Abstract Data Type
part two

ADT LIST

Outline

- Destructor
- Overloaded operator
- ADT list

Destructor

- Destructor is activated when an object of the class exits its scope
- a class can only have ONE destructor
- destructor is ONLY necessary in a class when data of the class has acquired dynamically allocated memory.

Destructor

- Destructor has the same name as the class, preceded by ~
- Destructor does not have return type

header file:

```
class  Time
{
 public:
     ….
       ~Time();
 private:
      …
};
```

implementation file t:

```
…
Time::~Time()
{
        cout << “an object exits its scope”
           << endl;   // for illustration only
}
…
```

Copy Constructor

- Copy constructor is activated when an object is created as a copy of another object

header file (time.h):

```
class  Time
{
 public:
      ….
        Time(const Time& t);
 private:
       …
};
```

implementation file (time.cpp):

```
…
Time::Time(const Time & t)
{
        hrs = t.hrs;
        mins = t.mins;
        secs = t.secs;
}
…
```

Copy Constructor

- Copy constructor is activated when an object is created as a copy of another object

client file (main.cpp):

```
#include “time.h”
int main()
{
        Time  firstTime(3, 4, 20);
        Time secondTime(firstTime);

        …
}
```

Overloaded Operator - struct

- Can declare member function in struct type
- All data and function members in struct are default to be public

- Define a new meaning for an existing operator

Specification file:

Overloaded Operator - struct

Specification file
"cardStruct.h"

```
… // preprocessor directive

struct CardStruct
{Overload
   int        suit;
   int        value, point;

   bool  operator< (const CardStruct& rhs) const;
   bool  smaller_than(const CardStruct& rhs,  int lead_suit)
              const;
};
```

Overloaded Operator - struct

Implementation file
"cardStruct.cpp"

```
#include "cardStruct.h"

bool CardStruct::operator<(const CardStruct& rhs) const
{
   return (value < rhs.value);
}
```

Overloaded Operator - struct

client file (main.cc):

```
#include “cardStruct.h”
int main()
{
   CardStruct p1, p2;
   p1.suit  = 1;
   p1.value = 2;
   p2.suit = 1;
   p2.value = 1;

   if (p1 < p2)
      cout << "p1 smaller than p2" << endl;
   else
      cout << "p1 greater than p2" << endl;
}
```

Implementation file

```
bool CardStruct::smaller_than(const CardStruct&rhs,
                                        int lead_suit) const
{
   if ((suit == lead_suit)&&(rhs.suit == lead_suit))
      return (*this < rhs);
   else if ((suit == lead_suit)&&(rhs.suit != lead_suit))
      return(false);
   else if ((suit != lead_suit)&&(rhs.suit!=lead_suit))
      return (*this < rhs);
   else
     return (true);
}
```

Overloaded Operator vs. Member function

client file (main.cc):

```
#include “cardStruct.h”
int main()
{
   CardStruct p1, p2;
   p1.suit  = 1;
   p1.value = 2;
   p2.suit = 1;
   p2.value = 1;

   if (p1.smaller_than(p2))
      cout << "p1 smaller than p2" << endl;
   else
      cout << "p1 greater than p2" << endl;
}
```

Overloaded Operator - **class**

- Define a new meaning for the operator ==
- It compares whether two **Time** objects are equal

header file (time.h):

```
class  Time
{
 public:
      ….
        bool operator==(const Time& t) const;
 private:
       …
};
```

Overloaded Operator - **class**

implementation file (time.cpp):

```
#include “time.h”
…
bool Time::operator==(const Time & t)  const
 {
        return((hrs== t.hrs) && (mins==t.mins) &&
                                (secs = t.secs));
 }
 …
```

Overloaded Operator - class

Client file (main.cpp):

```
#include “time.h”

int main()
{
        Time  firstTime(3, 4, 6);
        Time  secondTime(5, 2, 10);
        …

        if (firstTime == secondTime)
                cout << “Same time” << endl;
        …
```

Overloaded Operator - **class**

- Define a new meaning for the operator <<
- It compares whether two **Time** objects are equal

header file (time.h):

```
class  Time
{
 public:
     ….
   friend ostream& operator << (ostream& os, const Time& t);
   friend istream& operator>>(istream&is, Time&t);
 private:
     …
};
```

Overloaded Operator - class

implementation file (time.cpp):

```
#include “time.h”
…
ostream& operator<<(ostream& os, const Time & t)
{
        os << “Hour:  \t” << t.hrs << endl;
        os<< “Minutes: \t” << t.mins << endl;
        os << “Seconds:\t” << t.secs << endl;

        return os;
}
…
```

Overloaded Operator - class

Client file (main.cpp):

```
#include “time.h”

int main()
{
        Time  firstTime(3, 4, 6);
        …

        cout << firstTime;
```

Overloaded Operator - class

Client file (main.cpp):

```
#include “time.h”
#include <fstream>
int main()
{
        Time  firstTime(3, 4, 6);
        ofstream output(“report”);
        …

        output << firstTime;
```

Overloaded Operator - **class**

- Define a new meaning for the operator <<
- It compares whether two **Time** objects are equal

header file (time.h):

```
class  Time
{
 public:
     ….
   friend ostream& operator << (ostream&os, const Time& t);
   friend istream& operator>>(istream& is, Time&t);
 private:
     …
};
```

Overloaded Operator - class

implementation file (time.cpp):

```
#include “time.h”
…
istream& operator>>(istream& is, Time & t)
{
        is >> t.hrs;
        is>>t.mins;
        is >>t.secs;

        return is;
}
…
```

Overloaded Operator - class

Client file (main.cpp):

```
#include “time.h”

int main()
{
        Time  firstTime;

        …
        cin>>firstTime;
        …
```

Overloaded Operator - class

Client file (main.cpp):

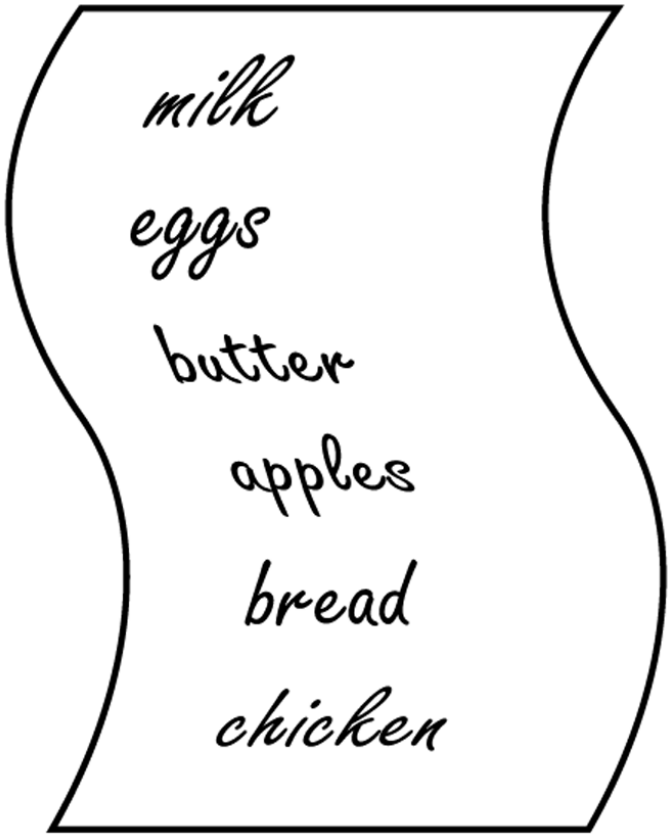
```
#include “time.h”
#include <fstream>
int main()
{
        Time  firstTime;
        ifstream myIn(“data”);

        …
        myIn >>firstTime;
        …
```

Exercise - Define an ADT Person specification, implementation and client

- Data: lastname, firstname, phone, id
- Operations:
  - Default constructor
  - Value constructor
  - Copy constructor
  - Methods to retrieve lastname, firstname, phone, and id
  - Destructor
  - Overloaded < operator (comparison based on lastname and firstname of the person)
  - Overloaded >> operator
  - Overloaded << operator

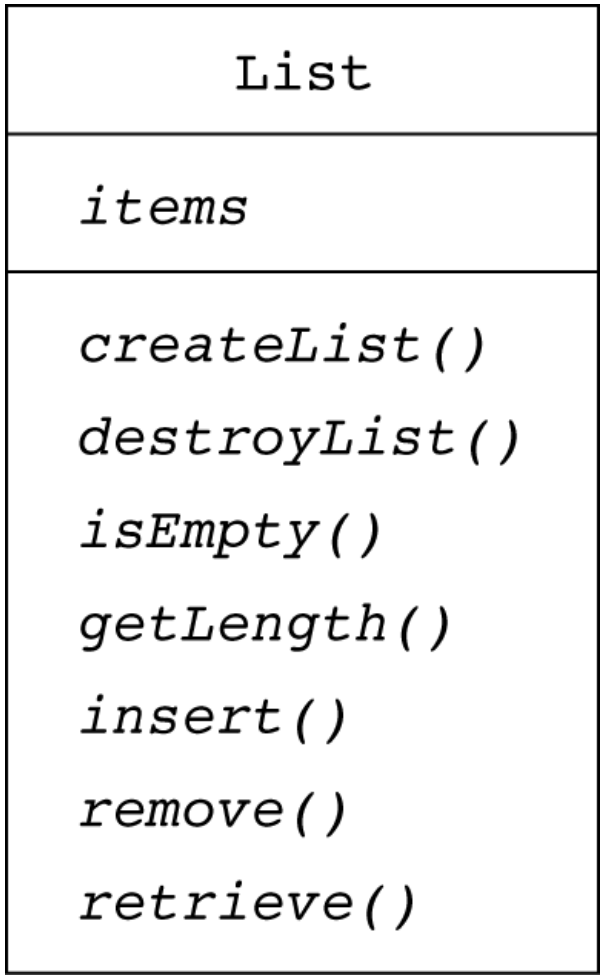
Another Example ADT



Grocery list

An Example ADT: a list

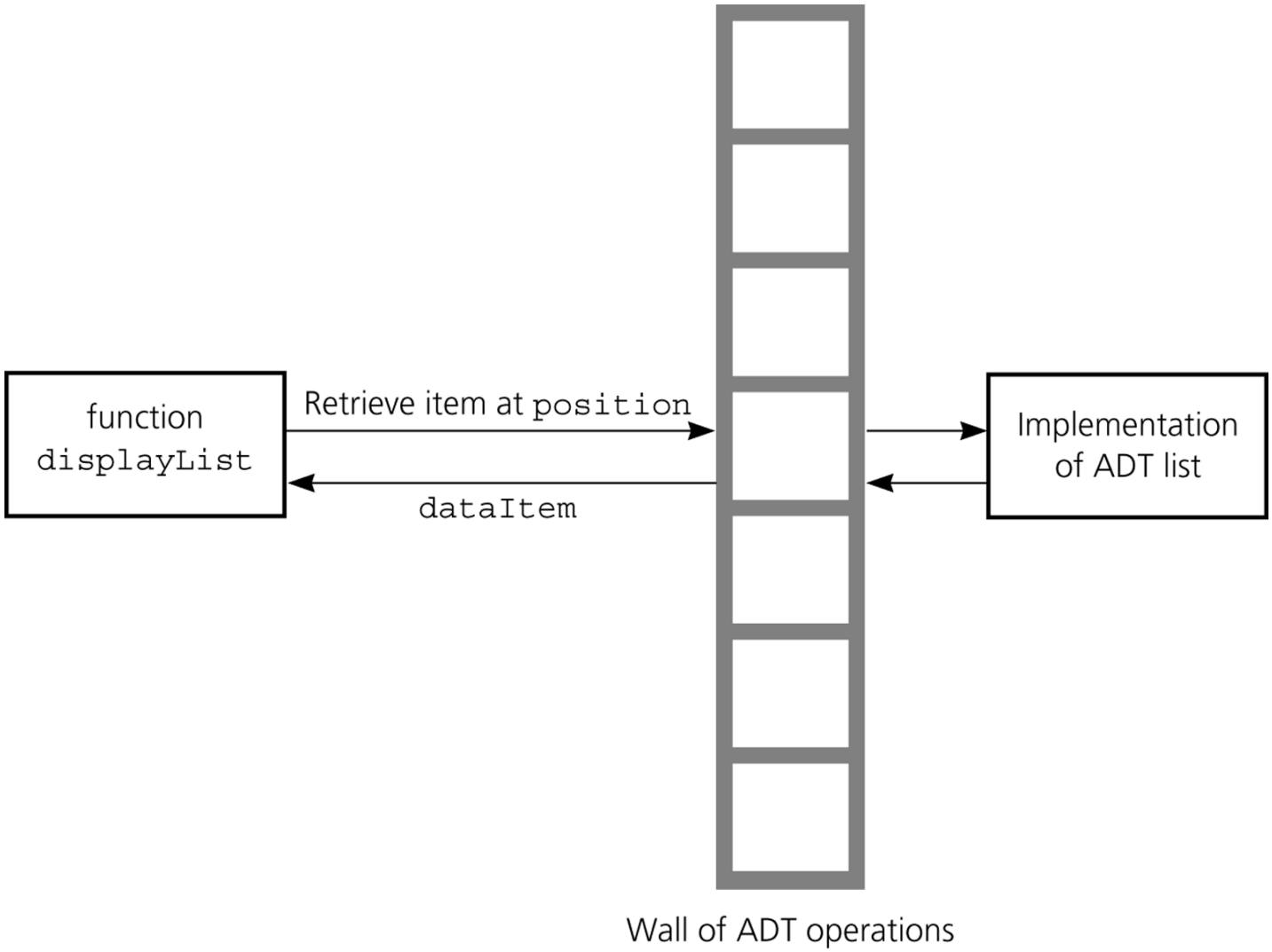
ADT list


List
items
createList()
destroyList()
isEmpty()
getLength()
insert()
remove()
retrieve()

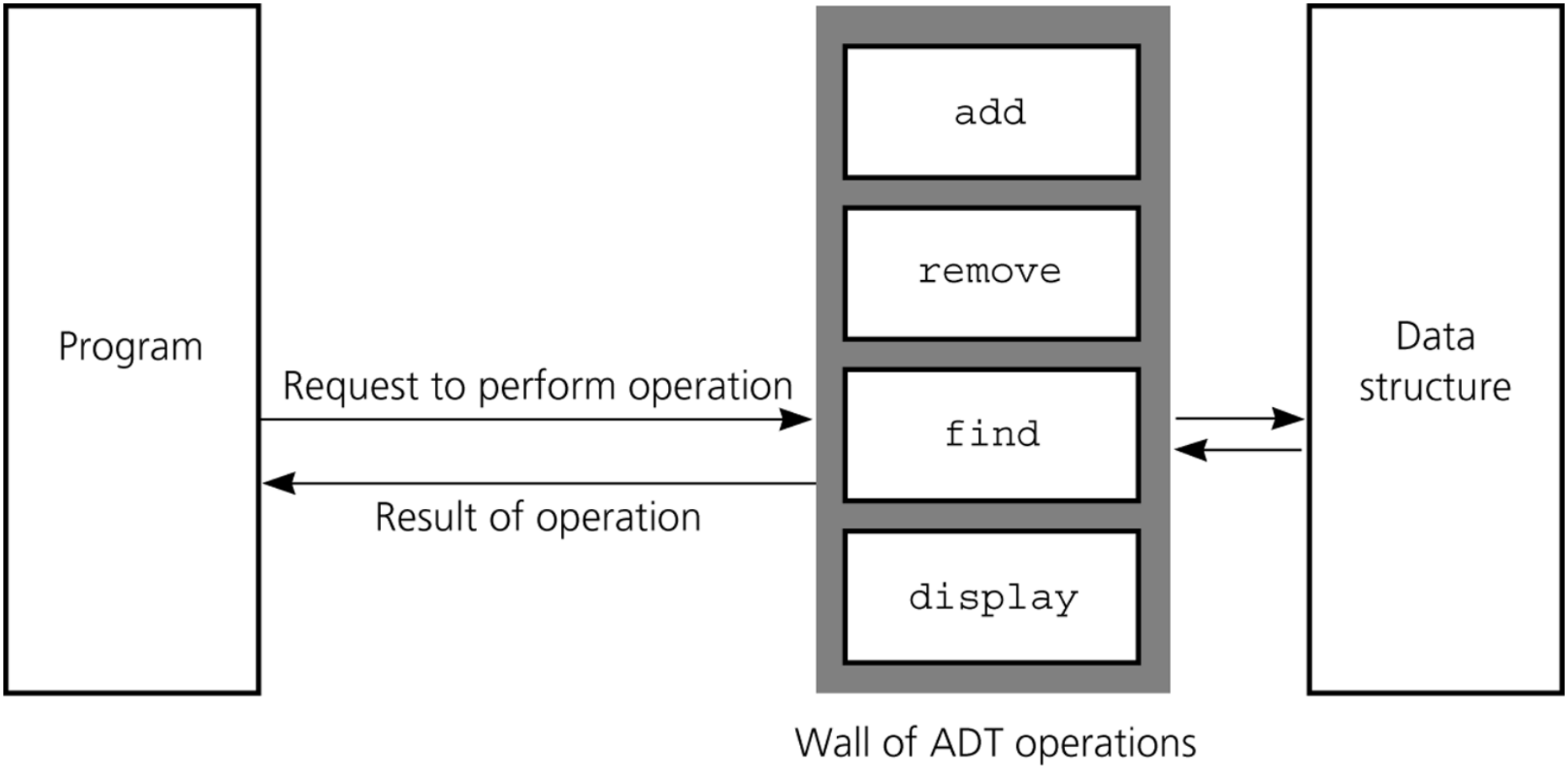
Specification vs. Implementation

- Two steps in ADT construction
  - Specification → what each operation does?
  - Implementation → How each operation does it? How is the data stored?

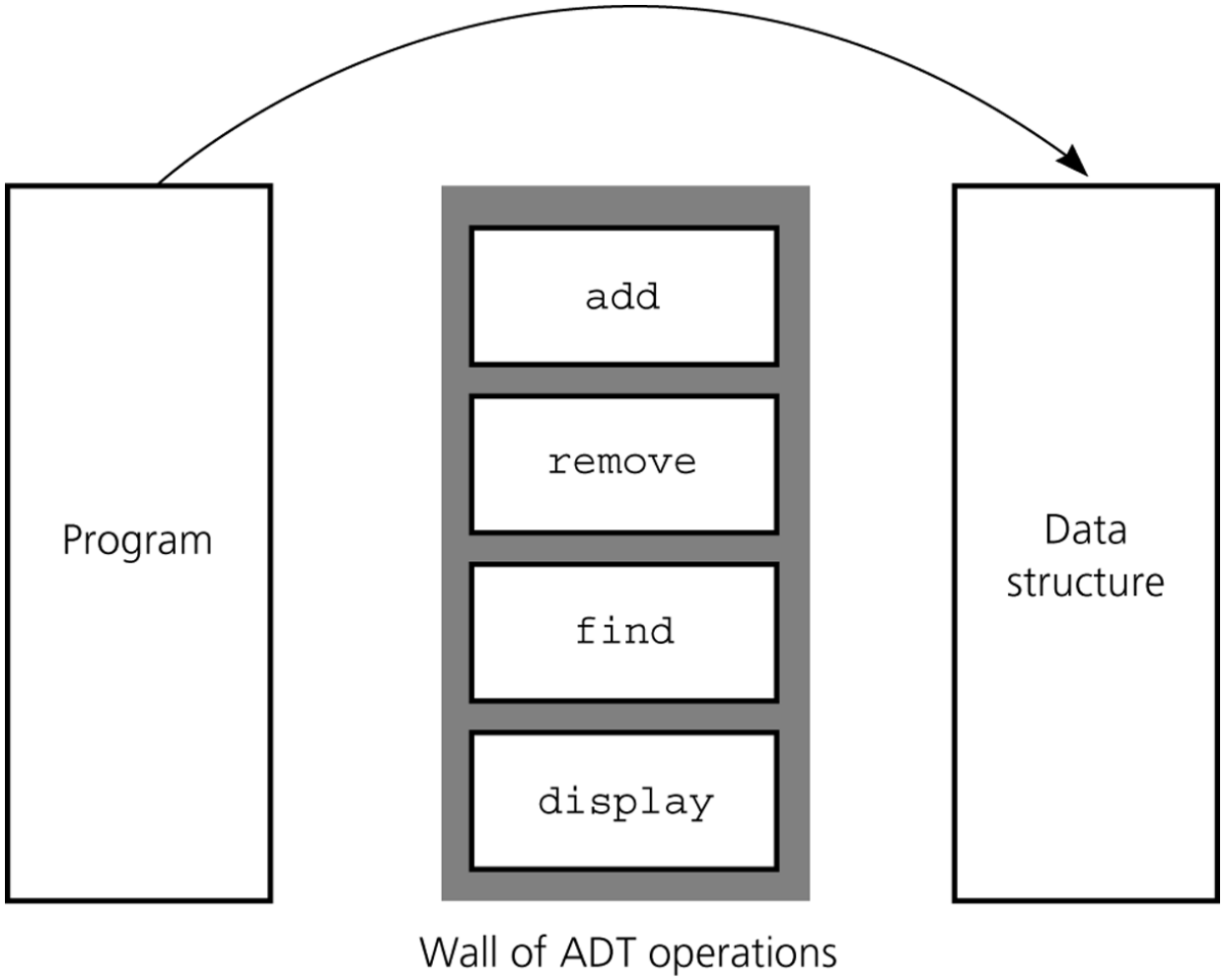
The Wall


function
displayList
Retrieve item at position
dataItem
Implementation
of ADT list
Wall of ADT operations

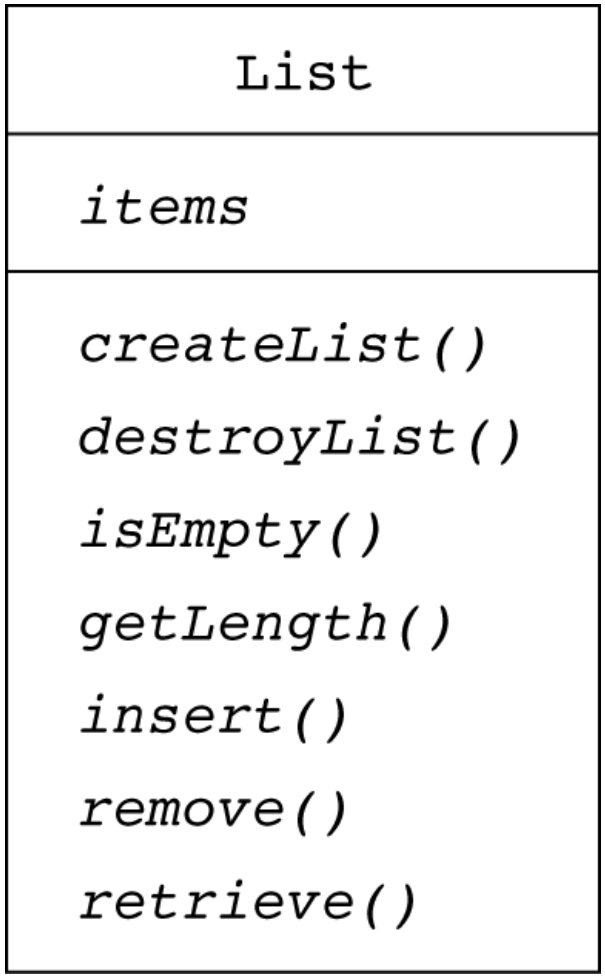
ADT operations provide access to a data structure



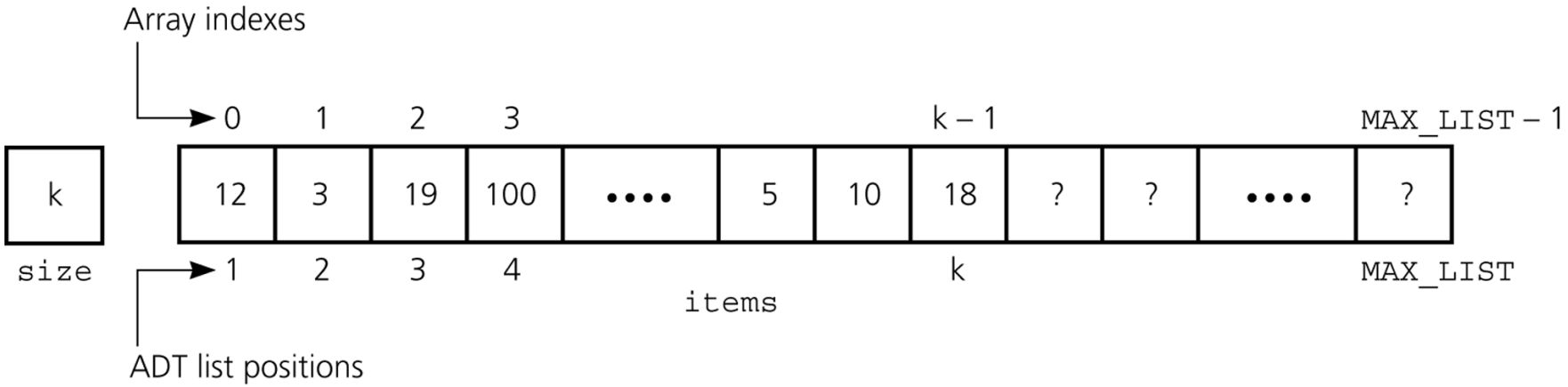
Violating the wall of ADT operations



ADT list


List
items
createList()
destroyList()
isEmpty()
getLength()
insert()
remove()
retrieve()

An array-based implementation of the ADT List



Pesudocode for the ADT list operations

createList() // creates an empty list

destroyList() // destroys a list

isEmpty():boolean // determines whether a list is empty

getLength() // Returns the number of items in the list

Retrieve(in index: integer, out dataItem: ListItemType, out success:boolean)

// copies the item at position index of a list into dataItem, if

// 1<=index<=getLength(). The list is left unchanged by this operation.

// The success flag indicates whether the retrieval was successful.

Pesudocode for the ADT list operations (cont.)

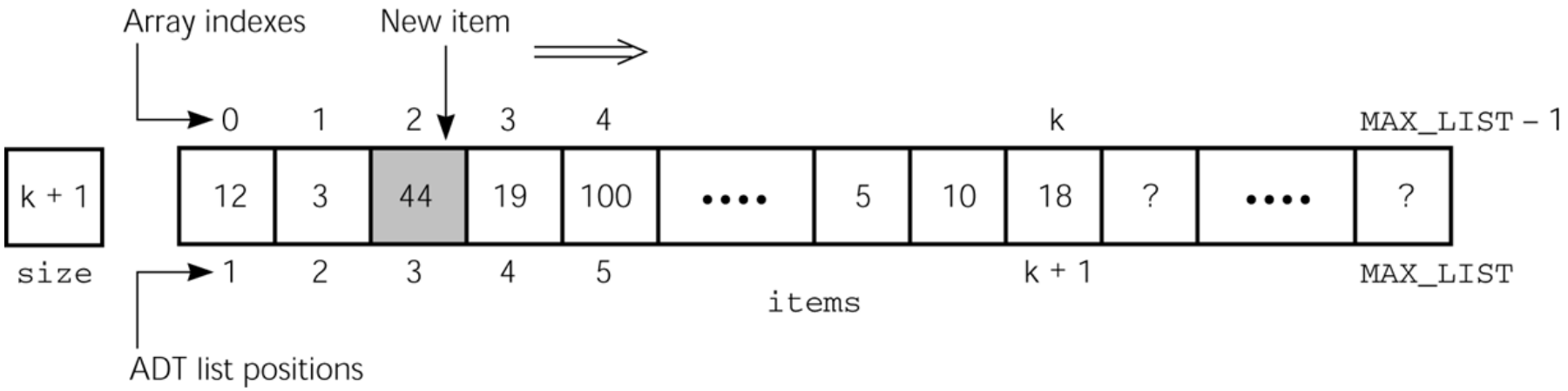
Insert(in index:integer, in newItem:ListItemType, out success:boolean)

// inserts newItem at position index of a list, if
1<=index<=getLength()+1

// if index<=getLegth(), items are renumbered as follows: the item at index becomes the item at index+1, the item at index+1 becomes the item at index+2, and so on.

// The success flag indicates whether the insertion was successful

Shifting items for insertion at position 3

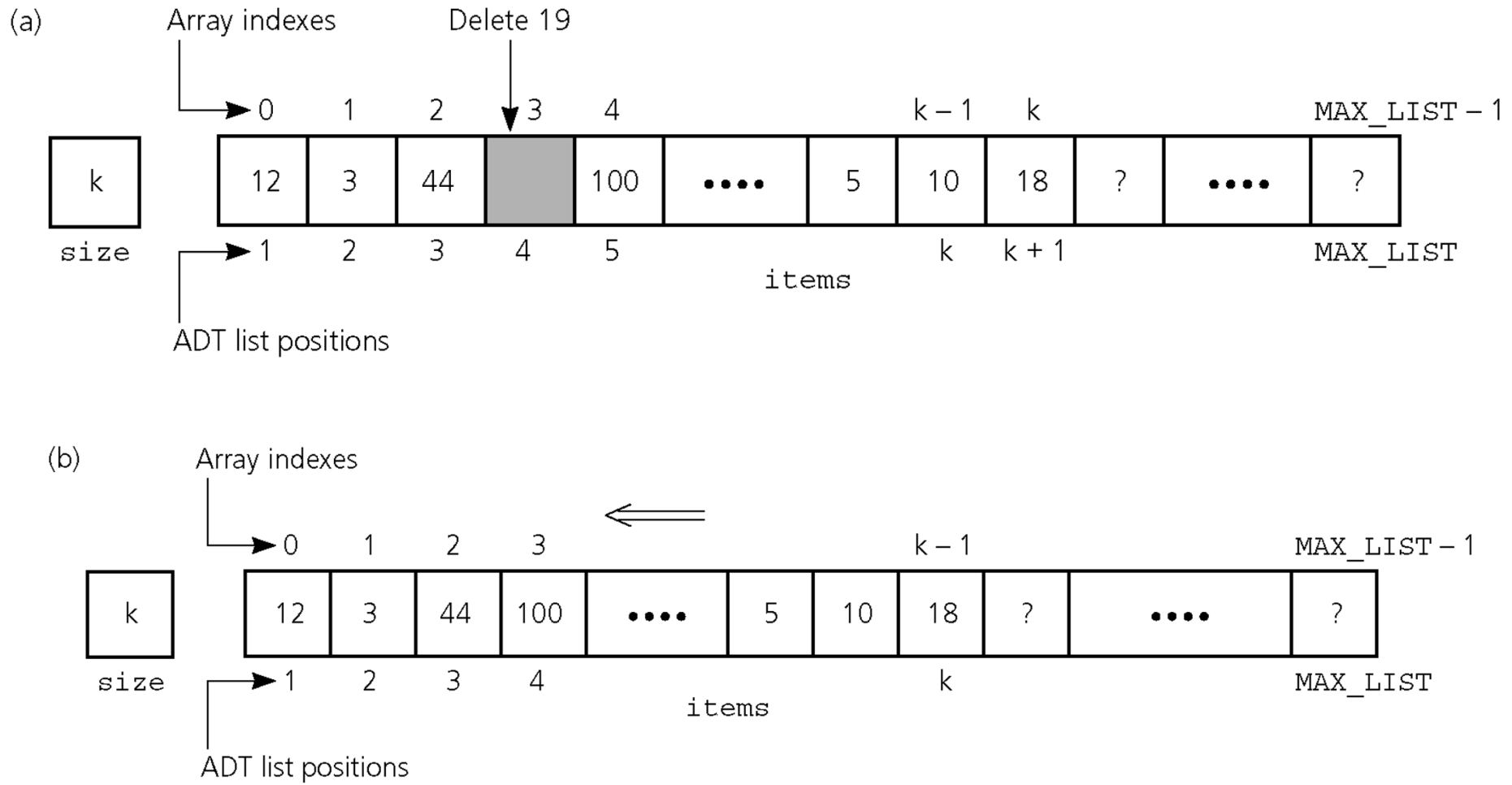


Pesudocode for the ADT list operations (cont.)

Remove(in index: integer, out success: boolean)

```
// removes the item at position index of a list, if
   1<=index<=getLength().
// If index < getLength(), items are renumbered as follows: The item
// at index+1 becomes the item at index, the item at index+2
// becomes the item at index+1, and so on.
// The success flag indicates whether the deletion was successful.
```

Deletion operation on list



Programming based on ADT specification

Example:

```
aList.createList();
aList.insert(1, milk, success);
aList.insert(2, eggs, success);
aList.insert(3, bread, success);
aList.insert(4, apple, success);
aList.insert(2, bacon, success);
aList.remove(3, success);
```

How to retrieve the 3rd item and store it in variable “item” ?

Application of the ADT list

```
DisplayList(in aList:List)
for (position=1 through aList.getLength())
{
        aList.retrieve(position, dataItem, success);
        Display dataItem;
}
```

Application of the ADT list

```
Replace(in aList:List, in i:integer, in newItem:ListItemType,
                        out success:boolean)

// replace the ith item on the aList with newItem.
// The success flag indicates whether the replacement was successful

{
        aList.remove(i, success);
        if (success)
                aList.insert(i, newItem, success);
}
```