Data Structures and Algorithms

List ADT implementations:

Array List

List ADT

- In computer science, a list or sequence is an abstract data type that represents a finite number of ordered values, where the same value may occur more than once.
- An instance of a list is a computer representation of the mathematical concept of a tuple or finite sequence; the (potentially) infinite analog of a list is a stream.
- If the same value occurs multiple times, each occurrence is considered a distinct item.

List ADT

- List ADT implementations:
  - –Array List
  - –Linked List
    - Singly Linked List
    - Doubly Linked List
    - Circular Linked List
      - –Singly Circular Linked List
      - –Doubly Circular Linked List

Array List

- List Abstract Data Type describes the behaviour of a basic list.
- Typically a list contains
  - add(element)push_front(element), push_back(element)
  - remove(index), remove(element)
  - get(index) , at(index)
- Array list which is based on dynamic array and the Linked list which is based on connected nodes are the two general implementions of List ADT.
- In addition to above function, Array List or Linked List may have functions speficific to implementation.

Array List

- Array list uses an array implement List ADT.
- Although the array is static, it is resized if needed.
- Each element added to list are stored in the array.
- For example assume we have a list of fruits and the below elements are added to list.

```
list1.add("apple");
list1.add("orange");
list1.add("banana");
list1.add("cherry");
```

Capacity=N

0	1	2	3	4	…	N-1
apple	orange	banana	cherry	-	-	-

The number of elements: list.length()=4

Array List

- All the elements of the array based list are sequential in memory. Hence adding an element to the end of the list or reading an element from an arbitrary location is independent from the size of the list.
- Inserting or removing an element which requires neighbor elements can be costly.
- Computational complexities:
  - add(element) → O(1)
  - insert(element, index) → O(n)
  - remove(element) → O(n)
  - get(index) or at(index) → O(1)

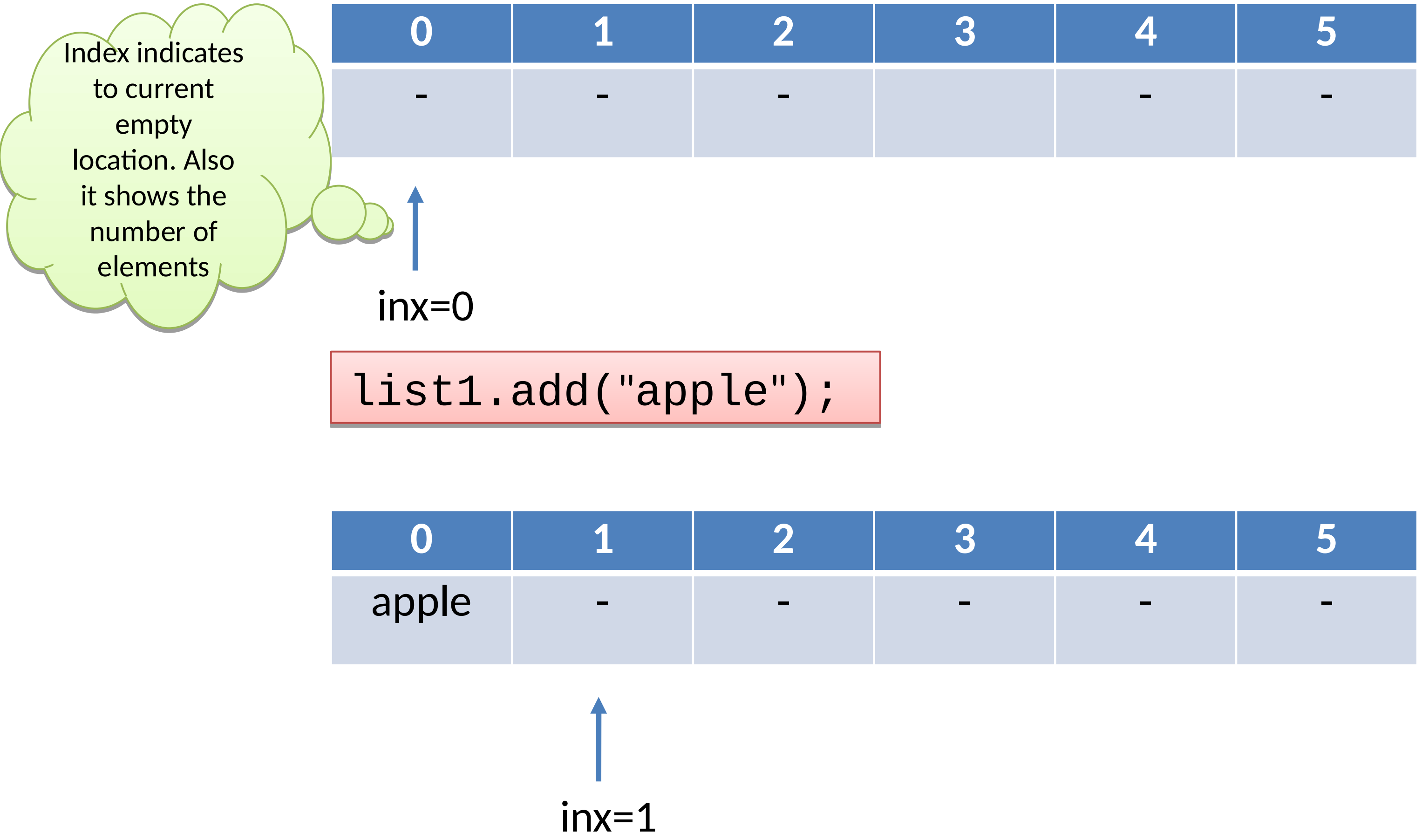
Array List

- Initially we will consider a list of integers for simplicity. Later we will convert it to a template class.
- The class structure can be used as below:

```
#include <iostream>
using namespace std;
#define MAX_SIZE 10
class ArrayList
{
private:
      int *Elements;
      int n;
      int capacity;
public:
ArrayList(int capacity = MAX_SIZE ) //constructor
{
      this->capacity = capacity;
      Elements = new int[capacity]; //array on heap
      n = 0;
}
void add(int veri); //push_back
int get(int pos);
void remove(int pos);
int length();
void clear();
~ArrayList() //destructor
{delete[] Elements;}
};
```

Adding an element

- Add operation



Index indicates to current empty location. Also it shows the number of elements

0	1	2	3	4	5
-	-	-		-	-

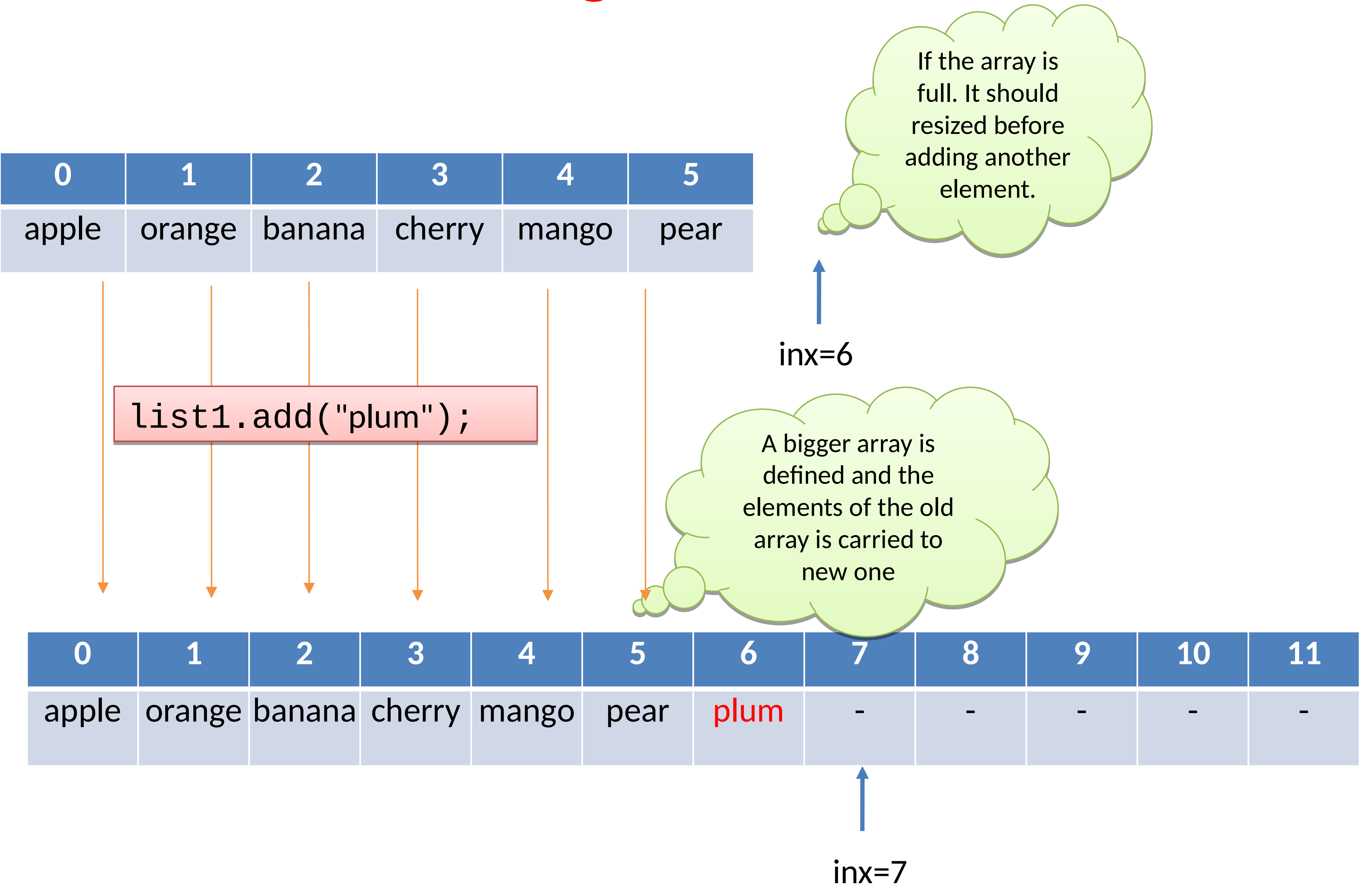
inx=0

```
list1.add("apple");
```

0	1	2	3	4	5
apple	-	-	-	-	-

inx=1

Adding an element



0	1	2	3	4	5
apple	orange	banana	cherry	mango	pear

```
list1.add("plum");
```

If the array is full. It should resized before adding another element.

inx=6

A bigger array is defined and the elements of the old array is carried to new one

0	1	2	3	4	5	6	7	8	9	10	11
apple	orange	banana	cherry	mango	pear	plum	-	-	-	-	-

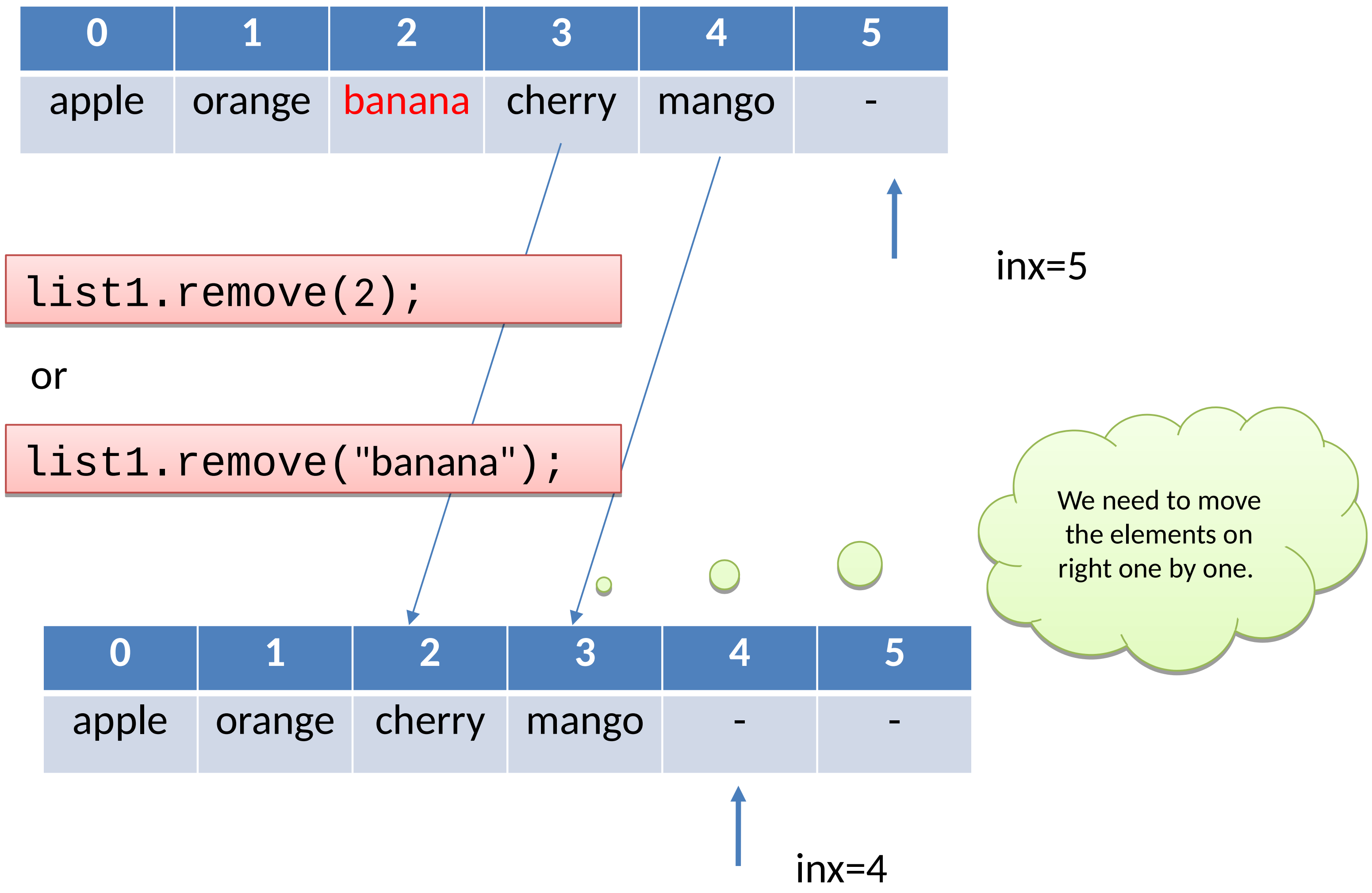
inx=7

Adding an element

```
void add(int data) //push_back
{
    //is capacity full
    if (n == capacity)
    resize();

    Elements[n] = data;
    n++;
}
void resize() { // increase the array size two times
    int newcap= capacity>0? 2 * capacity:1;
    int *Temp = new int[newcap];
    for (int i = 0; i < n; i++)
    Temp[i] = Elements[i];
    delete[] Elements; //delete the old array
    Elements = Temp; //use new array
    capacity = newcap;
    }
```

Removing an element



0	1	2	3	4	5
apple	orange	banana	cherry	mango	-

```
list1.remove(2);
```

or

```
list1.remove("banana");
```

0	1	2	3	4	5
apple	orange	cherry	mango	-	-

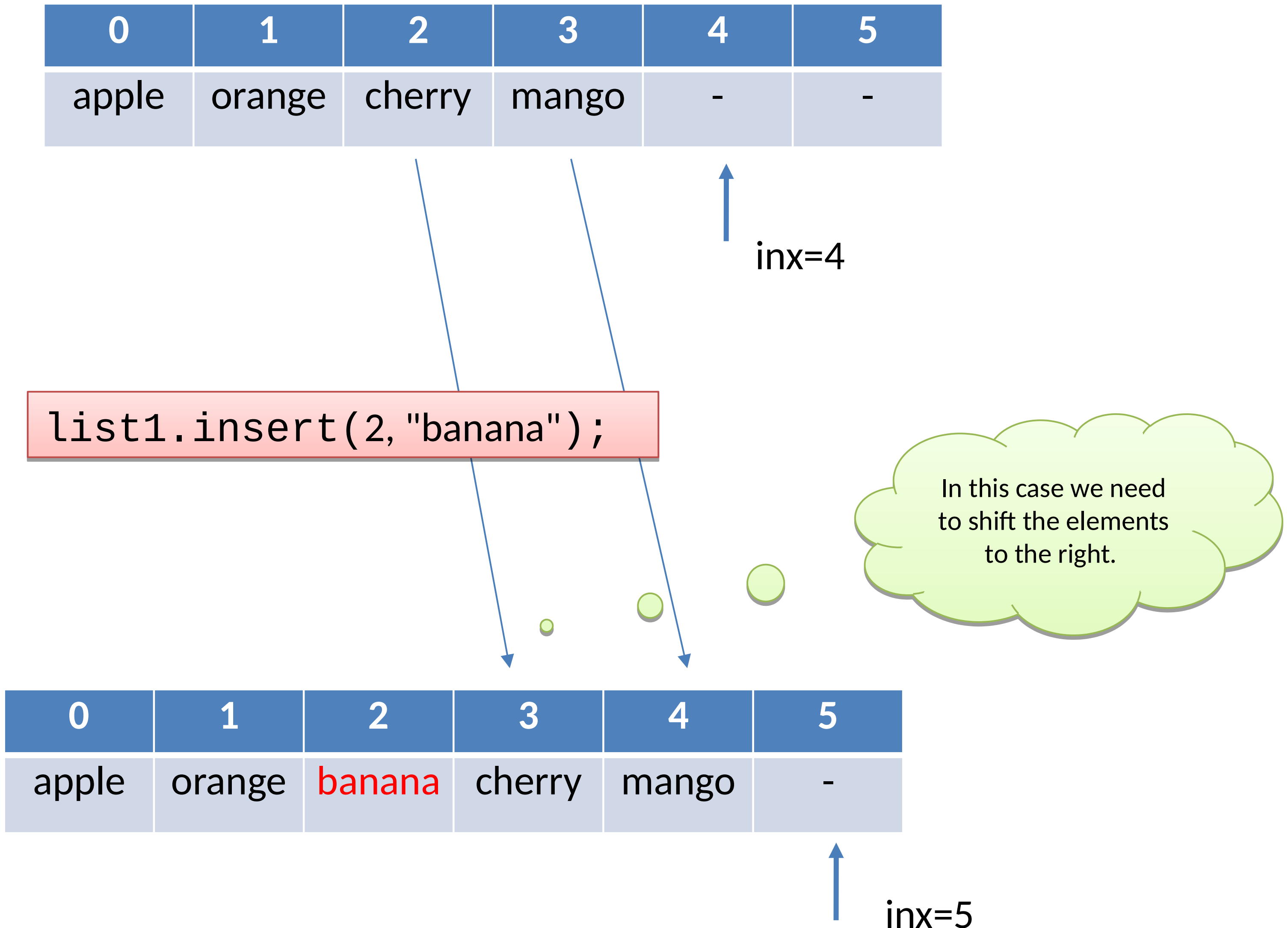
Removing an element

```
void remove(int pos) {
    //check if the entered position is
    valid
    if (pos < 0 || pos >= n) throw
    exception();

    //reduce the number of elements
    n--;

    //shift all the elements to the left
    for (int i = pos; i < n; i++)
    Elements[i] = Elements[i + 1];
}
```

Inserting an element



0	1	2	3	4	5
apple	orange	cherry	mango	-	-

```
list1.insert(2, "banana");
```

0	1	2	3	4	5
apple	orange	banana	cherry	mango	-

Inserting an element

```
void insert(int pos, int data)
{
    //check if the entered position is valid
    if (pos < 0 || pos >= n) throw exception();

    if (n == capacity) { //is capacity full
        cout << "Capacity is full. Resizing the array" <<
    endl;
        resize();
    }

    //make room for the new element at "pos"
    for (int i =n-1; i >= pos; i--)
        Elements[i+1] = Elements[i];

    //write the new element
    Elements[pos] = data;
    //increase the number of elements
    n++;
}
```

Other functions

```
void print() {
cout << "--Elements:" << endl;
for (int i = 0; i < n; i++)
cout << "\t" << Elements[i];
cout << endl;
}

void clear() {
delete[] Elements;
Elements = new
int[capacity]; //array on heap
n = 0;
}

int length() {
return n;
}
```

Example usage

```
int main() {
ArrayList *list1 = new ArrayList(3);
list1->add(10);  //0
list1->add(8);   //1
list1->add(33); //2
list1->add(11);  //3
list1->add(55);  //4
list1->add(88);  //5
list1->add(21);
list1->insert(3, 100);

//try catch block example
try {
cout << "list1->get(5)=" << list1->get(5) << endl;
cout << "list1->get(50)=" << list1->get(50) << endl;
} catch (exception &e) {
cout << e.what() << endl;
}

list1->print();
cout << "list1->length()=" << list1->length() << endl;
list1->remove(1);
list1->remove(3);

list1->print();
cout << "list1->length()=" << list1->length() << endl;

list1->clear();
list1->print();
cout << "list1->length()=" << list1->length() << endl;

return 0;
}
```