep array gap-free)

Operation. Remove the element at the index 3 (all the items after the removed item's index must be shuffled down):

list $\longrightarrow$ | 1 | 1 | 3 | 5 | 6 | 8 | $\perp$ | $\perp$ |

```
void List::remove(int index) {
  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";
  for (int i = index; i < m_size - 1; i++) {
    m_list[i] = m_list[i+1];
  }
  m_list[m_size-1] = "";
  m_size = m_size - 1;
}
```

Remove operation (Keep array gap-free)

Operation. Remove the element at the index 3 (all the items after the removed item's index must be shuffled down):

list $\longrightarrow$	1	1	3	6	6	8	$\perp$	$\perp$

```
void List::remove(int index) {
  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";
  for (int i = index; i < m_size - 1; i++) {
    m_list[i] = m_list[i+1];
  }
  m_list[m_size-1] = "";
  m_size = m_size - 1;
}
```

Remove operation (Keep array gap-free)

Operation. Remove the element at the index 3 (all the items after the removed item's index must be shuffled down):

list $\longrightarrow$

1	1	3	6	8	8	$\perp$	$\perp$

```
void List::remove(int index) {
  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";
  for (int i = index; i < m_size - 1; i++) {
    m_list[i] = m_list[i+1];
  }
  m_list[m_size-1] = "";
  m_size = m_size - 1;
}
```

Remove operation (Keep array gap-free)

Operation. Remove the element at the index 3 (all the items after the removed item's index must be shuffled down):

list $\longrightarrow$ | 1 | 1 | 3 | 5 | 8 | $\perp$ | $\perp$ | $\perp$ |

```
void List::remove(int index) {
  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";
  for (int i = index; i < m_size - 1; i++) {
    m_list[i] = m_list[i+1];
  }
  m_list[m_size-1] = "";
  m_size = m_size - 1;
}
```

Fixed-Array List Implementation

```
void List::insert(std::string data, int index) {
  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";
  if (m_size < m_capacity) {
    for (int i = m_size; i > index; i--) {
      m_list[i] = m_list[i-1];
    }
    m_list[index] = data;
    m_size = m_size + 1;
  }
  else {
    throw "List capacity reached";
  }
}
```

Pros:

- Fast access of the elements of the list
- Extremely memory efficient as very little memory is required other than that needed to store the contents

Cons:

- Slow deletion and insertion of elements
- Size must be known when the array is created and is fixed

Solution: Dynamic Arrays

```
class List {
private:
  std::string* m_list;
  int m_size;
  int m_capacity;
  int m_resize;
public:
  List();   // Default Constructor
  ~List();  // Default Destructor
  int getSize();  // returns number of items in the list
  bool isEmpty(); // return true if the list is empty; false otherwise
  void insert(std::string data); // insert at item to the end of the list
  void insert(std::string data, int index); // insert an item at a given index
  void remove(int index); // remove the item at a given index
  int search(std::string data); // return the index of the first occurence of the given data, -1 otherwise.
  std::string retrieve(int index); //retrieve an item at a given index
  void print(); // print the list
private:
  void resize(); // Dynamically resize by doubling
};
```

Dynamic Arrays: Insert

```
void List::insert(std::string data, int index) {

  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";

  if (m_size == m_capacity) resize();

  for (int i = m_size; i > index; i--) {
    m_list[i] = m_list[i-1];
  }
  m_list[index] = data;
  m_size = m_size + 1;
}
```

Dynamic Arrays: Insert

```
void List::insert(std::string data, int index) {

  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";

  if (m_size == m_capacity) resize();

  for (int i = m_size; i > index; i--) {
    m_list[i] = m_list[i-1];
  }
  m_list[index] = data;
  m_size = m_size + 1;
}
```

Operation. Insert element 99 at index 3:

list →	1	1	3	5	6	8	11	5

Dynamic Arrays: Insert

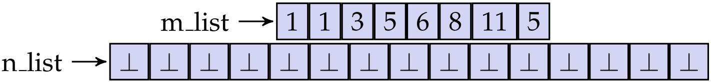
```
void List::resize() {
  m_resize = m_resize + 1;
  std::cout << "Resizing " << m_resize << " times and no of elements: " << m_size << std::endl;


  if (m_capacity == 0) {
    m_list = new std::string [1];
    m_capacity = 1;
  }
  else {
    std::string *n_list = new std::string [m_capacity*2];

    for (int i=0; i< m_capacity; i++) {
      n_list[i] = m_list[i];
    }
    delete[] m_list;

    m_list = n_list;
    m_capacity = m_capacity * 2;
  }
}
```

Operation. Insert element 99 at index 3:



Dynamic Arrays: Insert

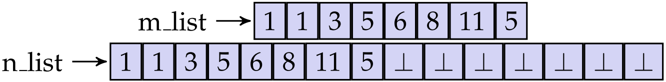
```
void List::resize() {
  m_resize = m_resize + 1;
  std::cout << "Resizing " << m_resize << " times and no of elements: " << m_size << std::endl;


  if (m_capacity == 0) {
    m_list = new std::string [1];
    m_capacity = 1;
  }
  else {
    std::string *n_list = new std::string [m_capacity*2];

    for (int i=0; i< m_capacity; i++) {
      n_list[i] = m_list[i];
    }
    delete[] m_list;

    m_list = n_list;
    m_capacity = m_capacity * 2;
  }
}
```

Operation. Insert element 99 at index 3:



Dynamic Arrays: Insert

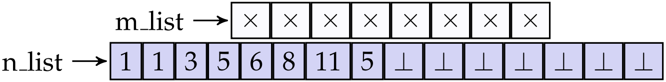
```
void List::resize() {
  m_resize = m_resize + 1;
  std::cout << "Resizing " << m_resize << " times and no of elements: " << m_size << std::endl;


  if (m_capacity == 0) {
    m_list = new std::string [1];
    m_capacity = 1;
  }
  else {
    std::string *n_list = new std::string [m_capacity*2];

    for (int i=0; i< m_capacity; i++) {
      n_list[i] = m_list[i];
    }
    delete[] m_list;

    m_list = n_list;
    m_capacity = m_capacity * 2;
  }
}
```

Operation. Insert element 99 at index 3:



Dynamic Arrays: Insert

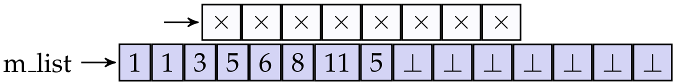
```
void List::resize() {
  m_resize = m_resize + 1;
  std::cout << "Resizing " << m_resize << " times and no of elements: " << m_size << std::endl;


  if (m_capacity == 0) {
    m_list = new std::string [1];
    m_capacity = 1;
  }
  else {
    std::string *n_list = new std::string [m_capacity*2];

    for (int i=0; i< m_capacity; i++) {
      n_list[i] = m_list[i];
    }
    delete[] m_list;

    m_list = n_list;
    m_capacity = m_capacity * 2;
  }
}
```

Operation. Insert element 99 at index 3:



Dynamic Arrays: Remove

```
void List::remove(int index) {
  if ((index < 0) ||  (index > m_size))  throw "Index out of bounds";
  for (int i = index; i < m_size - 1; i++) {
    m_list[i] = m_list[i+1];
  }
  m_list[m_size-1] = "";
  m_size = m_size - 1;
}
```

Dynamic Arrays: Summary

- Before every insertion, check to see if the array needs to resize (grow)
- Growing by doubling works well in practice: to insert n items you need to $\log_2(n)$ resize operations!
- When resizing happens, it is time-consuming!
- Deleting and inserting is the middle still remain efficient (Amortized Cost)!

Abstract Data Types

Abstract data types are an instance of a general principle in software engineering, that combines the following ideas:

1. *Abstraction*. Hiding low-level details with a simpler higher-level idea.
2. *Modularity*. Dividing a system into modules where each module can be separately designed, implemented, and tested.
3. *Encapsulation*. Building walls around the functionality of a module such that bugs in other parts can not damage its integrity, and correctness of the module is its own responsibility.
4. *Information hiding*. Hiding implementation details of a module from the rest of the system, so that those details can be changed without requiring to change the rest of the system.
5. *Separation of concerns*. Making each module responsible for a specific feature (or "concern") rather than distributing responsibilities across multiple modules.