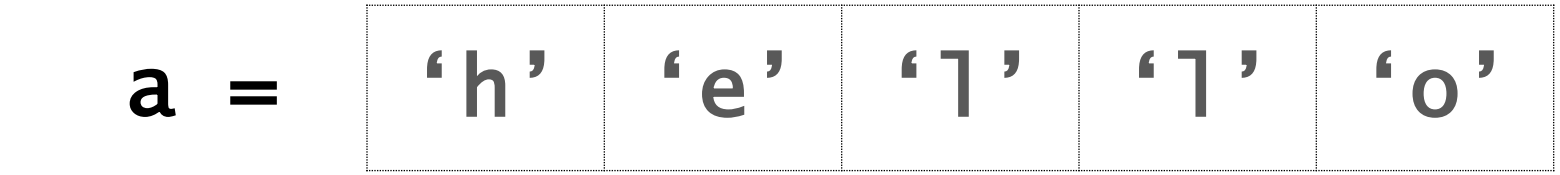
Array, Vector, List

Shusen Wang

Array

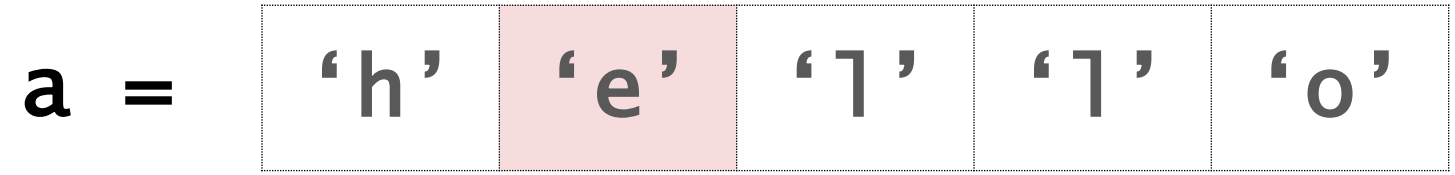
Array



- `char a[5] = {'h', 'e', 'l', 'l', 'o'};`

Array: Fixed size, contiguous memory.

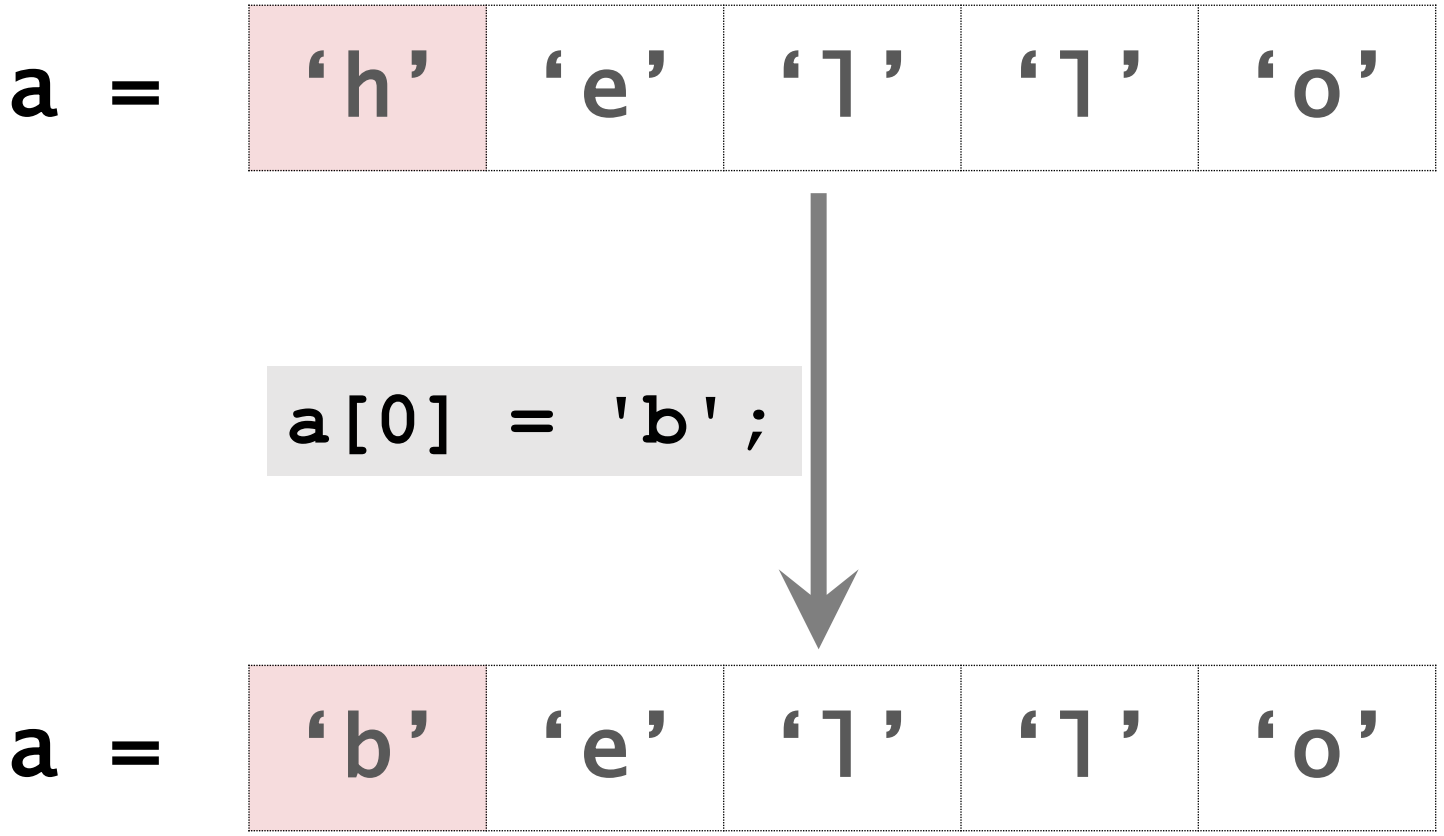
Array




```
// random access
```

- `char x = a[1];   // x is 'e'`

Array



Array

- We can create a size-n array in this way:

```
char a[n];
```

- When writing the code, n must be known.
- What if n is unknown until program is running?

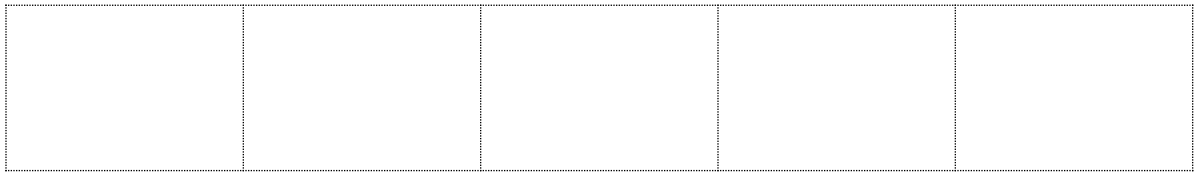
Dynamic Allocation of Arrays

a = NULL

- `char* a = NULL;`
- `int n; // array size`
- `cin >> n; // read in the size, e.g., get n=5`

Dynamic Allocation of Arrays

a =



- `char* a = NULL;`
- `int n; // array size`
- `cin >> n; // read in the size, e.g., get n=5`
- `a = new char[n];`

Dynamic Allocation of Arrays

a =	‘h’	‘e’	‘l’	‘l’	‘o’

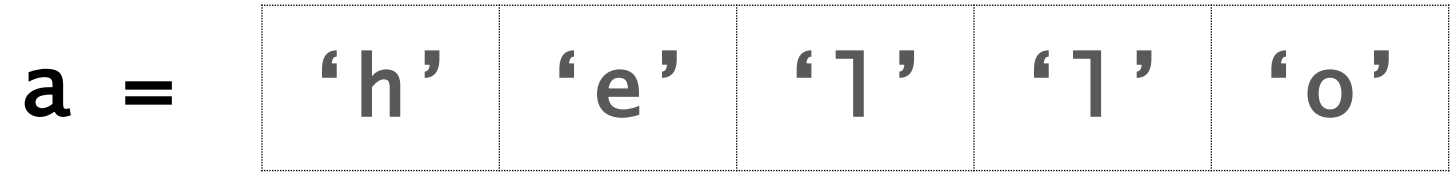
```
// store something in the array
```

- `a[0] = 'h';`
- `a[1] = 'e';`

⋮

- `a[4] = 'o';`

Dynamic Allocation of Arrays




```
// When done, free memory.
// Otherwise, memory leak can happen.
```

- **`delete [] a;`**

Dynamic Allocation of Arrays

```
a =  NULL
```

```
// When done, free memory.
// Otherwise, memory leak can happen.
```

- **`delete [] a;`**
- **`a = NULL;`**

Properties of Array

a =	'h'	'e'	'l'	'l'	'o'

1. The size is fixed. (New elements cannot be appended.)
2. Random access using `a[i]` has $O(1)$ time complexity.
3. Removing an element in the middle has $O(n)$ time complexity. (Move the remaining items leftward.)

Vector

Vector

- Vector is almost the same to array.
- The main difference is that vector's capacity can automatically grow.
- New elements can be appended using `push_back()` in $O(1)$ time (on average).
- The last element can be removed using `pop_back()` in $O(1)$ time.

Vector

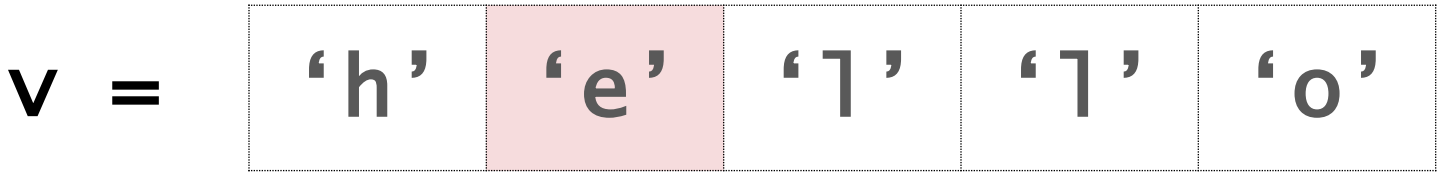
v =

‘h’	‘e’	‘l’	‘l’	‘o’

- `vector<char> v = {'h', 'e', 'l', 'l', 'o'};`

Vector: dynamic size, contiguous memory.

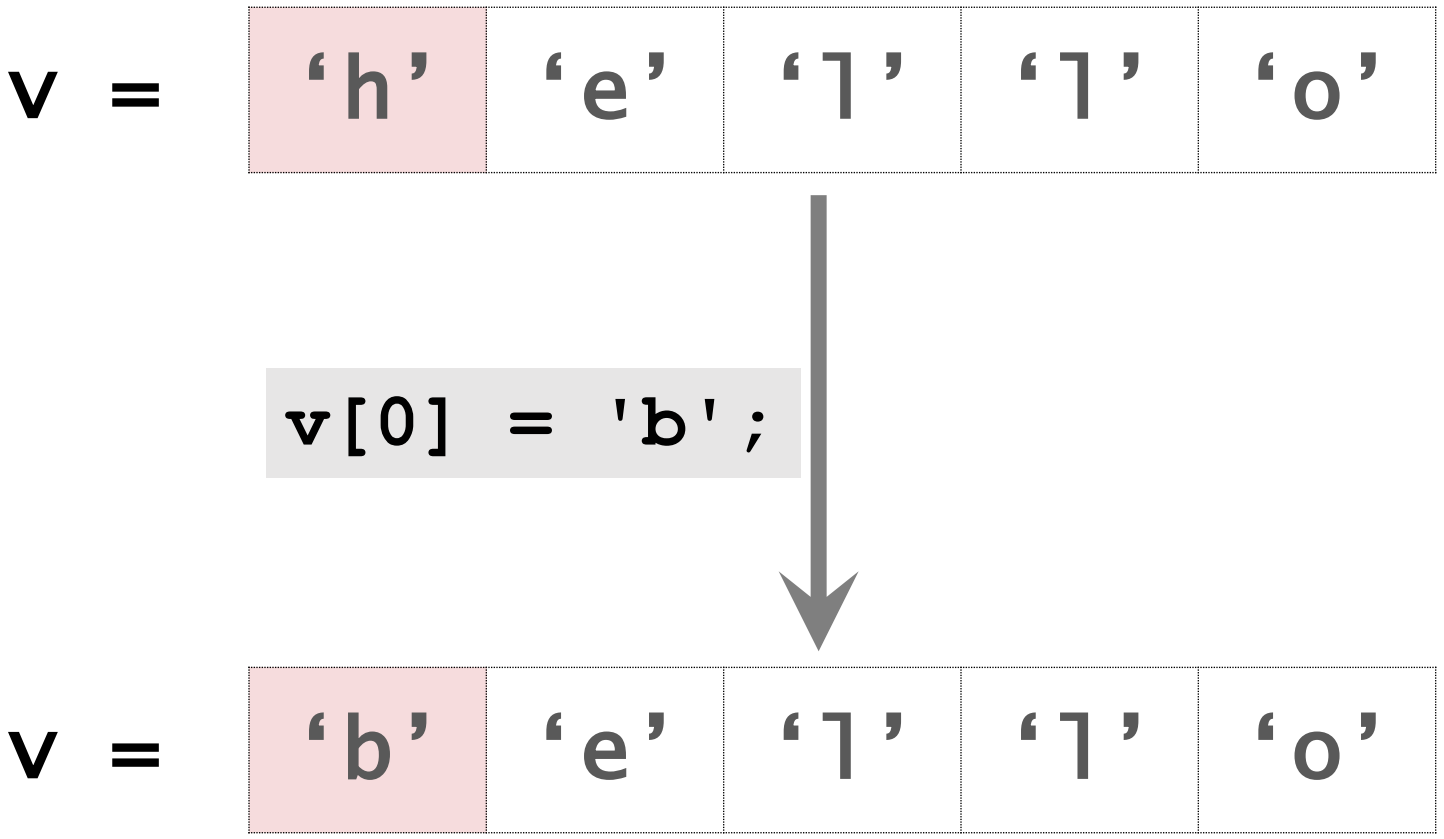
Vector



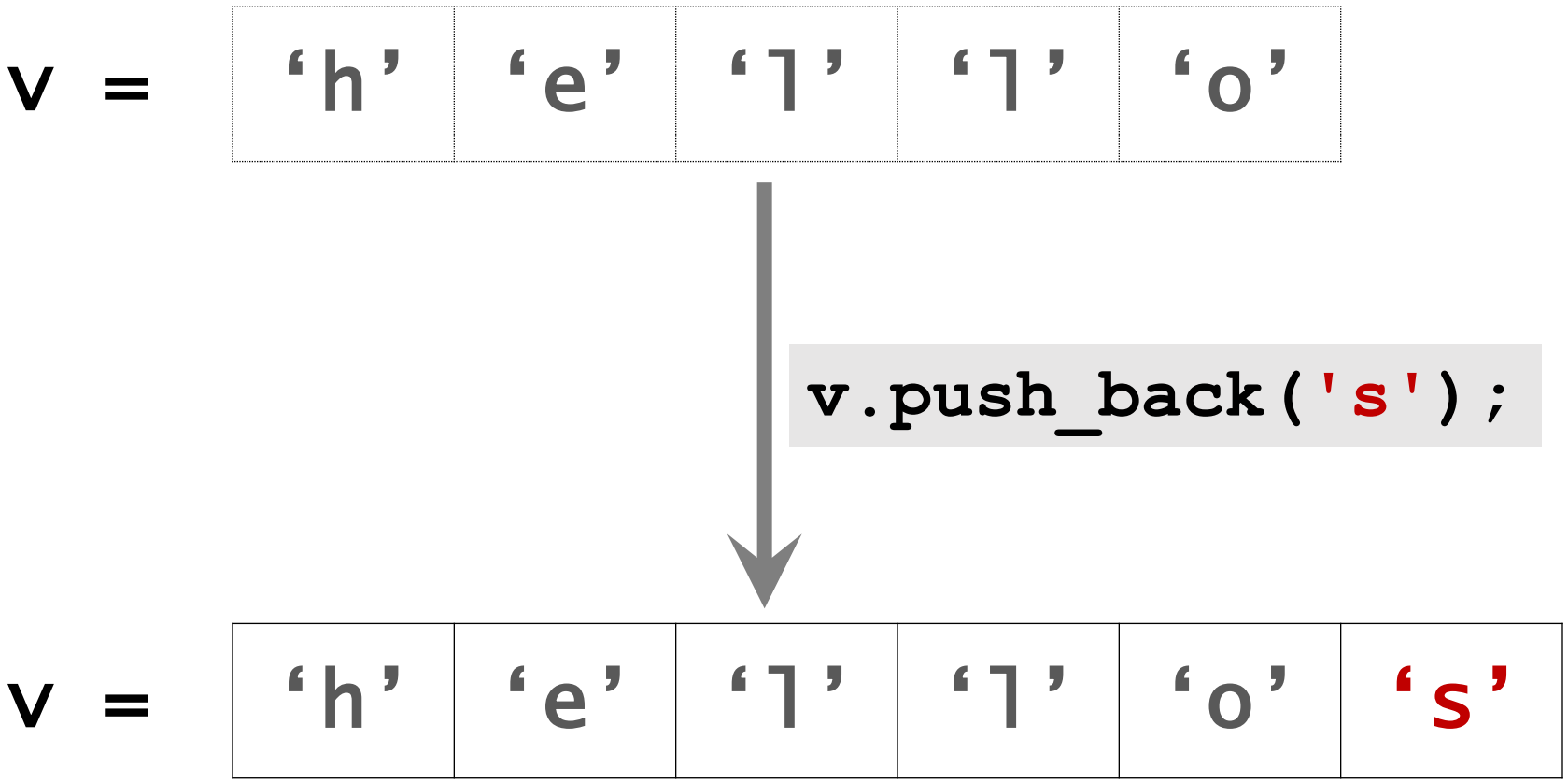

```
// random access
```

- `char x = v[1];   // x is 'e'`

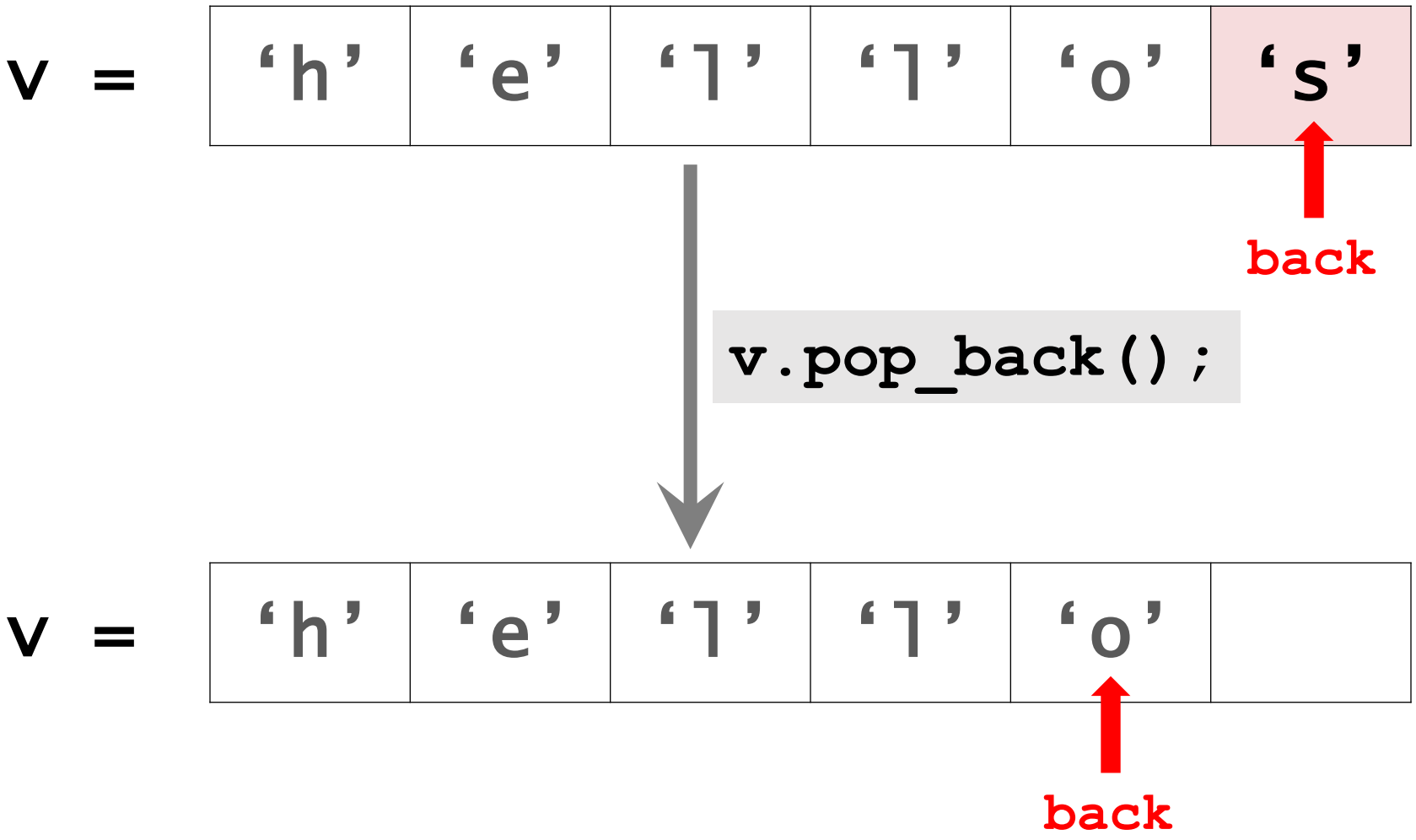
Vector



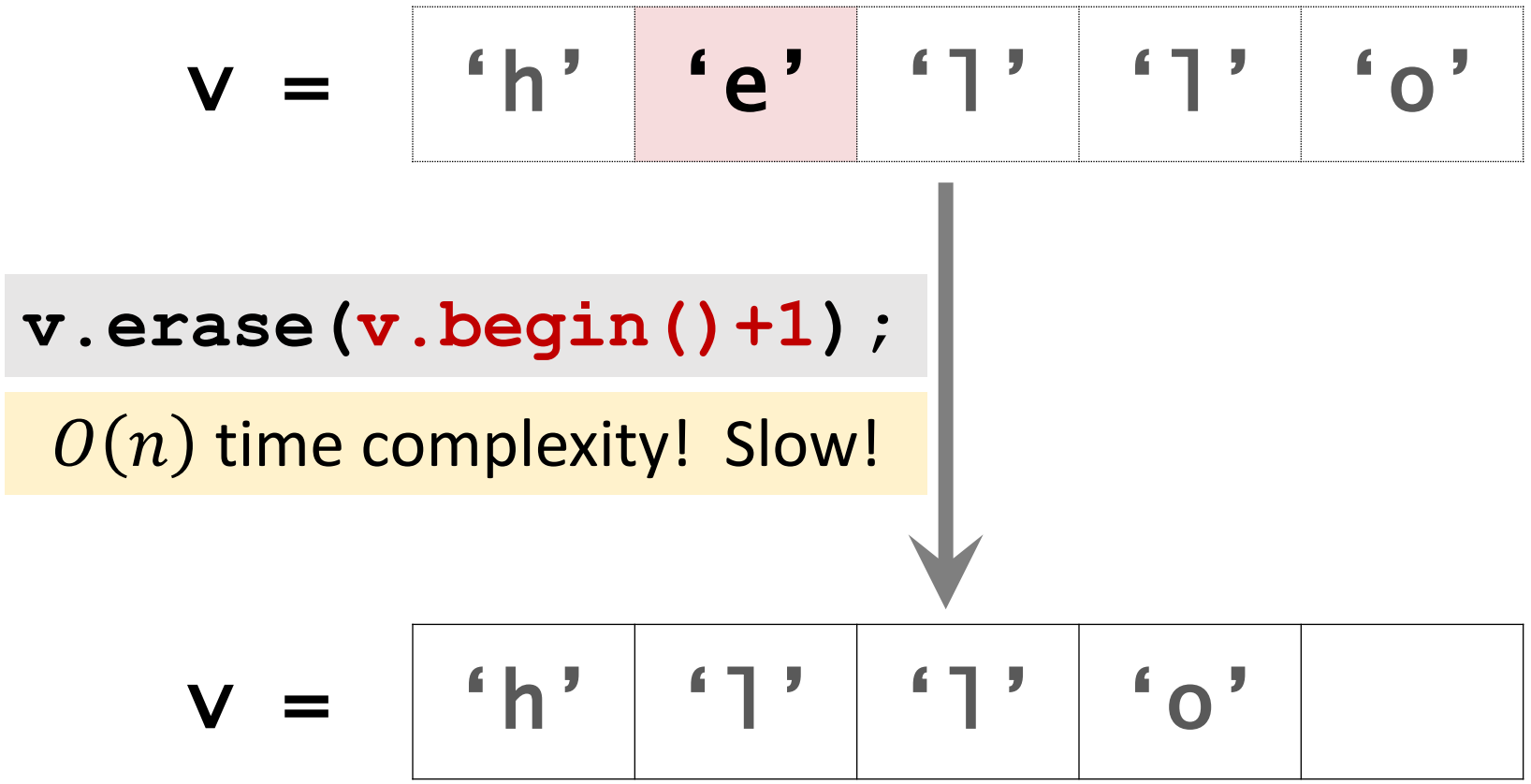
Insert



Delete



Delete



Vector capacity can grow

- `vector<char> v(100);`
- `cout << v.size();` `// print “100”`
- `cout << v.capacity();` `// print “100”`

Vector capacity can grow

- `vector<char> v(100);`
- `cout << v.size();` `// print “100”`
- `cout << v.capacity();` `// print “100”`

- `v.push_back('x');`
- `cout << v.size();` `// print “101”`
- `cout << v.capacity();` `// print “200”`

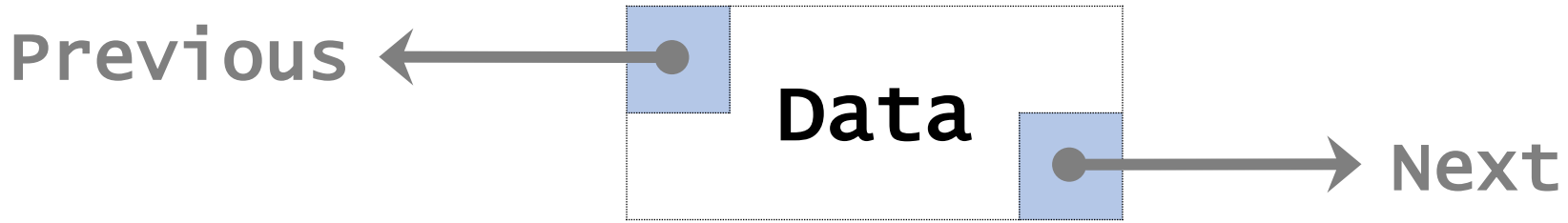
Vector capacity can grow

What happened when size is going to exceed capacity?

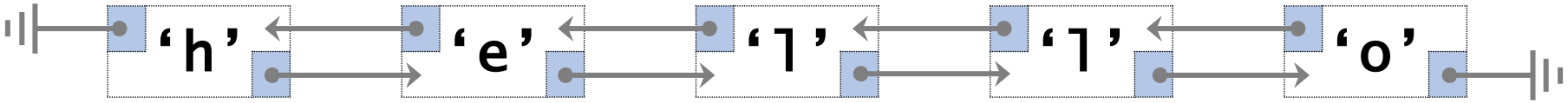
- Create a new array of size 200.
- Copy the 100 elements from the old array to the new.
- Put the new element at the 101st position.
- Free the old array from memory.

List

A Node



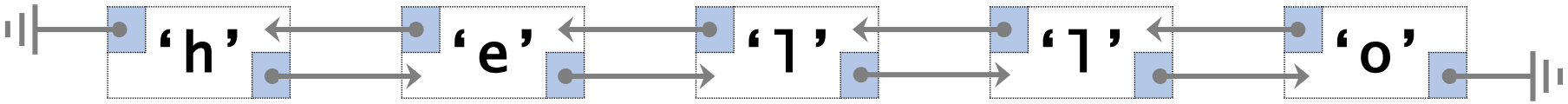
Doubly Linked List



- `list<char> l = {'h', 'e', 'l', 'l', 'o'};`

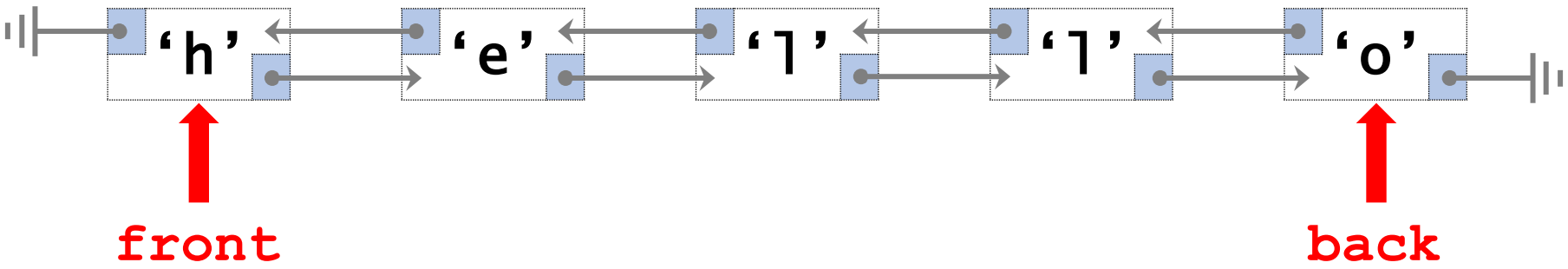
List: dynamic size, not contiguous memory.

Doubly Linked List



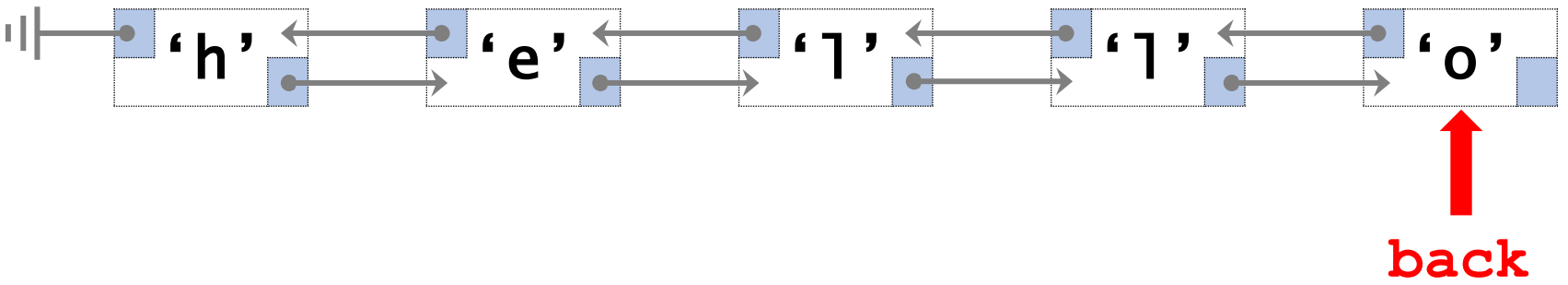
- `cout << l[2]; // does not work`
- `l[0] = 'a';  // does not work`

Doubly Linked List



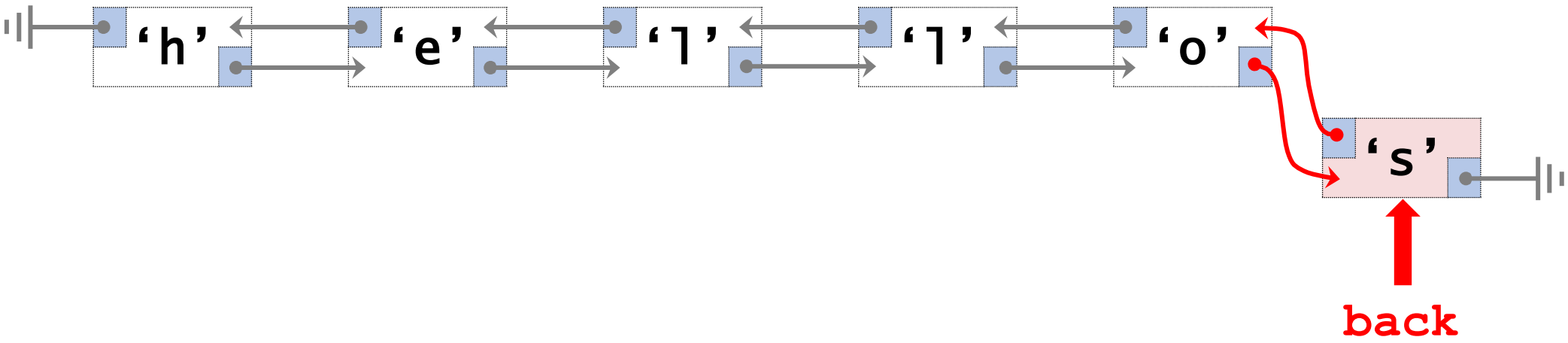
- `cout << l.front(); // print 'h'`
- `cout << l.back();  // print 'o'`

Insertion




```
l.push_back('s');
```

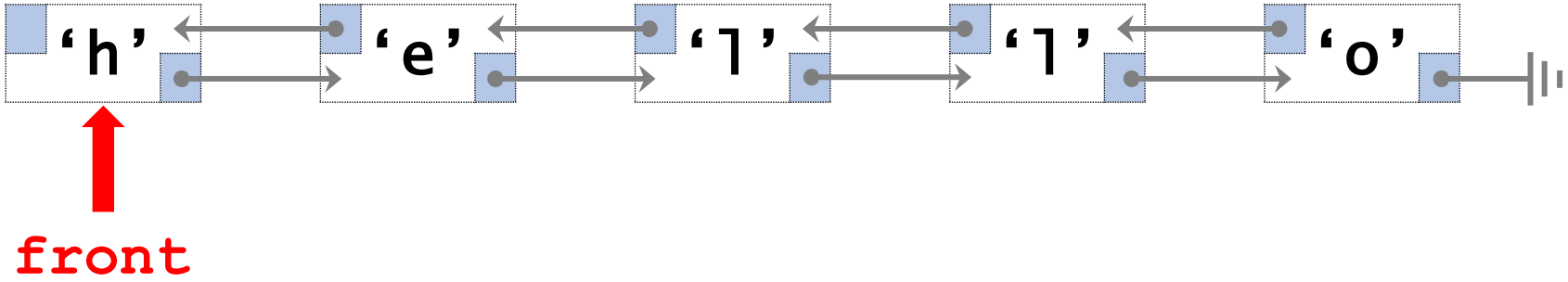
Insertion




```
l.push_back('s');
```

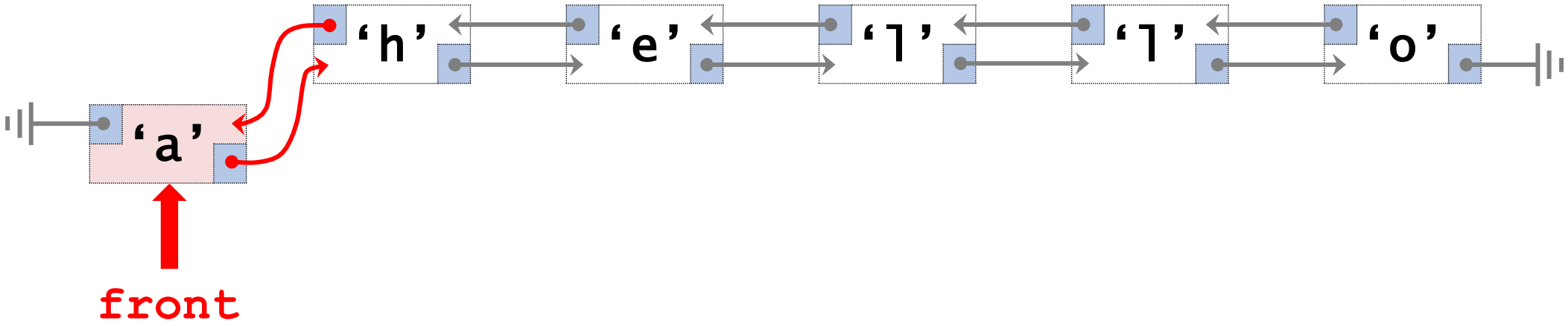
Only O(1) time.

Insertion




```
l.push_front('a');
```

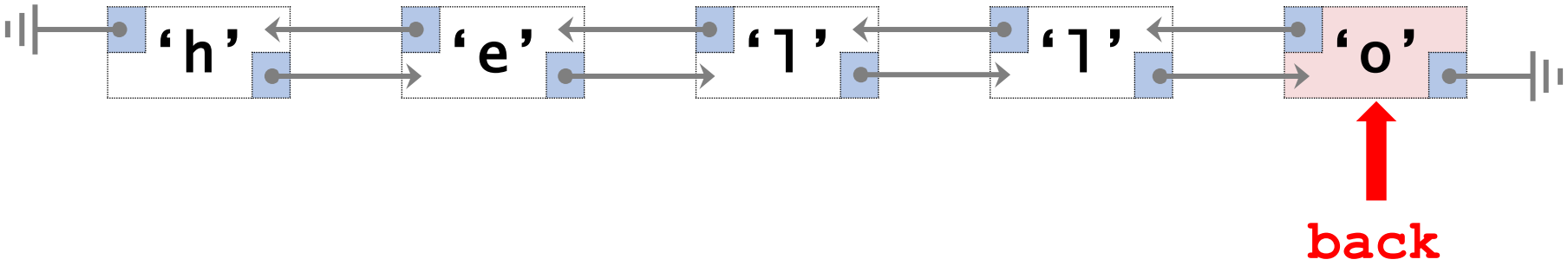
Insertion




```
l.push_front('a');
```

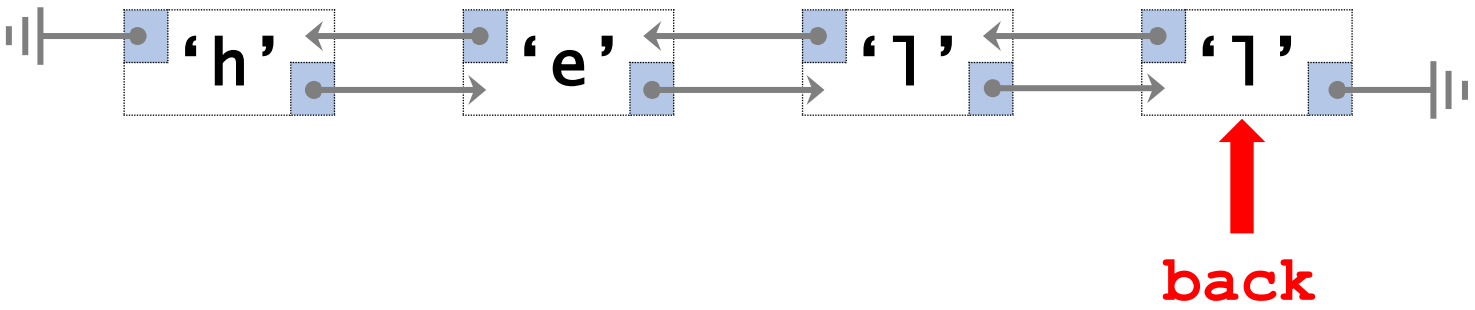
Only O(1) time.

Delete Node




```
l.pop_back();
```

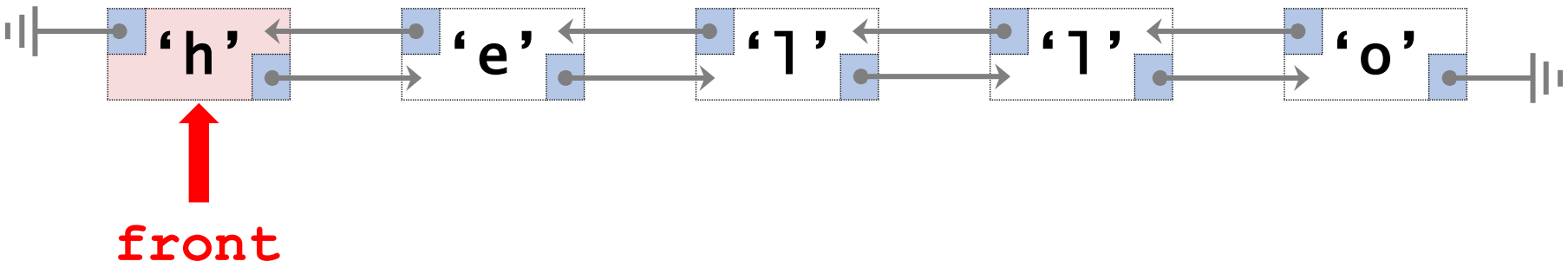
Delete Node




```
l.pop_back();
```

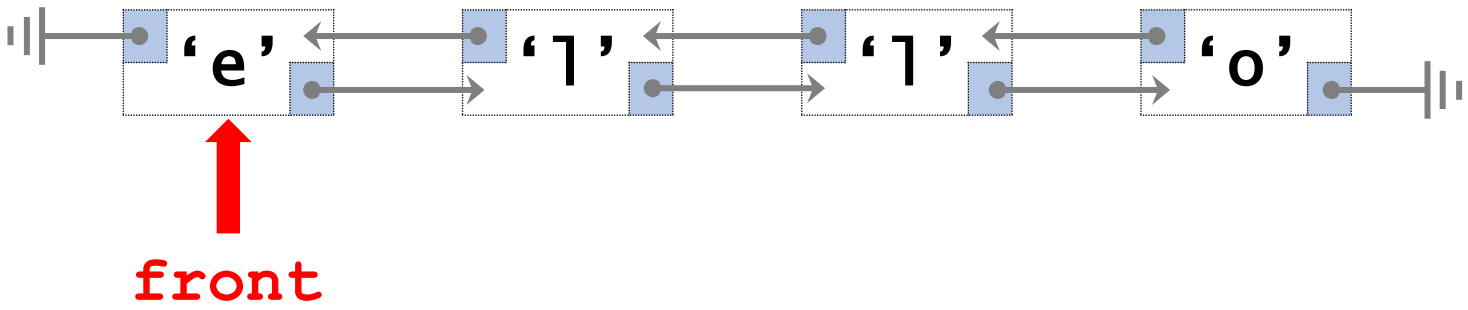
Only O(1) time.

Delete Node




```
l.pop_front();
```

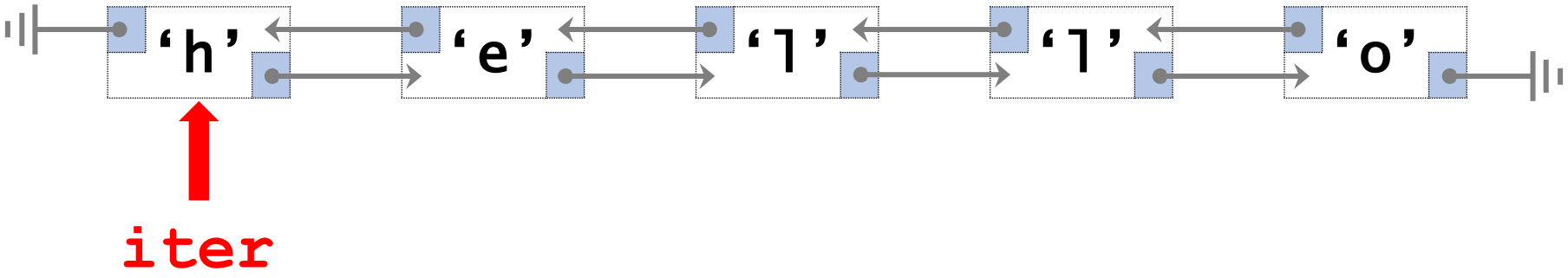
Delete Node




```
l.pop_front();
```

Only O(1) time.

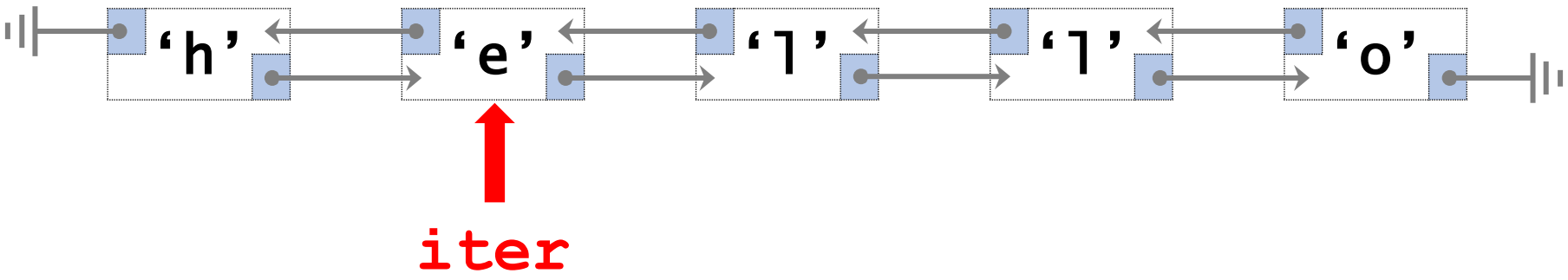
Iterator



- `list<char>::iterator iter = l.begin();`
- `cout << *iter; // print 'h'`

- `iter++;`

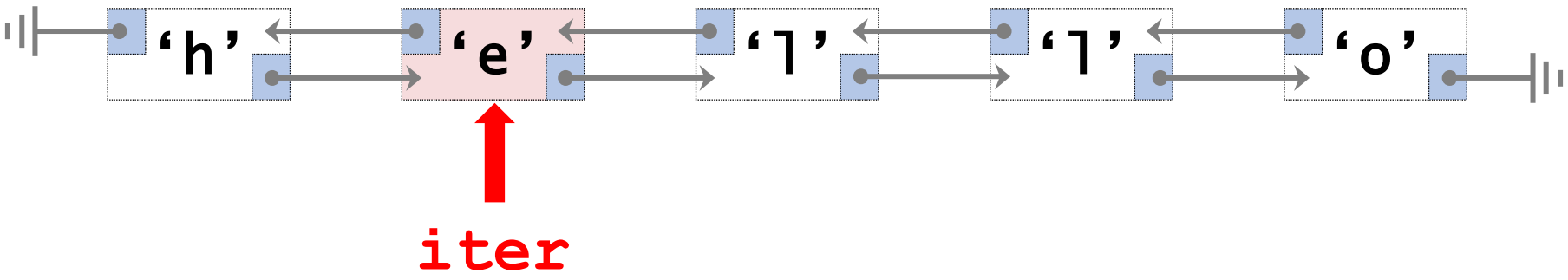
Iterator



- `list<char>::iterator iter = l.begin();`
- `cout << *iter; // print 'h'`

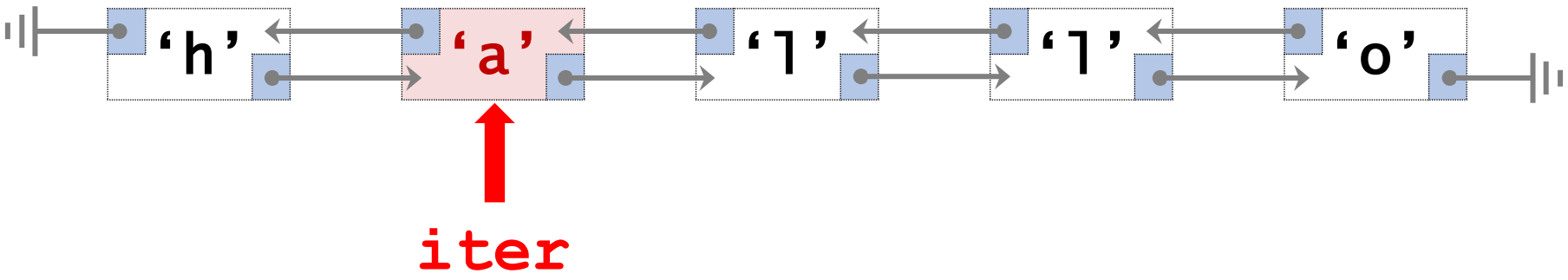
- `iter++;`
- `cout << *iter; // print 'e'`

Iterator



- `*iter = 'a'; // change 'e' to 'a'`

Iterator



- `*iter = 'a'; // change 'e' to 'a'`

Summary

Properties

	Array	Vector	List
Size	fixed	can increase and decrease	can increase and decrease

Properties

	Array	Vector	List
Size	fixed	can increase and decrease	can increase and decrease
Memory	contiguous	contiguous	not contiguous

Time Complexities

	`Array`	`Vector`	`List`
Rand Access	$O(1)$	$O(1)$	---

Time Complexities

	`Array`	`Vector`	`List`
Rand Access	$O(1)$	$O(1)$	---
`push_back()`	---	$O(1)$ (average)	$O(1)$

Time Complexities

	`Array`	`Vector`	`List`
Rand Access	$O(1)$	$O(1)$	---
`push_back()`	---	$O(1)$ (average)	$O(1)$
`Pop_back()`	---	$O(1)$	$O(1)$

Time Complexities

	`Array`	`Vector`	`List`
Rand Access	$O(1)$	$O(1)$	---
`push_back()`	---	$O(1)$ (average)	$O(1)$
`Pop_back()`	---	$O(1)$	$O(1)$
`insert()`	---	$O(n)$ (average)	$O(1)$

Time Complexities

	`Array`	`Vector`	`List`
Rand Access	$O(1)$	$O(1)$	---
`push_back()`	---	$O(1)$ (average)	$O(1)$
`Pop_back()`	---	$O(1)$	$O(1)$
`insert()`	---	$O(n)$ (average)	$O(1)$
`erase()`	---	$O(n)$ (average)	$O(1)$

Which shall we use?

- **Array:** Fixed size throughout.
- **Vector:**
  - Random access, i.e., read or write the i-th element for any i.
  - Insertion or deletion only at the end.
  - Occasional insertion or deletion in the front or middle.
- **List:**
  - Sequentially visiting elements; no random access.
  - Frequent insertion or deletion at any position.

Thank You!