Programming Abstractions

CS106B

Cynthia Lee

Topics:

- **Classes**
  - Practice making our own classes
  - LinkedList class
  - ArrayList class

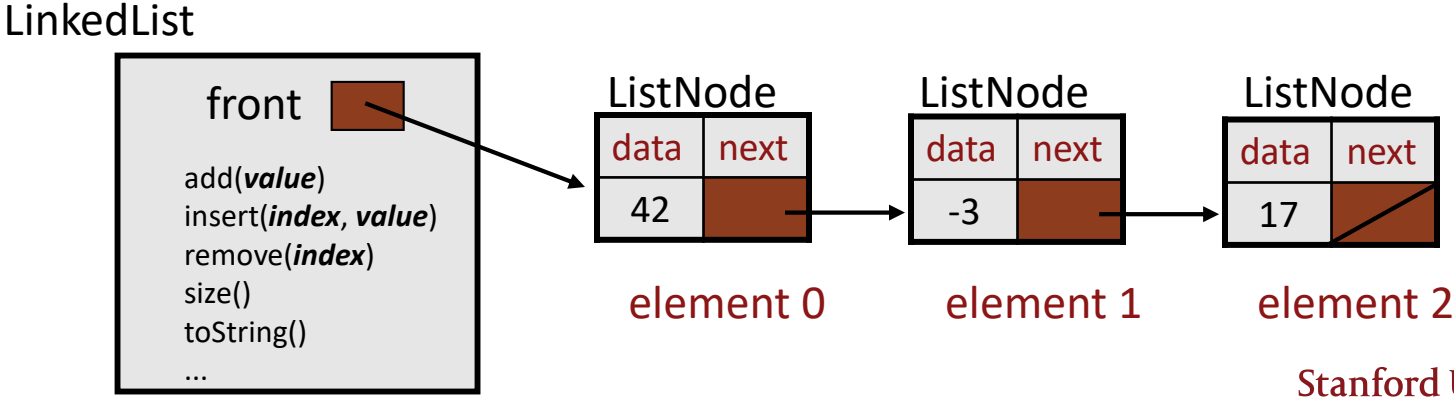
Linked List Data Structure

Putting the ListNode to use

A LinkedList class

Let's write a collection class named `LinkedList`.

- Has the same public members as `ArrayList`, `Vector`, etc.
  - `add`, `clear`, `get`, `insert`, `isEmpty`, `remove`, `size`, `toString`
- The list is internally implemented as a **chain of linked nodes**
  - The `LinkedList` keeps a pointer to its `front` node as a field
  - `NULL` is the end of the list; a `NULL` front signifies an empty list

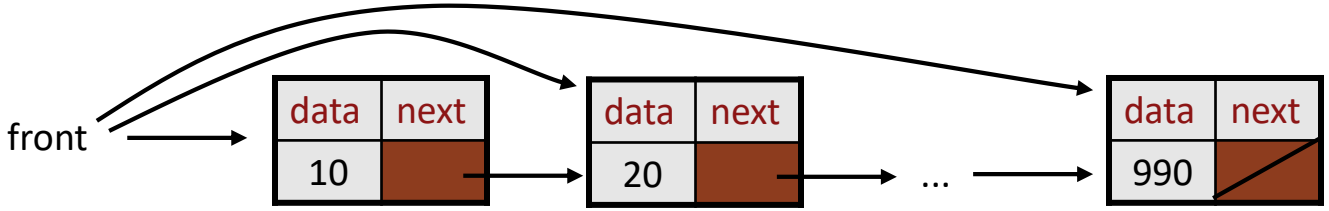


Traversing a list? (BUG version)

What's wrong with this approach to traverse and print the list?

```
while (front != NULL) {

    cout << front->data << endl;
    front = front->next;    // move to next node
}
```

- *It loses the linked list as it is printing it!*

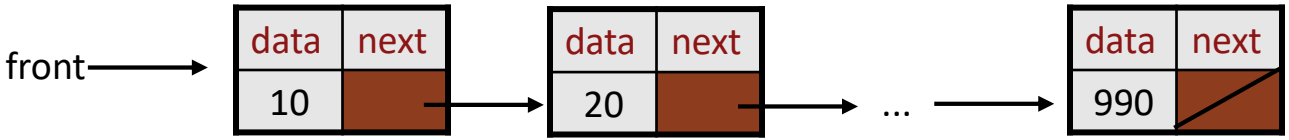


Traversing a list (12.2) (bug fixed version)

The correct way to print every value in the list:

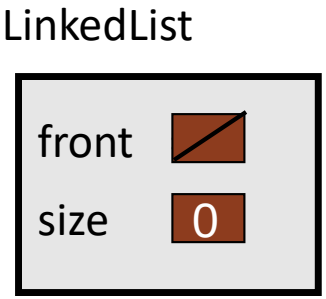
```
ListNode* current = front;
while (current != NULL) {
    cout << current->data << endl;
    current = current->next;  // move to next node
}
```

- Changing `current` does not damage the list.



LinkedList.h

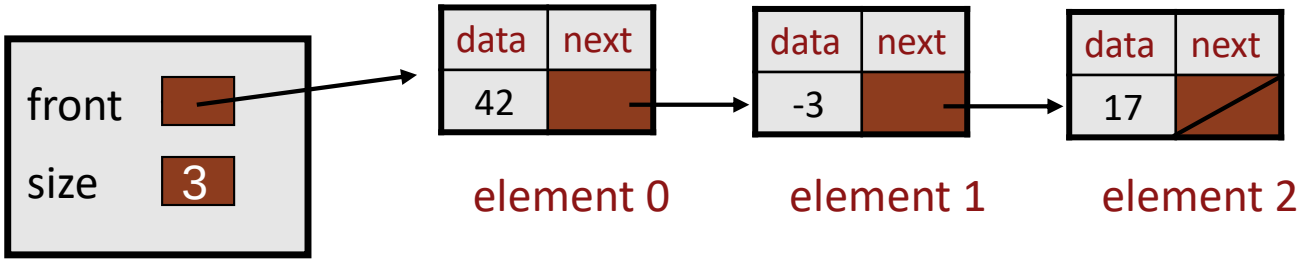
```
class LinkedList {
public:
    LinkedList();
    ~LinkedList();
    void add(int value);
    void clear();
    int get(int index) const;
    void insert(int index, int value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, int value);
    int size() const;

private:
    ListNode* front;
    int size;
};
```



Implementing add

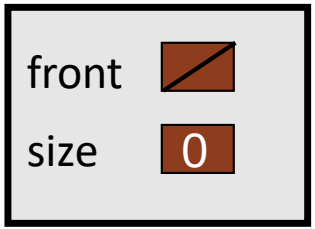
```
// Appends the given value to the end of the list.
void LinkedList::add(int value) {
    ...
}
```

- What pointer(s) must be changed to add a node to the end of a list?
- What different cases must we consider?

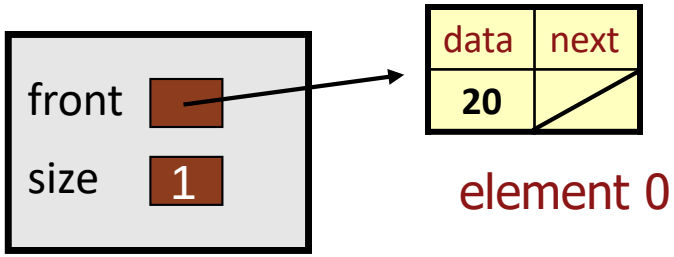


Case 1: Add to empty list

Before adding 20:



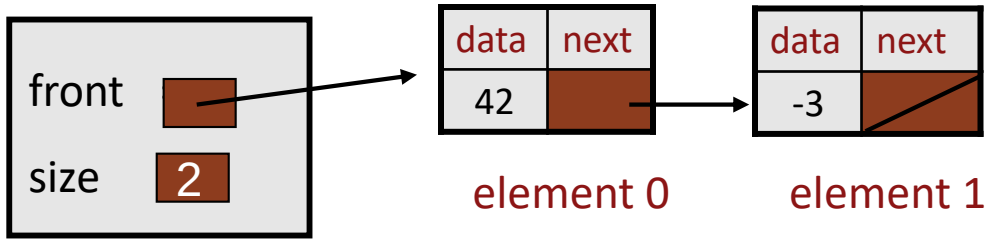
After:



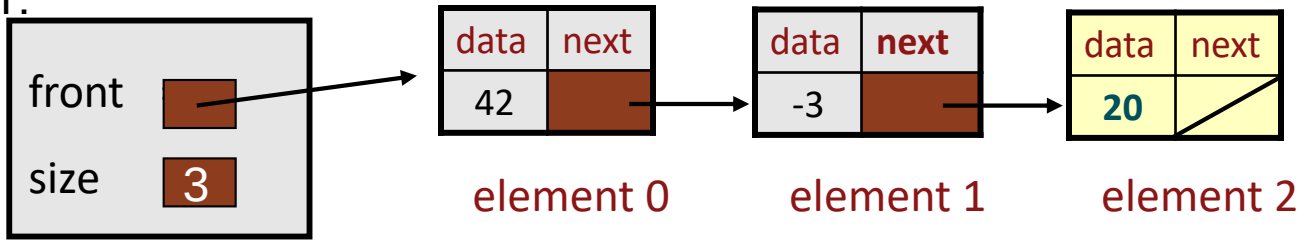
- We must create a new node and attach it to the list.
- For an empty list to become non-empty, we must change `front`.

Case 2: Non-empty list

Before adding value 20 to end of list:

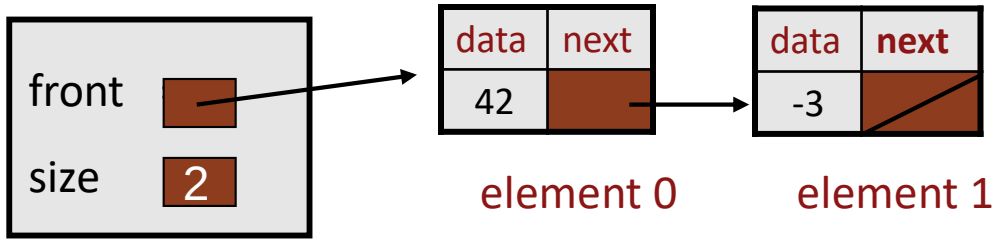


After:



Don't fall off the edge!

Must modify the `next` pointer of the last node.



- Where should `current` be pointing, to add 20 at the end?

Q: What loop test will stop us at this place in the list?

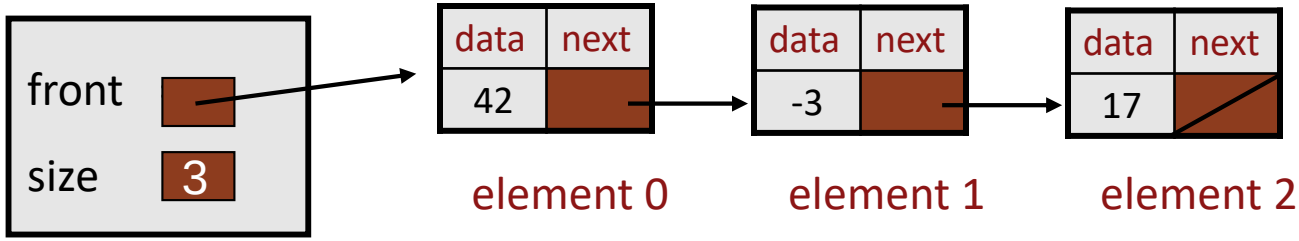
A. `while (current != NULL) { ...`
B. `while (front != NULL) { ...`
C. `while (current->next != NULL) { ...`
D. `while (front->next != NULL) { ...`

Code for add

```
// Adds the given value to the end of the list.
void LinkedList::add(int value) {
    if (front == NULL) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list
        ListNode* current = front;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = new ListNode(value);
    }
    size++;
}
```

Implementing get

```
// Returns value in list at given index.
int LinkedList::get(int index) {
    ...
}
```

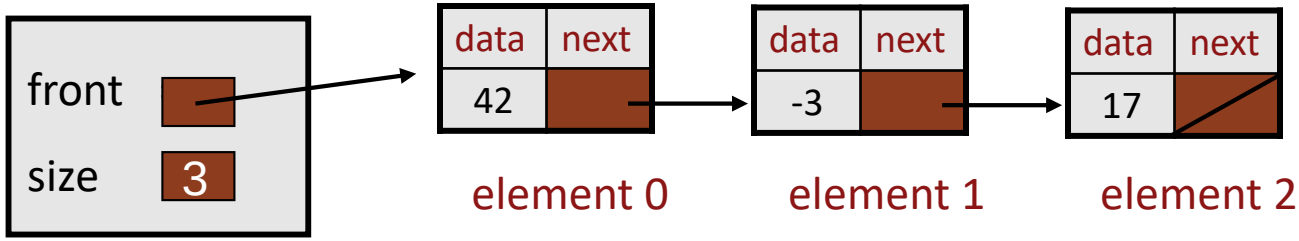


Code for get

```
// Returns value in list at given index.
// Precondition: 0 <= index < size()
int LinkedList::get(int index) {
    ListNode* current = front;
    for (int i = 0; i < index; i++) {
        current = current->next;
    }
    return current->data;
}
```

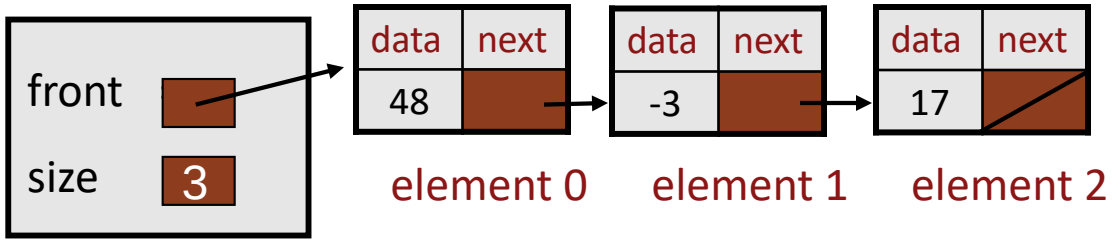
Implementing insert

```
// Inserts the given value at the given index.
void LinkedList::insert(int index, int value) {
    ...
}
```

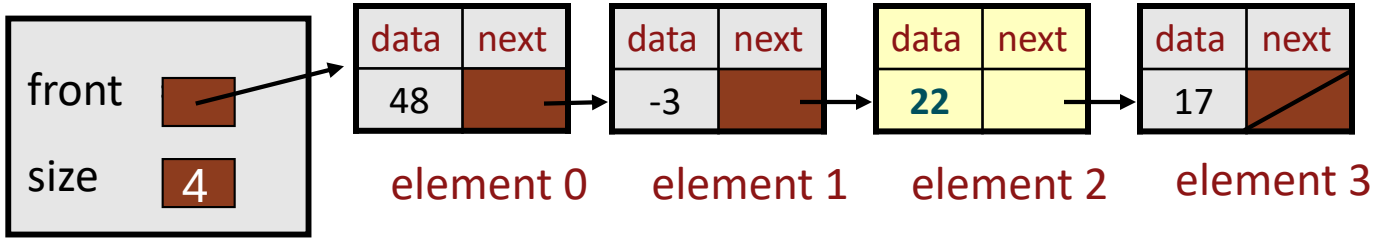


Inserting into a list

Before inserting element at index 2:



After:



Q: How many times to advance `current` to insert at index *i* ?

A. *i* - 1 **B.** *i* **C.** *i* + 1 **D.** none of the above

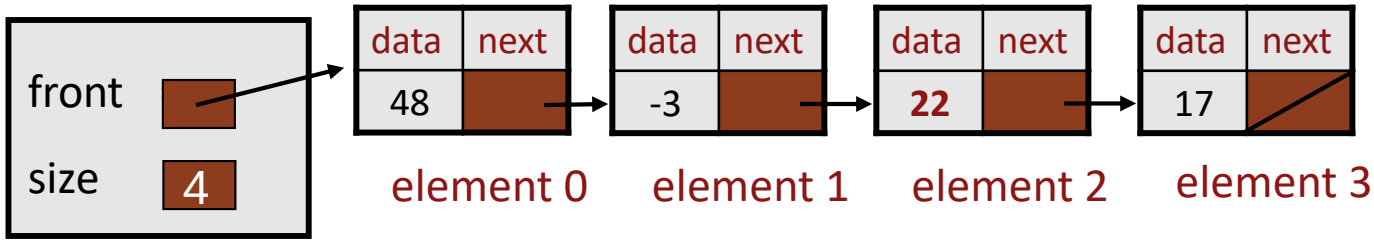
Code for insert

```
// Inserts the given value at the given index.
// Precondition: 0 <= index <= size()
void LinkedList::insert(int index, int value) {
    if (index == 0) {
        // adding to an empty list
        front = new ListNode(value, front);
    } else {
        // inserting into an existing list
        ListNode* current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current->next;
        }
        current->next =
            new ListNode(value, current->next);
    }
    size++;
}
```

Implementing remove

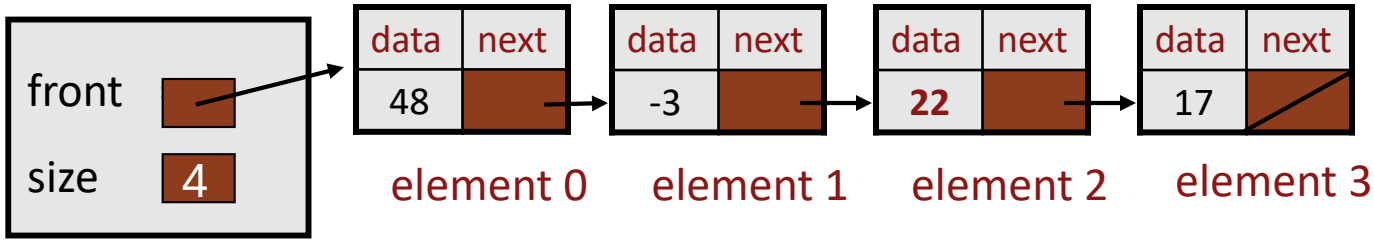
```
// Removes value at given index from list.
void LinkedList::remove(int index) {
    ...
}
```

- What pointer(s) must be changed to remove a node from a list?
- What different cases must we consider?

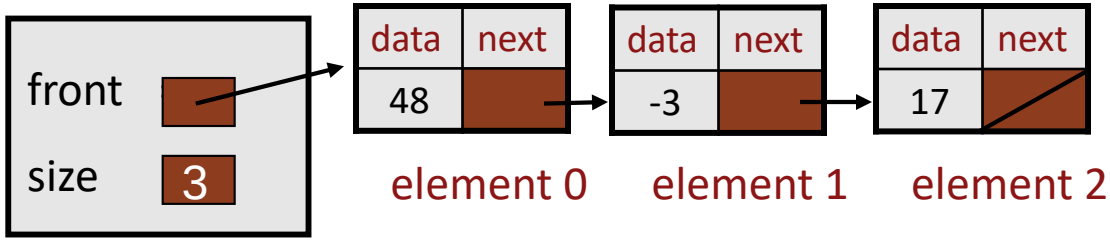


Removing from a list

Before removing element at index 2:



After:



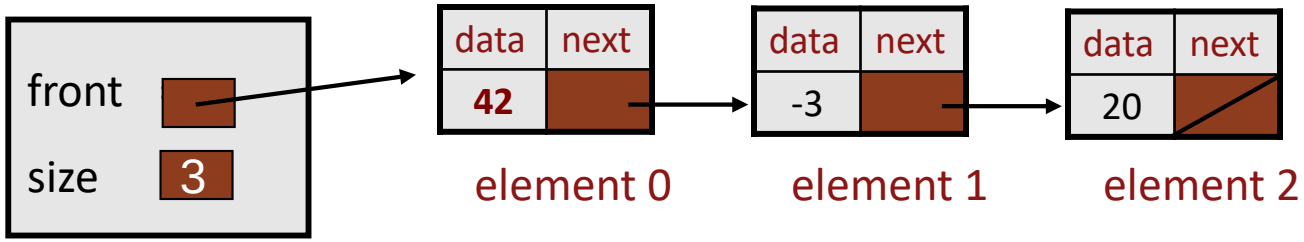
trash

Where should `current` be pointing?

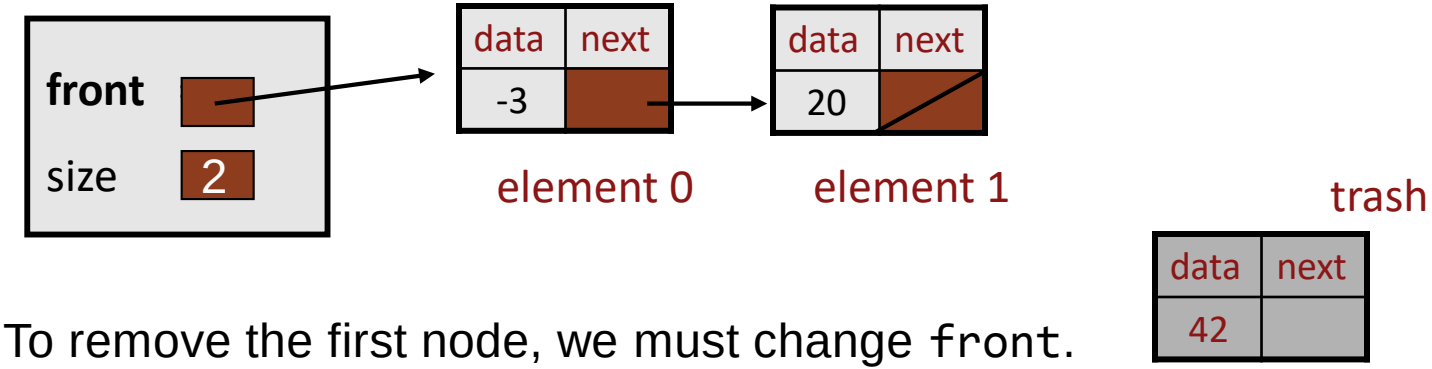
How many times should it advance from `front`?

Removing from front

Before removing element at index 0:



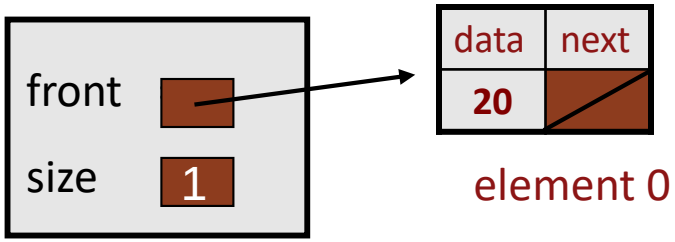
After:



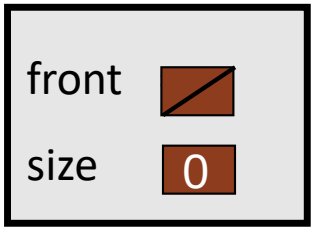
To remove the first node, we must change `front`.

Removing the only element

Before:



After:



- We must change the front field to store `NULL` instead of a node.
- Do we need a special case to handle this?

Code for remove

```
// Removes value at given index from list.
// Precondition: 0 <= index < size()
void LinkedList::remove(int index) {
    ListNode* trash;
    if (index == 0) {   // removing first element
        trash = front;
        front = front->next;
    } else {            // removing elsewhere in the list
        ListNode* current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current->next;
        }
        trash = current->next;
        current->next = current->next->next;
    }
    size--;
    delete trash;
}
```

Other list features

Add the following public members to the `LinkedList`:

- `size()`
- `isEmpty()`
- `set(`***`index`***`, `***`value`***`)`
- `clear()`
- `toString()`

Add preconditions and exception tests as appropriate.

Implementing our ArrayList

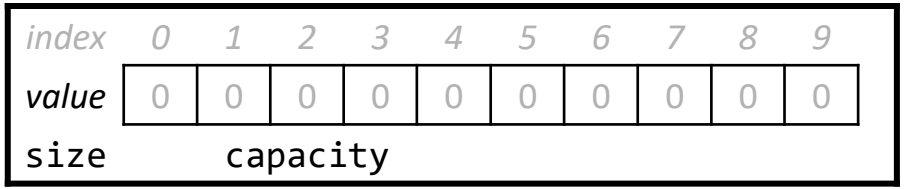
Making our own container class!

How a Vector works (our ArrayList will work same way)

Inside a Vector is an **array** storing the elements you have added.

- Typically the array is larger than the data added so far, so that it has some extra slots in which to put new elements later.
  - When we say `size`, we mean the number of items currently stored, and we say `capacity` to refer to the total space.

```
Vector<int> v;
v.add(42);
v.add(-5);
v.add(17);
```



Exercise

Let's write a class that implements a growable array of integers.

- We'll call it `ArrayList`. It will be very similar to the C++ `Vector`.
- its behavior:
  - `add(`***`value`***`)`
  - `get(`***`index`***`),`
  - `size(),`
  - `remove(`***`index`***`)`
  - `indexOf(`***`value`***`),`
  - `toString()`

  `insert(`***`index, value`***`)`
  `set(`***`index, value`***`)`
  `isEmpty()`
  `contains(`***`value`***`)`

  ...
- We'll start with an array of **length (`capacity`) 10** by default, and grow it as needed.

ArrayList.h

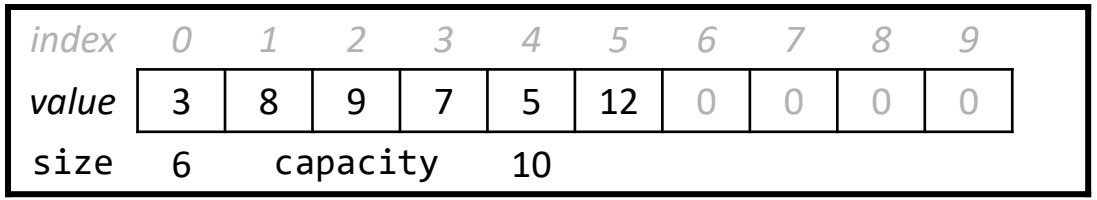
```
#ifndef _arraylist_h
#define _arraylist_h
#include <string>
using namespace std;

class ArrayList {
public:
    ArrayList();
    void add(int value);
    void clear();
    int get(int index) const;
    void insert(int index, int value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, int value);
    int size() const;
    string toString() const;

private:
    int* myElements;   // array of elements
    int myCapacity;    // length of array
    int mySize;        // number of elements added
};

#endif
```

Implementing add

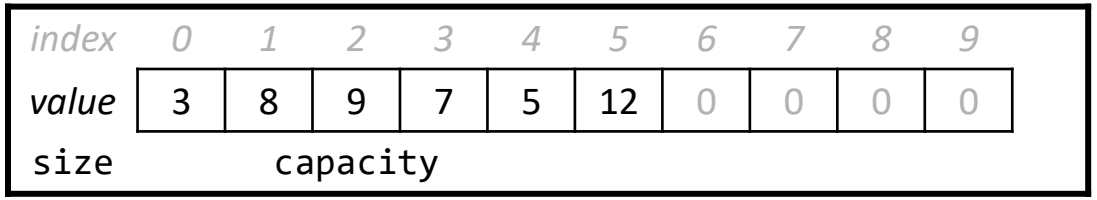
How do you append to the end of a list? `list.add(42);`

- place the new value in slot number `size`
- increment `size`

Before



After



Implementing `insert`

How do you insert in the middle of a list? `list.insert(3, 42);`

- shift elements right to make room for the new element
- increment `size`

Before

index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	7	5	12	0	0	0	0

size 6 capacity 10

After

index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	**42**	7	5	12	0	0	0

size **7** capacity 10

Q: In which direction should our array-shifting loop traverse?

A. left-to-right **B.** right-to-left **C.** either is fine

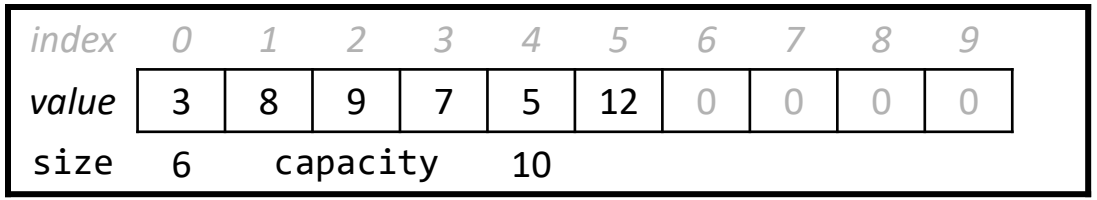
insert solution

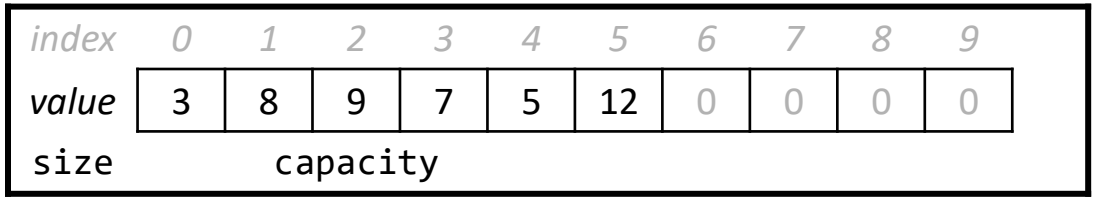
```
// in ArrayList.cpp
void ArrayList::insert(int index, int value) {
    // shift right to make room
    for (int i = mySize; i > index; i--) {
        myElements[i] = myElements[i - 1];
    }
    myElements[index] = value;
    mySize++;
}
```

Implementing `clear`

How do you clear the list? `list.clear();`

- change `size` to 0
- do we need to zero out all the data?





Other members

Let's implement the following member functions in our list:

- `size()` - Returns the number of elements in the list.
- `get(`***`index`***`)` - Returns the value at a given index.
- `set(`***`index, value`***`)` - Changes the value at the given index.
- `isEmpty()` - Returns `true` if list contains no elements.
  - (Why bother to write this if we already have a `size` function?)

- `toString()` - String of the list such as `"{4, 1, 5}"`.
- `operator <<` - Make the list printable to `cout`

Other members code

```
// in ArrayList.cpp
int ArrayList::get(int index) {
    return myElements[index];
}

void ArrayList::set(int index, int value) {
    myElements[index] = value;
}

int ArrayList::size() {
    return mySize;
}

bool ArrayList::isEmpty() {
    return mySize == 0;
}
```

Implementing remove

How do you remove an element from a list? `list.remove(2);`

- shift elements left to cover the deleted element
- decrement *size*

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	**9**	7	5	12	0	0	0	0
size	6	*capacity*	10							

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	7	5	12	0	0	0	0	0
size	**5**	*capacity*	10							

Q: In which direction should our array-shifting loop traverse?

A. left-to-right **B.** right-to-left **C.** either is fine

remove solution

```
// in ArrayList.cpp
void ArrayList::remove(int index) {
    // shift left to cover up the slot
    for (int i = index; i < mySize; i++) {
        myElements[i] = myElements[i + 1];
    }
    mySize--;
}
```

Destructor (12.3)

```
// ClassName.h
~ClassName();
```

```
// ClassName.cpp
ClassName::~ClassName() { ...
```

destructor: Called when the object is deleted by the program.
(when the object goes out of {} scope; opposite of a constructor)

- Useful if your object needs to do anything important as it dies:
  - saving any temporary resources inside the object
  - freeing any dynamically allocated memory used by the object's members
  - ...

- Does our `ArrayList` need a destructor? If so, what should it do?

Destructor solution

```
// in ArrayList.cpp
void ArrayList::~ArrayList() {
    delete[] myElements;
}
```

Extra topic: Resizing

What to do when you run out of space

Running out of space

What if the client wants to add more than 10 elements?

index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	7	5	12	4	8	1	6
size	10	capacity	10							

- `list.add(75);  // add an 11th element`

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
value	3	8	9	7	5	12	4	8	1	6	**75**	0	0	0	0	0	0	0	0	0
size	**11**	capacity				**20**														

- Answer: **Resize the array** to one twice as large.
  - Make sure to *free the memory* used by the old array!

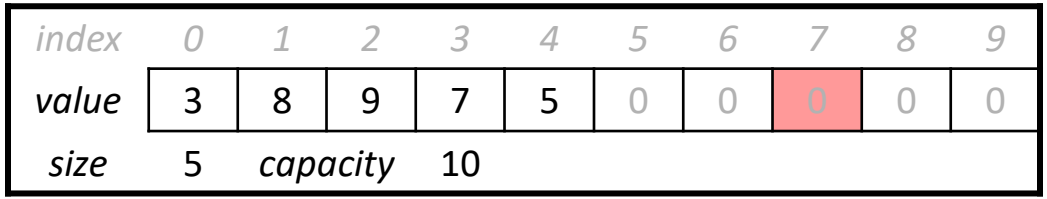
Resize solution

```
// in ArrayList.cpp
void ArrayList::checkResize() {
    if (mySize == myCapacity) {
        // create bigger array and copy data over
        int* bigger = new int[2 * capacity]();
        for (int i = 0; i < myCapacity; i++) {
            bigger[i] = myElements[i];
        }
        delete[] myElements;
        myElements = bigger;
        myCapacity *= 2;
    }
}
```

Problem: size vs. capacity

What if the client accesses an element past the size?

```
list.get(7)
```



- Currently the list allows this and returns 0.
  - Is this good or bad? What (if anything) should we do about it?

Private helpers

```
// in ClassName.h file
private:
    returnType name(parameters);
```

a **private member** function can be called only by its own class

- your object can call the "helper" function, but clients cannot call it

```
void ArrayList::checkIndex(int i, int min, int max) {
    if (i < min || i > max) {
        throw "Index out of range";
    }
}
```

Extra topic: Template classes

Something that Stanford library containers have that our ArrayList lacks.

Template function (14.1-2)

```
template<typename T>
returntype name(parameters) {
    statements;
}
```

Template: A function or class that accepts a *type parameter(s)*.

- Allows you to write a function that can accept many types of data.
- Avoids redundancy when writing the same common operation on different types of data.

- Templates can appear on a single function, or on an entire class.

- FYI: Java has a similar mechanism called *generics*.

Template func example

```
template<typename T>
T max(T a, T b) {
    if (a < b) { return b; }
    else       { return a; }
}
```

- The template is *instantiated* each time you use it with a new type.
  - The compiler actually generates a new version of the code each time.
  - The type you use must have an operator < to work in the above code.

```
int i    = max(17, 4);          // T = int
double d = max(3.1, 4.6);       // T = double
string s = max(string("hi"),    // T = string
               string("bye"));
```

Template class (14.1-2)

Template class: A class that accepts a type parameter(s).

- In the header and cpp files, mark each class/function as templated.
- Replace occurrences of the previous type `int` with `T` in the code.

```
// ClassName.h
template<typename T>
class ClassName {
    ...
};

// ClassName.cpp
template<typename T>
type ClassName::name(parameters) {
    ...
}
```

Recall: ArrayList.h

```
class ArrayList {
public:
    ArrayList();
    ~ArrayList();
    void add(int value);
    void clear();
    int get(int index) const;
    void insert(int index, int value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, int value) const;
    int size() const;
    string toString() const;
private:
    int* elements;
    int mysize;
    int capacity;
    void checkIndex(int index, int min, int max) const;
    void checkResize();
};
```

Template ArrayList.h

```
template <typename T> class ArrayList {
public:
    ArrayList();
    ~ArrayList();
    void add(T value);
    void clear();
    T get(int index) const;
    void insert(int index, T value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, T value) const;
    int size() const;
    string toString() const;
private:
    T* elements;
    int mysize;
    int capacity;
    void checkIndex(int index, int min, int max) const;
    void checkResize();
};
```

Template ArrayList.cpp

```
template <typename T>
ArrayList<T>::ArrayList() {
    myCapacity = 10;
    myElements = new T[myCapacity];
    mySize = 0;
}

template <typename T>
void ArrayList<T>::add(T value) {
    checkResize();
    myElements[mySize] = value;
    mySize++;
}

template <typename T>
T ArrayList<T>::get(int index) const {
    checkIndex(index, 0, mySize - 1);
    return myElements[index];
}

...
```

Template .h and .cpp

Because of an odd quirk with C++ templates, the separation between .h header and .cpp implementation must be reduced.

- Either write all the bodies in the .h file (suggested),
- Or `#include` the .cpp at the end of .h file to join them together.

```
// ClassName.h
#ifndef _classname_h
#define _classname_h

template<typename T>
class ClassName {
    ...
};

#include "ClassName.cpp"
#endif   // _classname_h
```