Better Code: Data Structures

Sean Parent | Principal Scientist

Andres Amador

Better Code

- Regular Types
  - Goal: Implement Complete and Efficient Types
- Algorithms
  - Goal: No Raw Loops
- Data Structures
  - Goal: No Incidental Data Structures
- Runtime Polymorphism
  - Goal: No Raw Pointers
- Concurrency
  - Goal: No Raw Synchronization Primitives
- …

Better Code

- Regular Types
  - Goal: Implement Complete and Efficient Types
- Algorithms
  - Goal: No Raw Loops
- Data Structures
  - Goal: No Incidental Data Structures
- Runtime Polymorphism
  - Goal: No Raw Pointers
- Concurrency
  - Goal: No Raw Synchronization Primitives
- …

Goal: No incidental data structures

What is an *incidental* data structure?

What is a data structure?

What is a data structure?

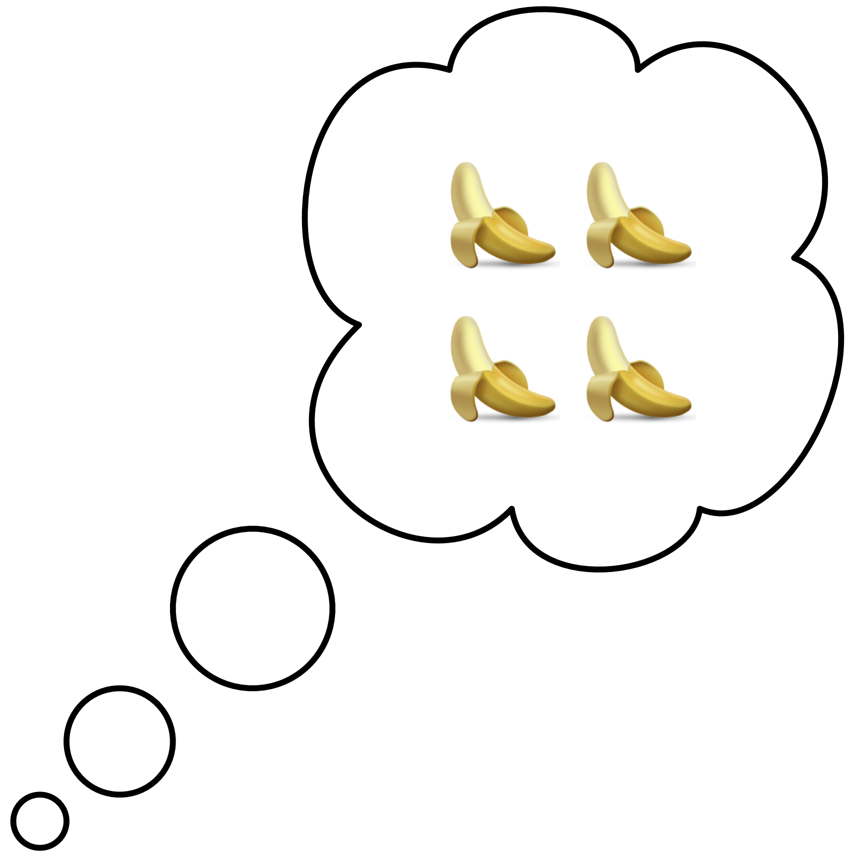
Definition: A data structure is a format for organizing and storing data.

What is a structure?

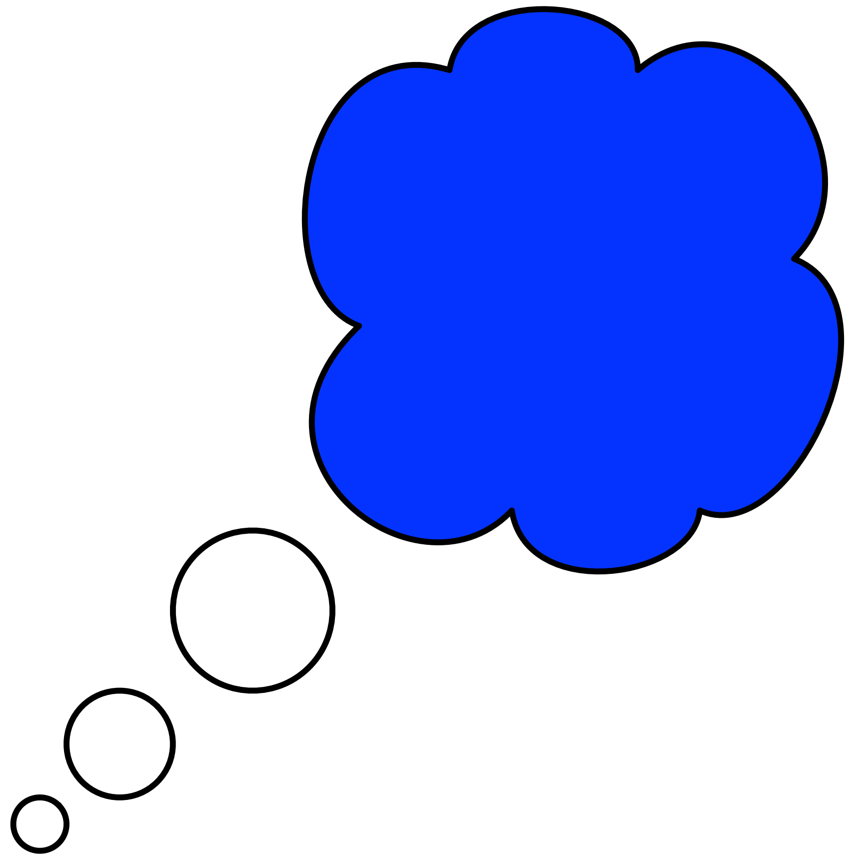
What is a structure?

Definition: A structure on a set consists of additional entities that, in some manner, relate to the set, endowing the collection with meaning or significance.

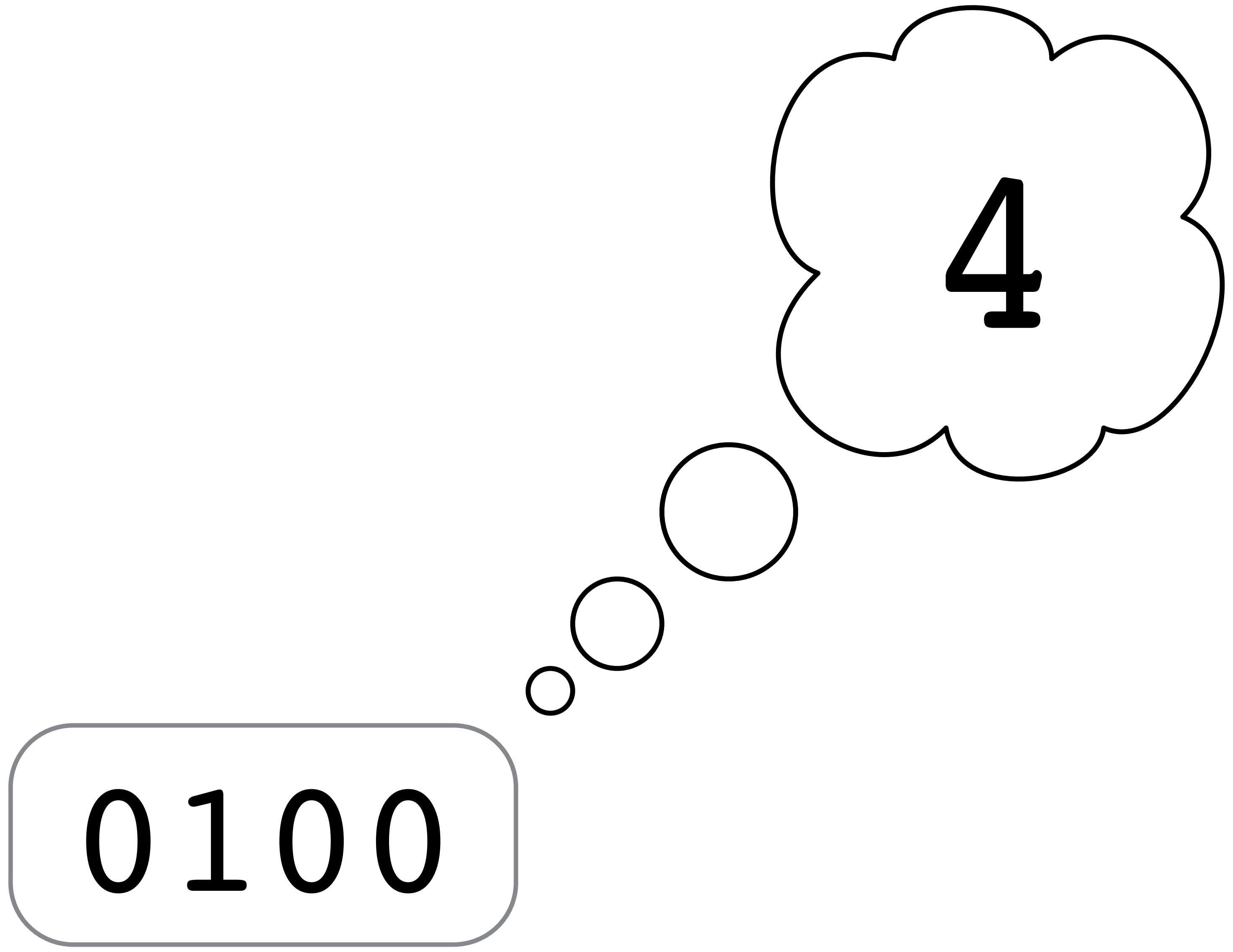
0100



0100



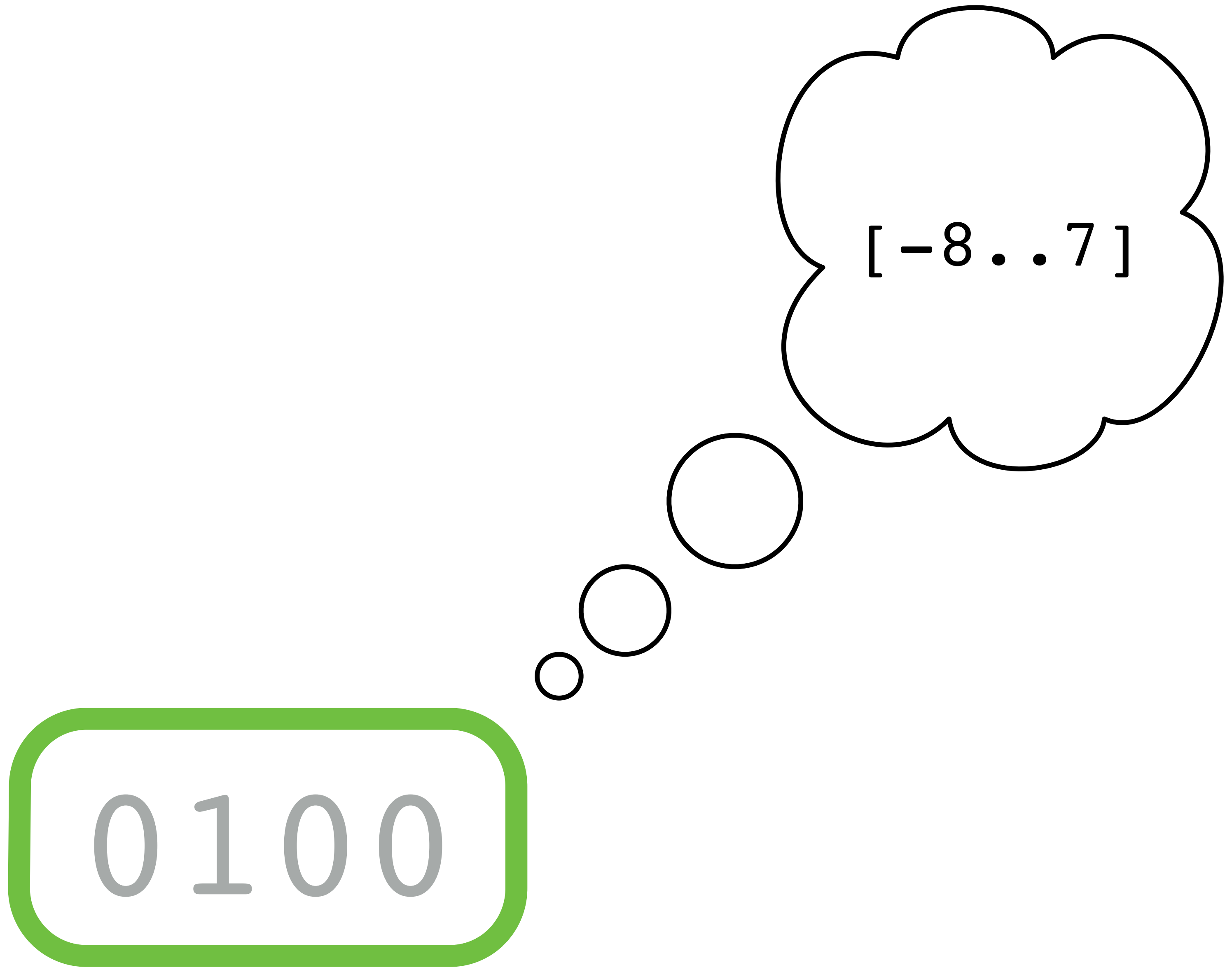
0100



4

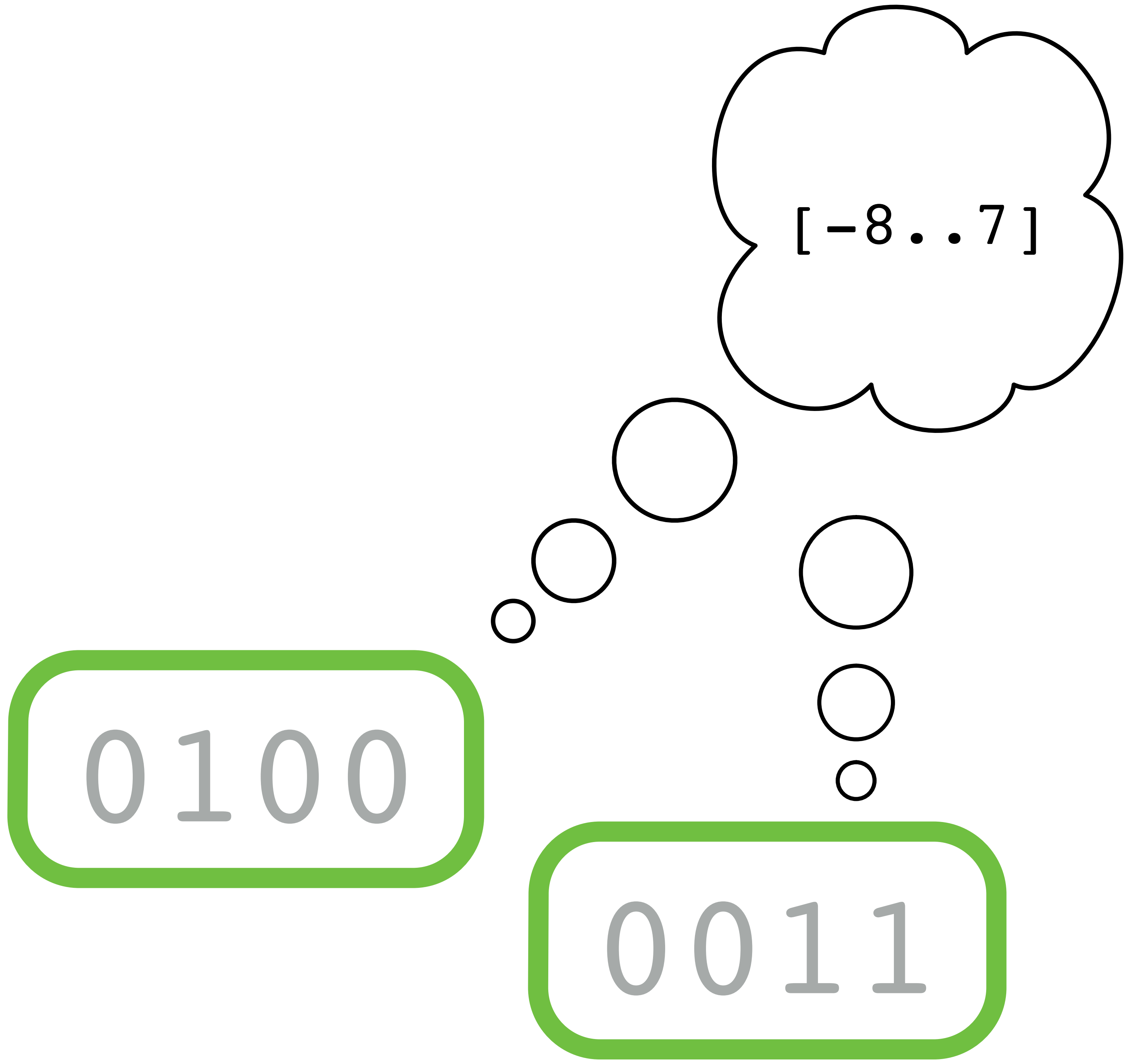
0100

0100



0100

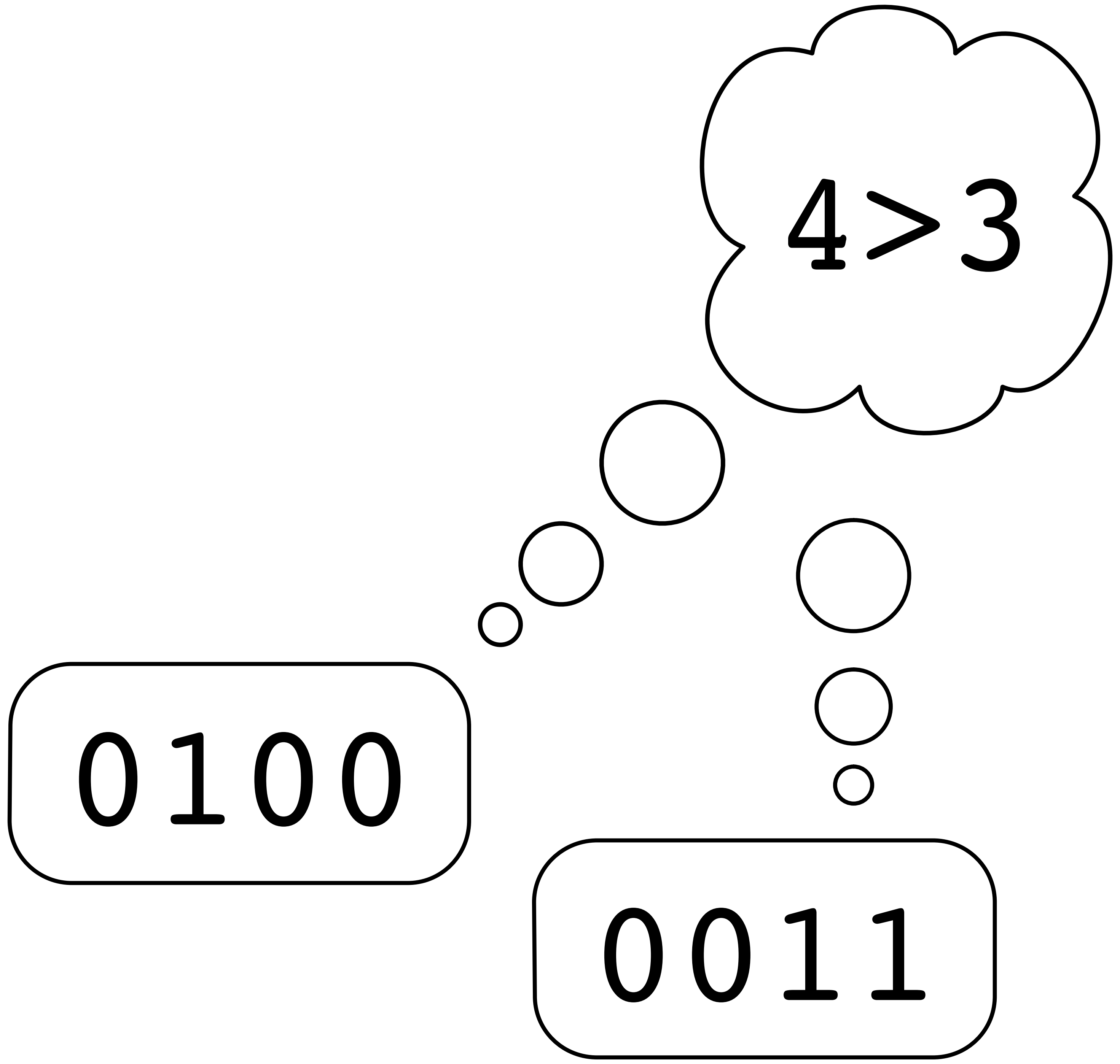
[-8..7]



[-8..7]

0100

0011



4>3

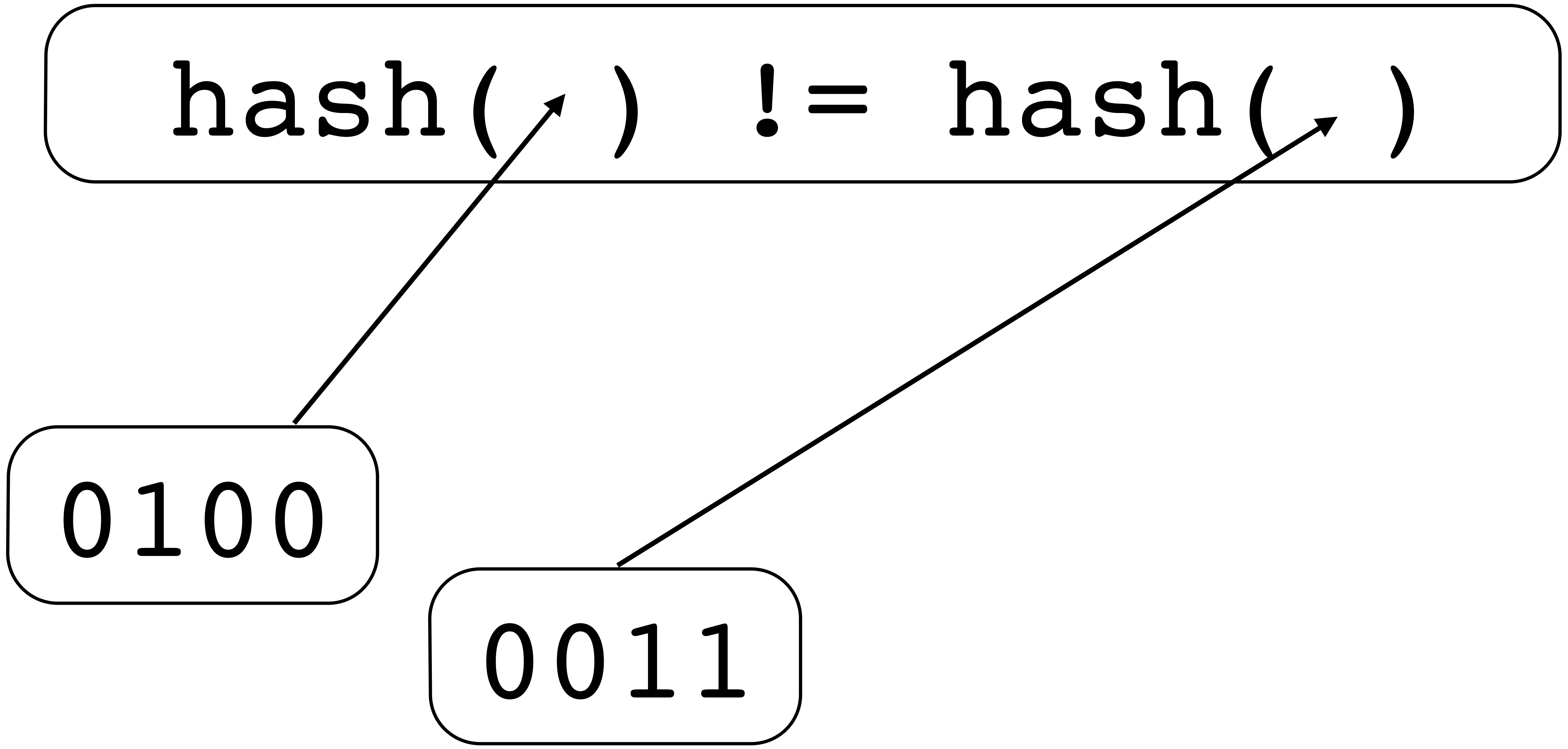
0100

0011

`hash( ) != hash( )`

0100

0011



`hash( ) != hash( )`

0100

0011

Memory Space

110000101111011111110111101100110011001110

001100101001111010011011010100100001001000

110000101000110011000000111001010011010100

100110010001101110011100001010011111011101

0001110011110000011 **0100** 110010001011111100

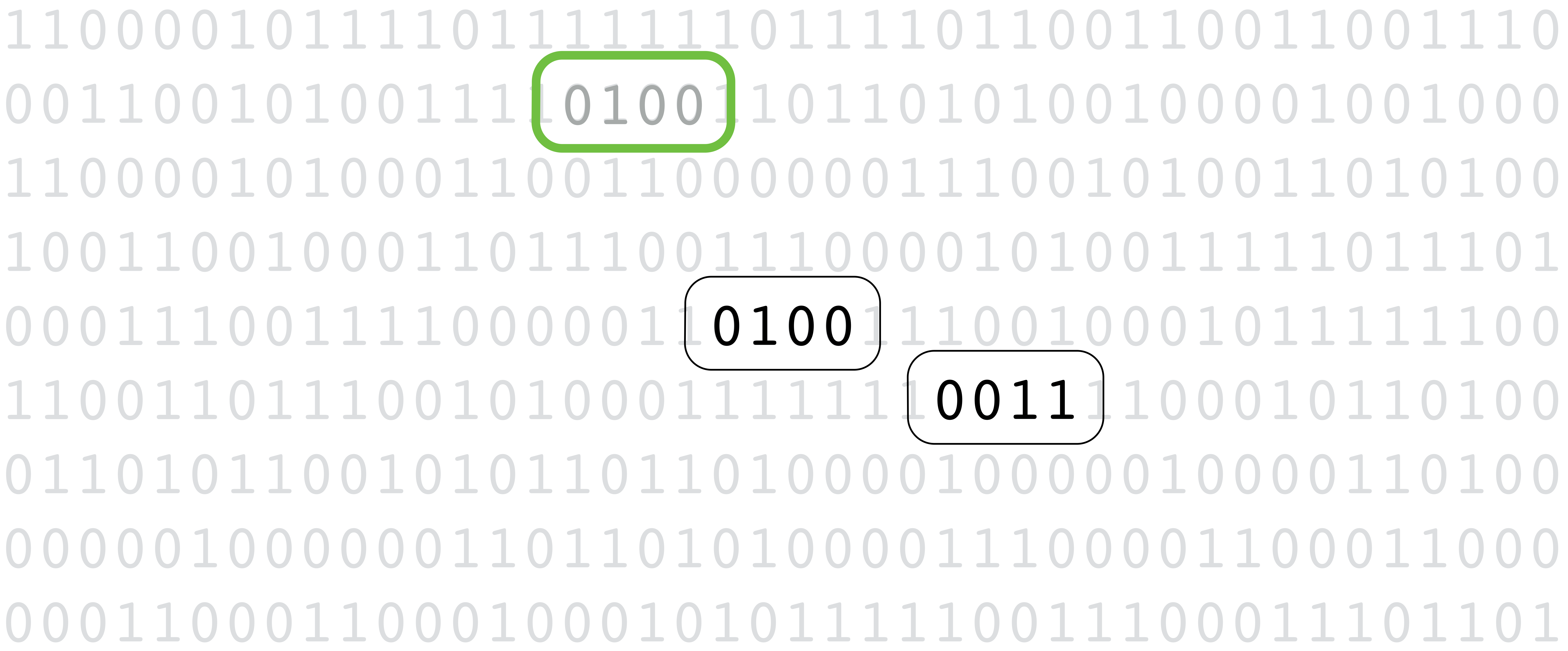
110011011100101000111111 **0011** 100010110100

011010110010101101101000010000010000110100

000001000000110110101000011100001100011000

000110001100010001010111110011100011101101

Memory Space



110000101111011111110111101100110011001110

001100101001111010011011010100100001001000

110000101000110011000000111001010011010100

100110010001101110011100001010011111011101

000111001111000001101001110010001011111100

110011011100101000111111100111100010110100

011010110010101101101000010000010000110100

000001000000110110101000011100001100011000

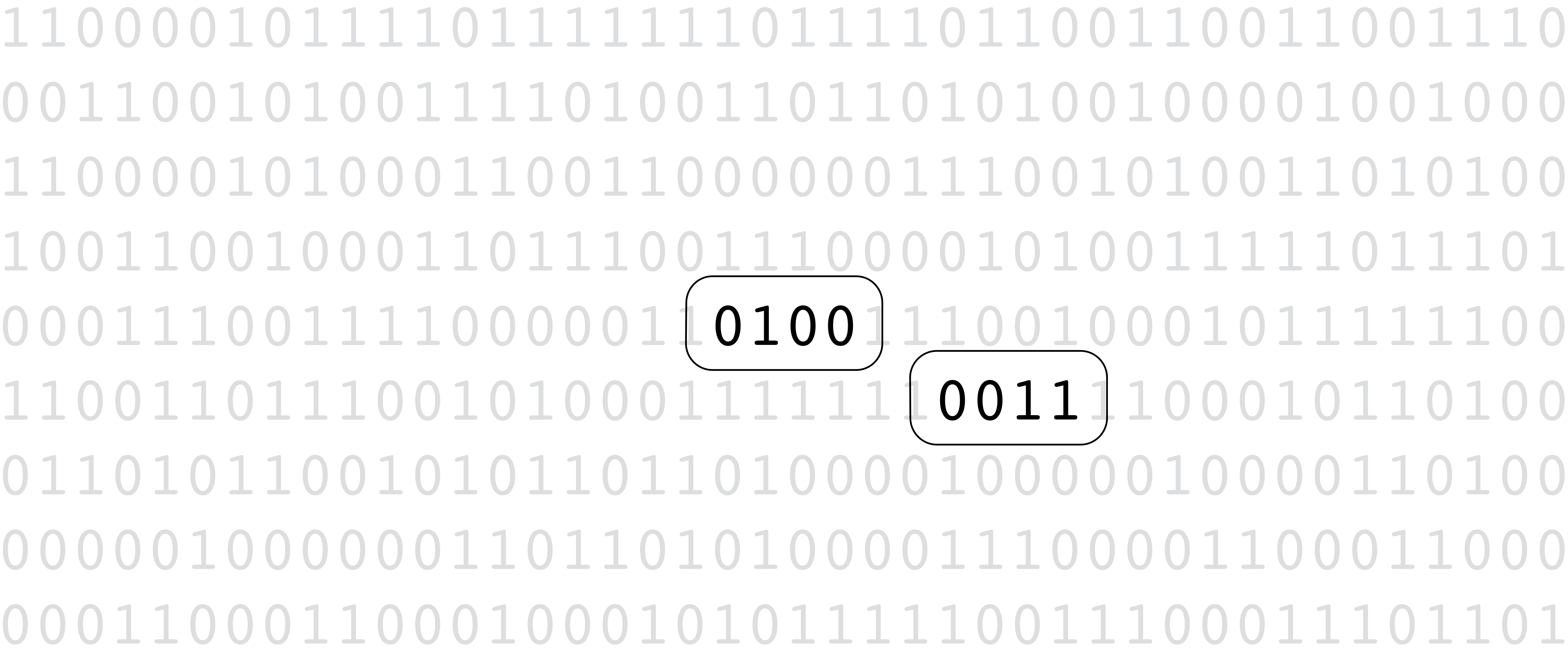
000110001100010001010111110011100011101101

0100

0100

0011

Memory Space



110000101111011111110111101100110011001110

001100101001111010011011010100100001001000

110000101000110011000000111001010011010100

100110010001101110011100001010011111011101

0001110011110000011**0100**1110010001011111100

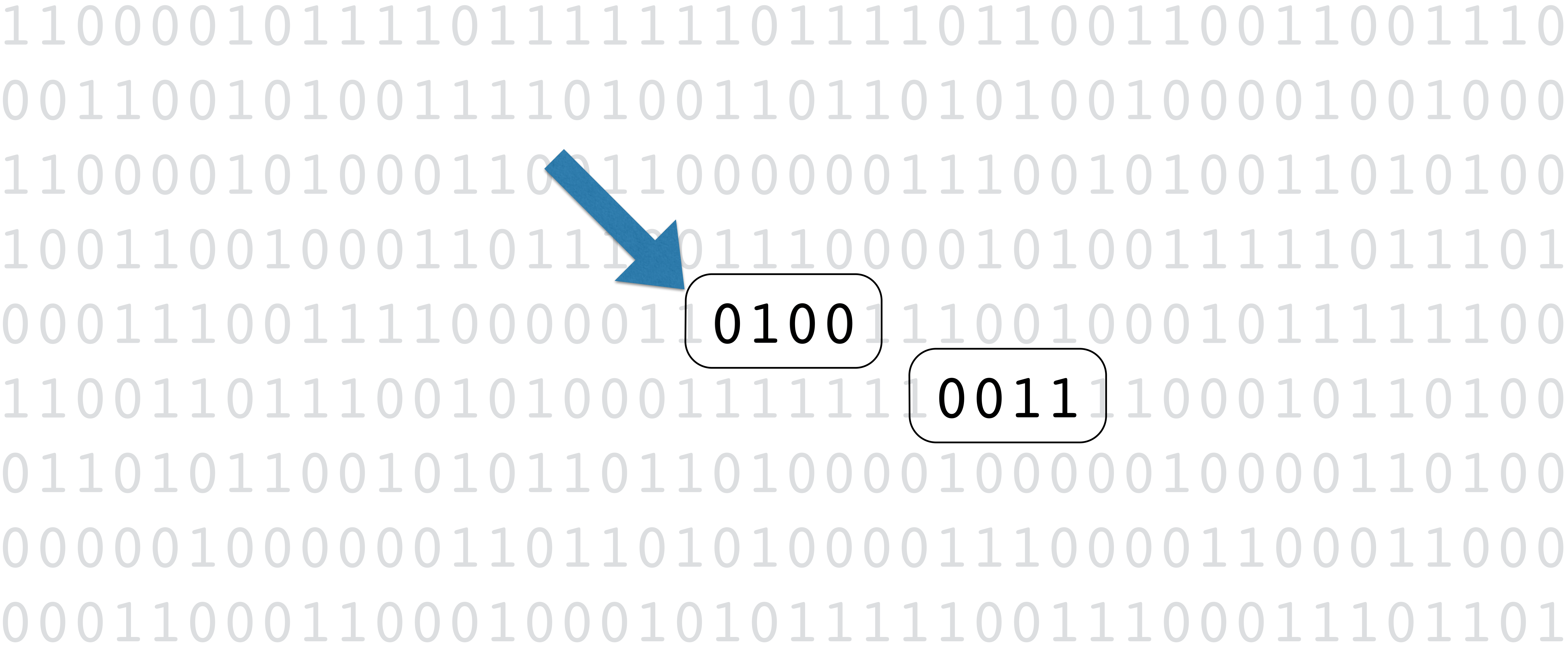
1100110111001010001111111**0011**1100010110100

011010110010101101101000010000010000110100

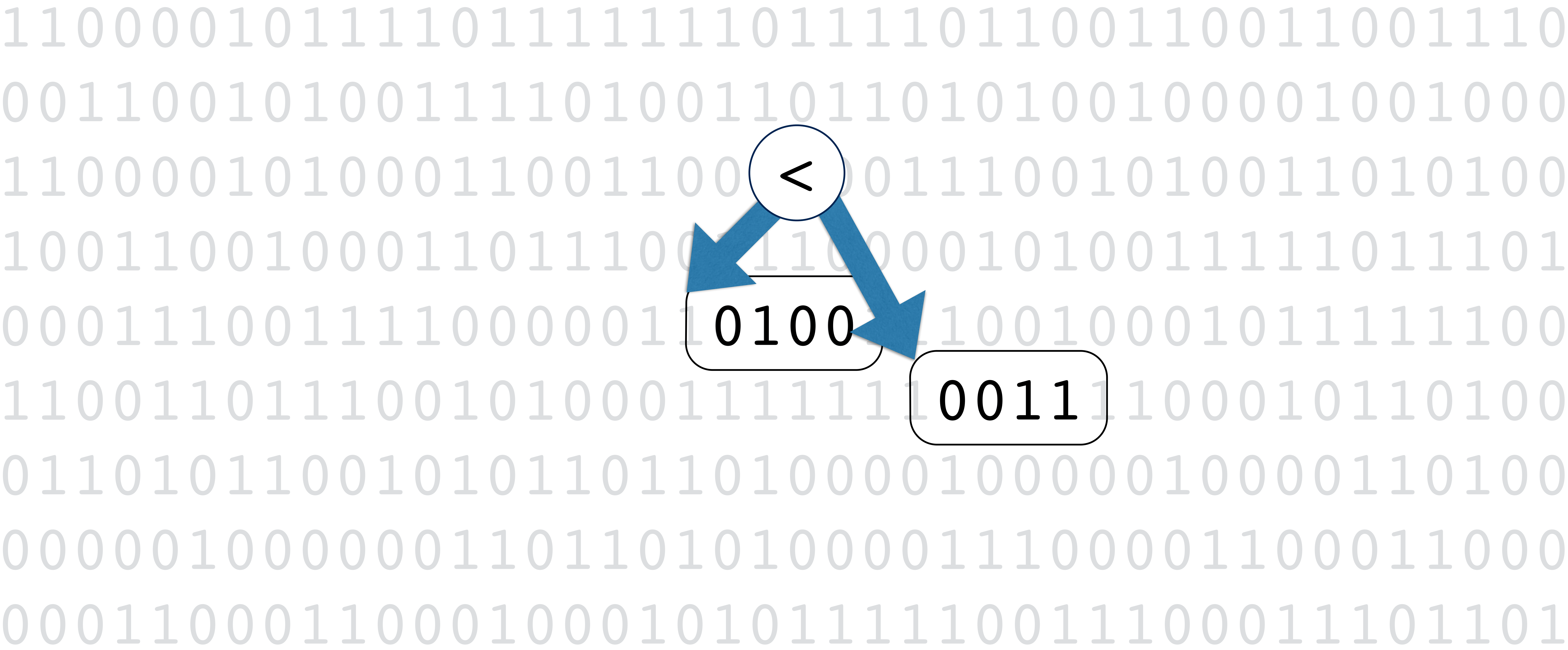
000001000000110110101000011100001100011000

000110001100010001010111110011100011101101

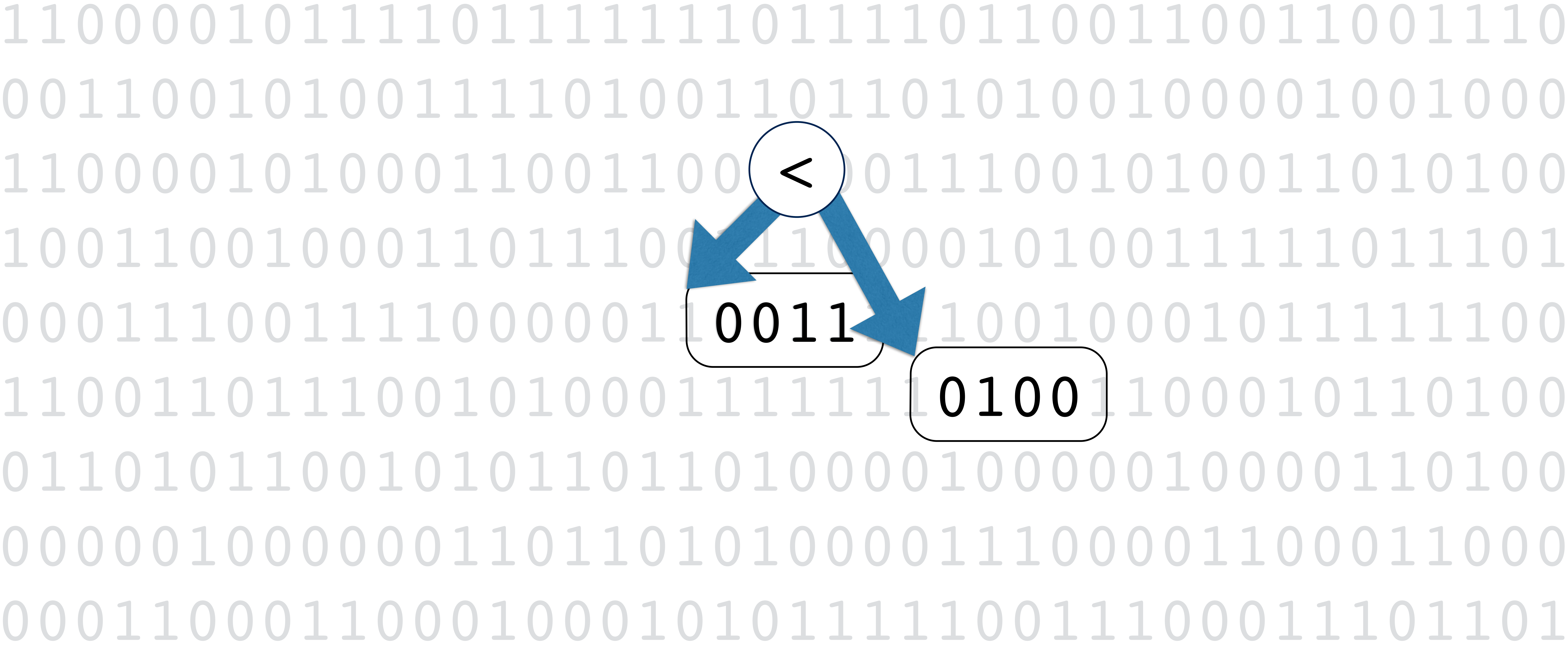
Memory Space



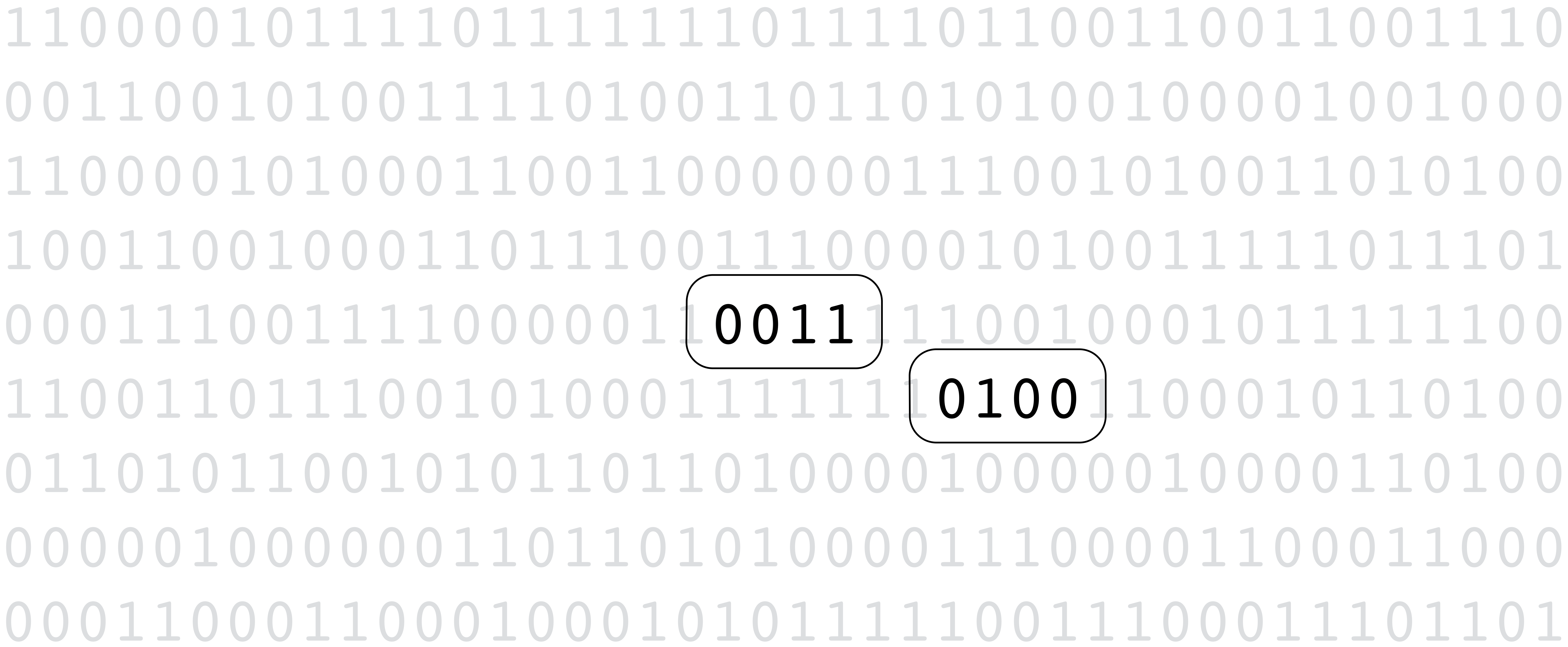
Memory Space



Memory Space



Memory Space



110000101111011111110111101100110011001110

001100101001111010011011010100100001001000

110000101000110011000000111001010011010100

100110010001101110011100001010011111011101

00011100111100000011 0011 1110010001011111100

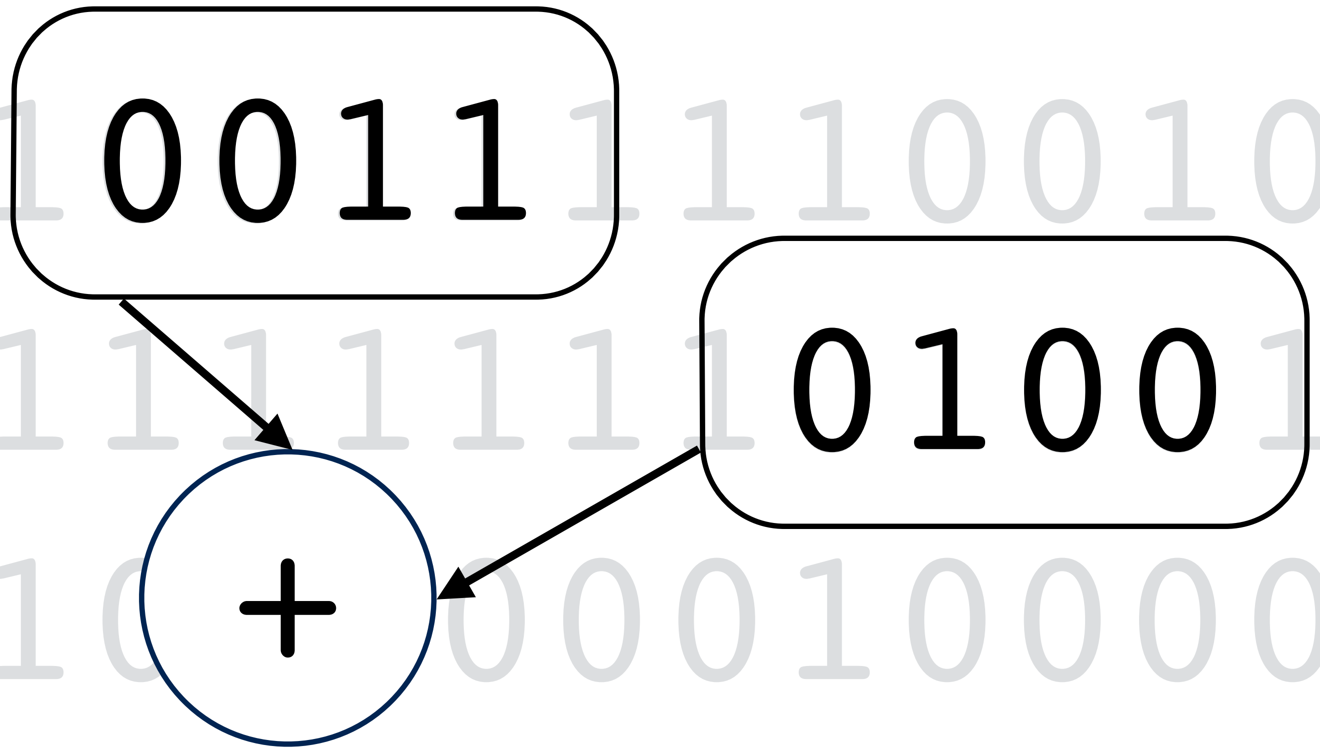
110011011100101000111111 0100 1100010110100

011010110010101101101000010000010000110100

000001000000110110101000011100001100011000

000110001100010001010111110011100011101101

Memory Space



110000101111011111110111101100110011001110

001100101001111010011011010100100001001000

110000101000110011000000111001010011010100

100110010001101110011100001010011111011101

000111001111000001100111110010001011111100

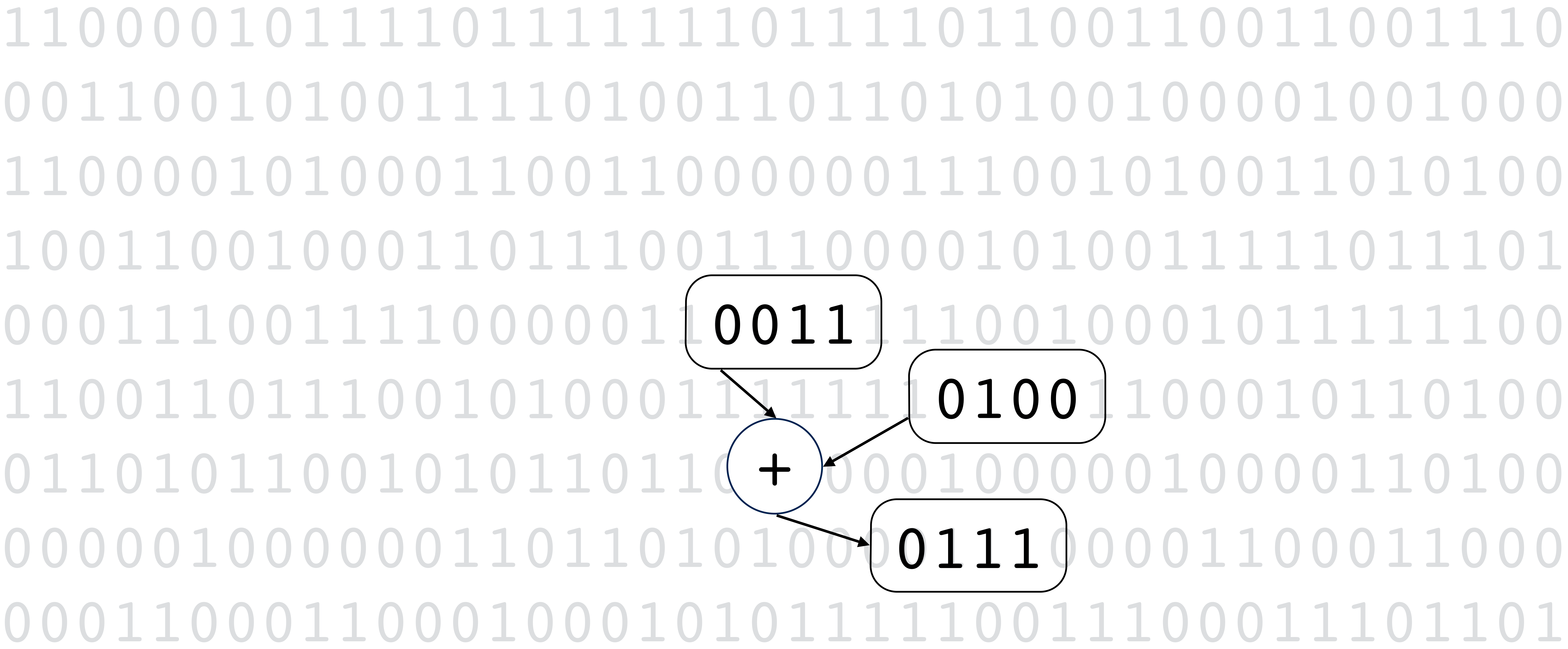
110011011100101000111111101001100010110100

01101011001010110110 + 00010000010000110100

000001000000110110101000011100001100011000

000110001100010001010111110011100011101101

Memory Space

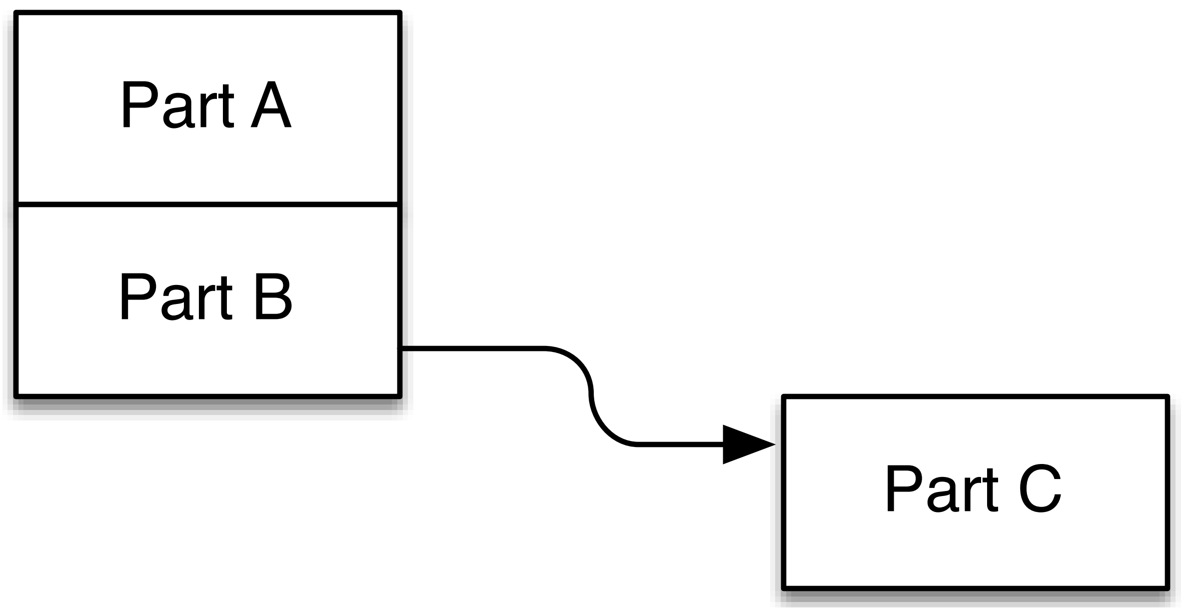


Whole-Part Relationships and Composite Objects

Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

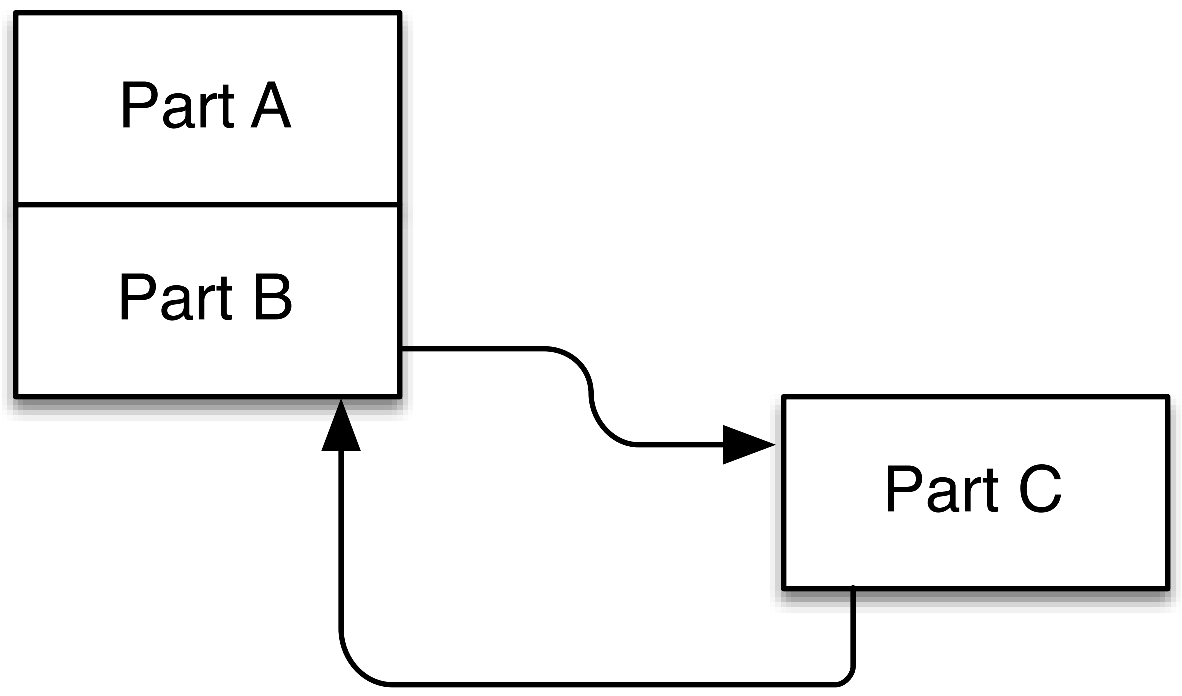
- Connected



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

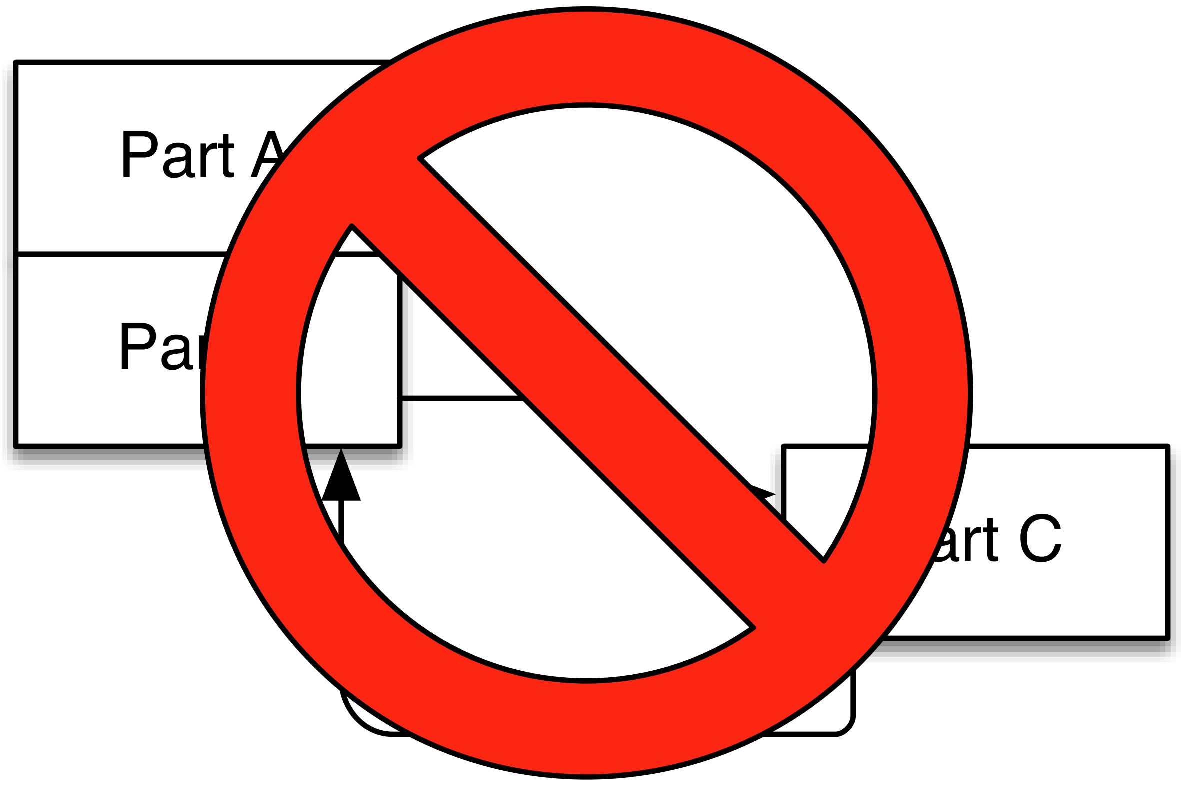
- Connected
- Noncircular



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

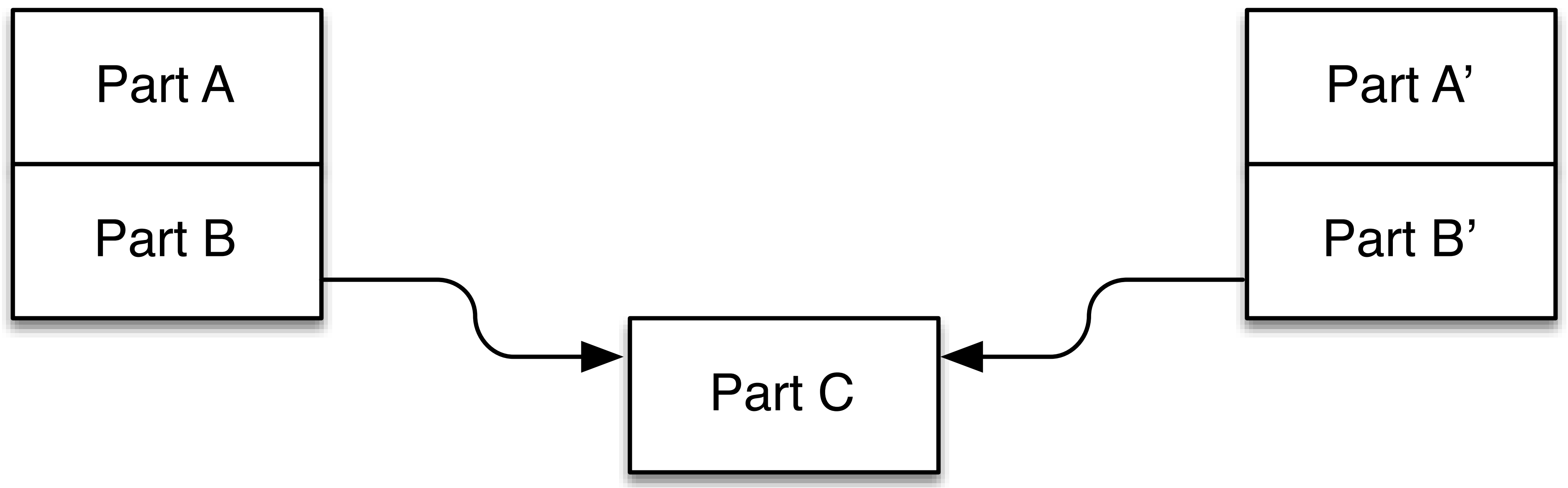
- Connected
- Noncircular



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

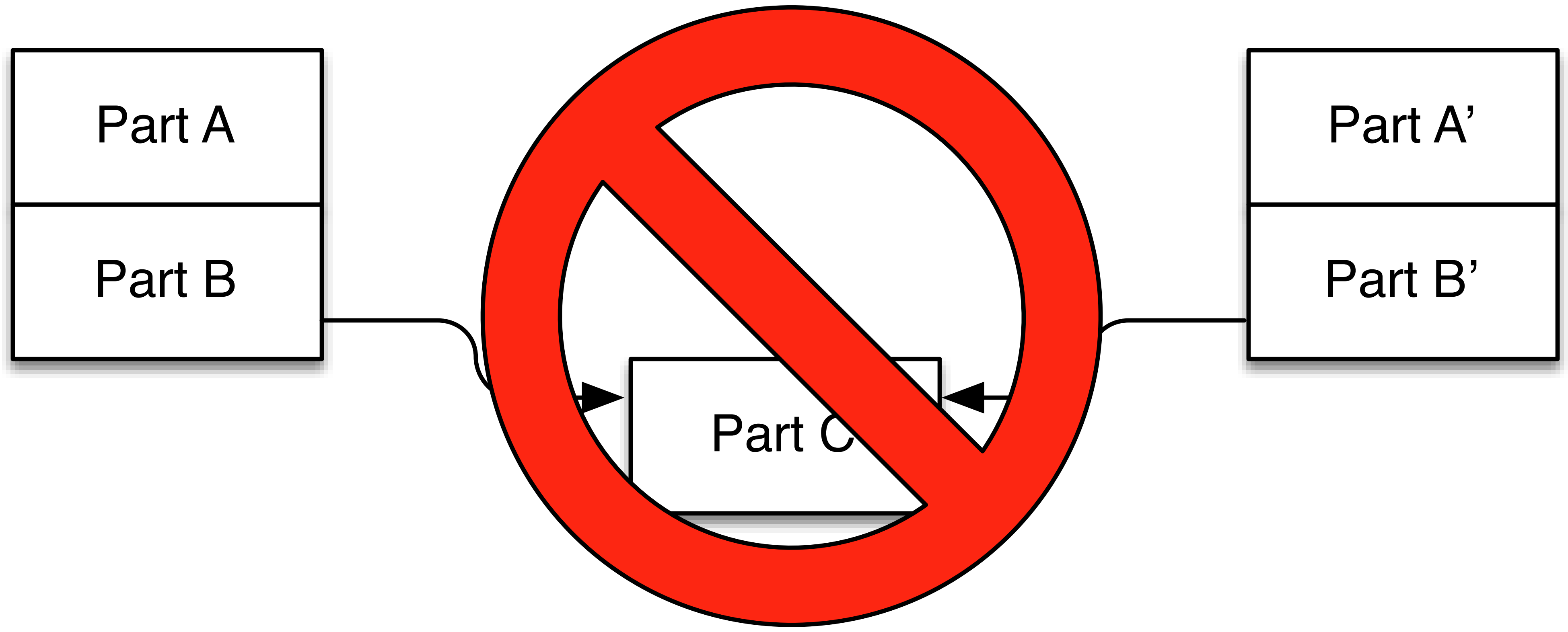
- Connected
- Noncircular
- Logically Disjoint



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

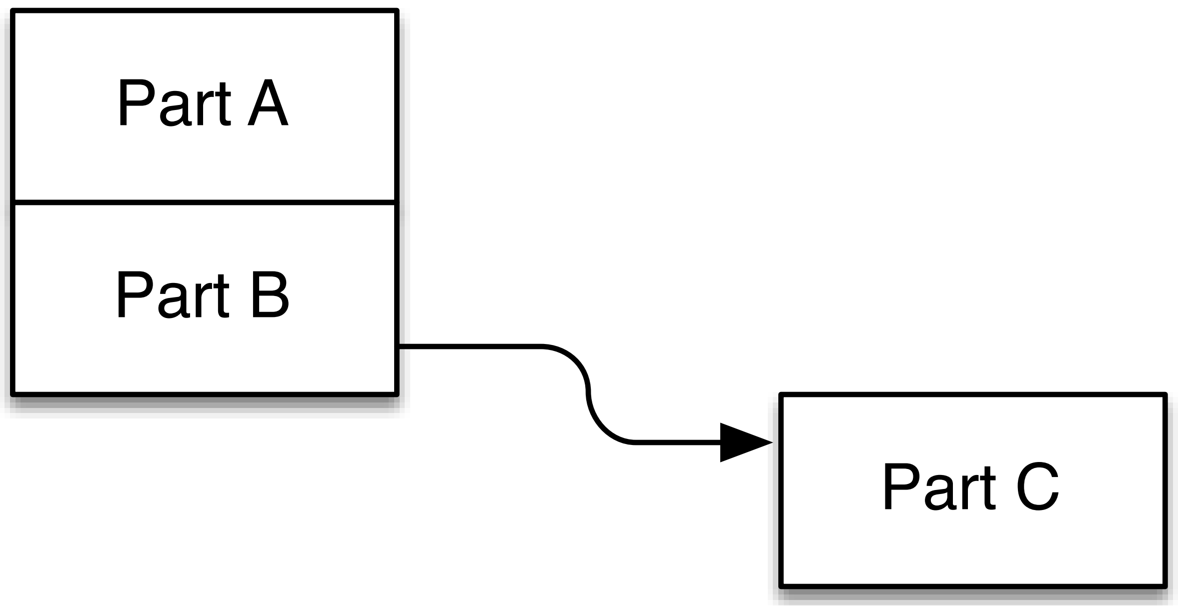
- Connected
- Noncircular
- Logically Disjoint



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

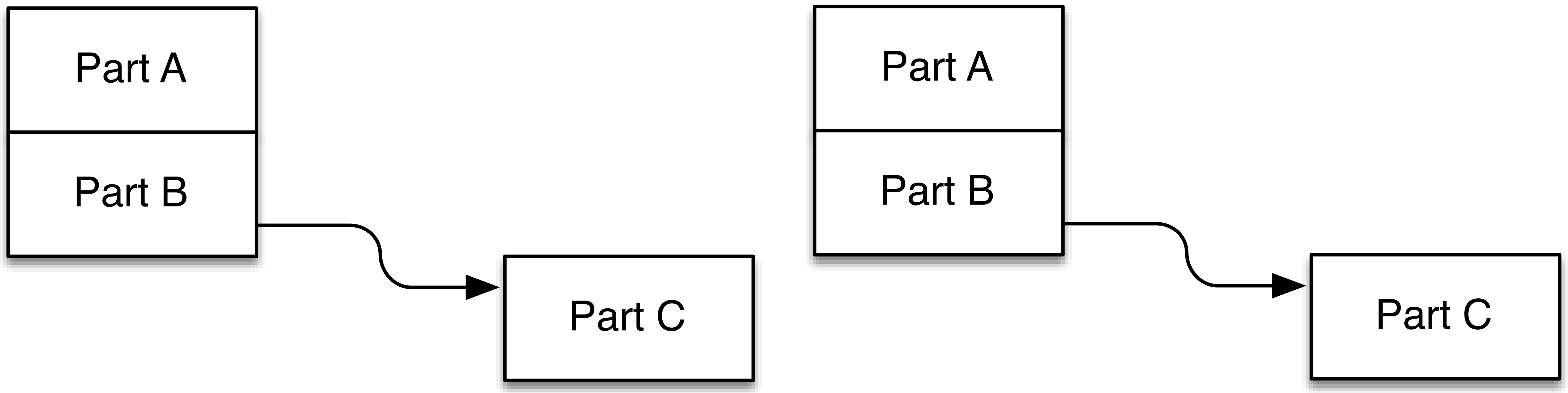
- Connected
- Noncircular
- Logically Disjoint
- Owning



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

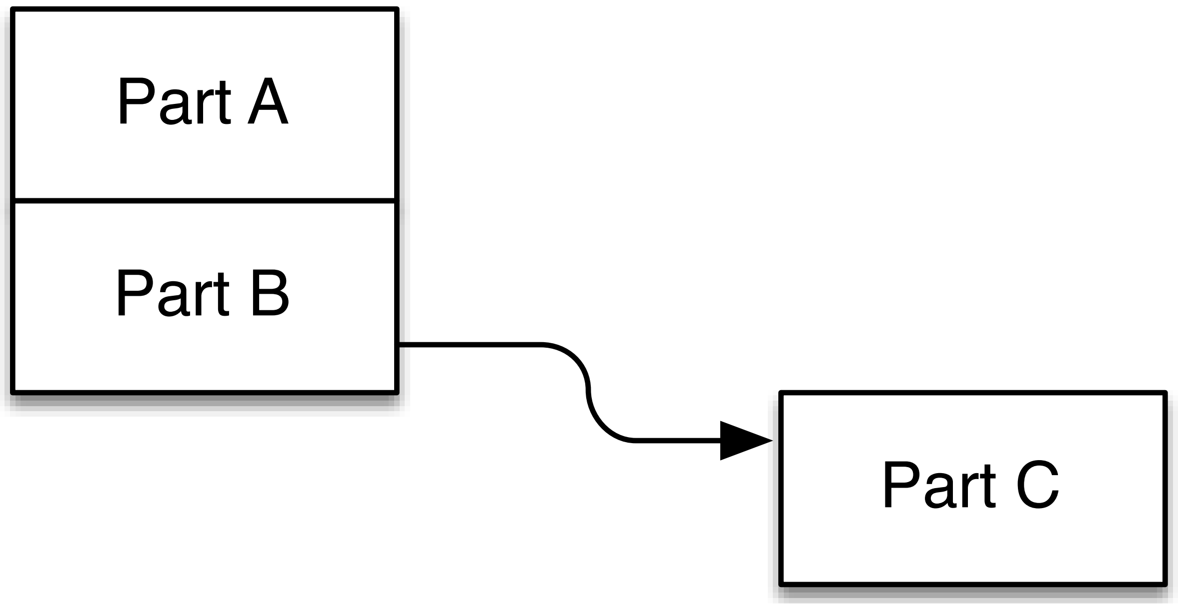
- Connected
- Noncircular
- Logically Disjoint
- Owning



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint
- Owning

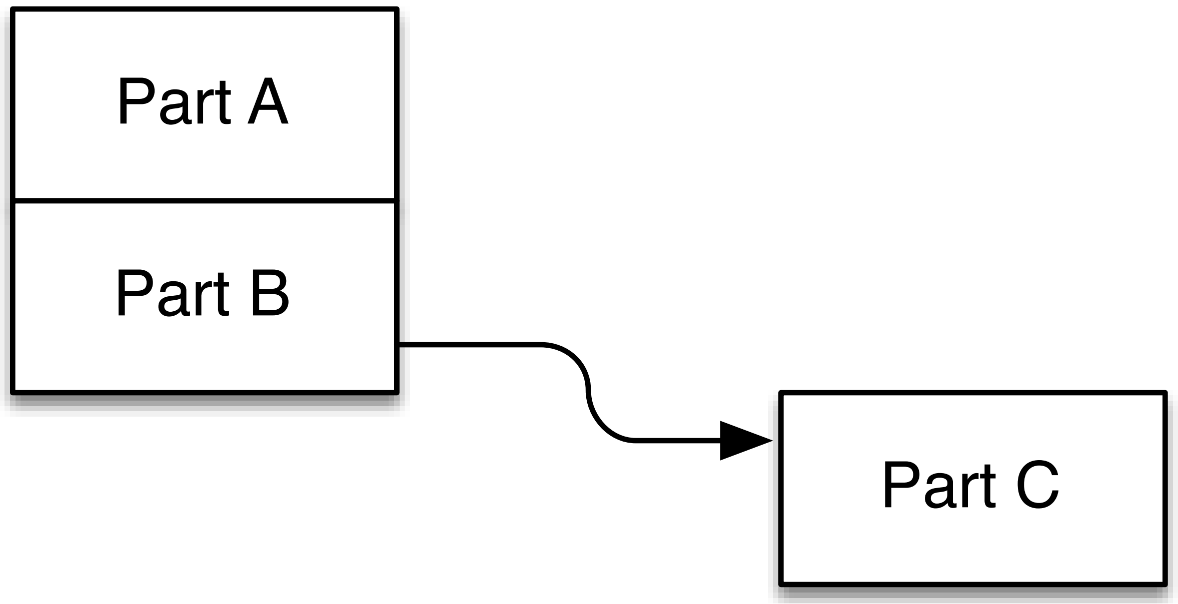


Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint
- Owning

- Standard Containers are Composite Objects



Elements of Programming, Chapter 12

What is a data structure?

What is a data structure?

Definition: A structure utilizing value, physical, and representational relationships to encode semantic relationships on a collection of objects.

What is a data structure?

Definition: A structure utilizing value, physical, and representational relationships to encode semantic relationships on a collection of objects.

The choice of encoding can make a dramatic difference on the performance of operations.

Data Structure Performance

3GHz processor, from Chandler Carruth talk - Credit to Jeff Dean

Data Structure Performance

- Hierarchical Memory Structure

Register Access	0.1	ns
L1 Cache	0.5	ns
L2 Cache	7.0	ns
Memory	100.0	ns

3GHz processor, from Chandler Carruth talk - Credit to Jeff Dean

Data Structure Performance

- Hierarchical Memory Structure

Register Access	0.1	ns
L1 Cache	0.5	ns
L2 Cache	7.0	ns
Memory	100.0	ns

- RAM behaves much like a disk drive

3GHz processor, from Chandler Carruth talk - Credit to Jeff Dean

Data Structure Performance

- Hierarchical Memory Structure

Register Access	0.1	ns
L1 Cache	0.5	ns
L2 Cache	7.0	ns
Memory	100.0	ns

- RAM behaves much like a disk drive

$$\log_2 1{,}000{,}000{,}000{,}000 = 40$$

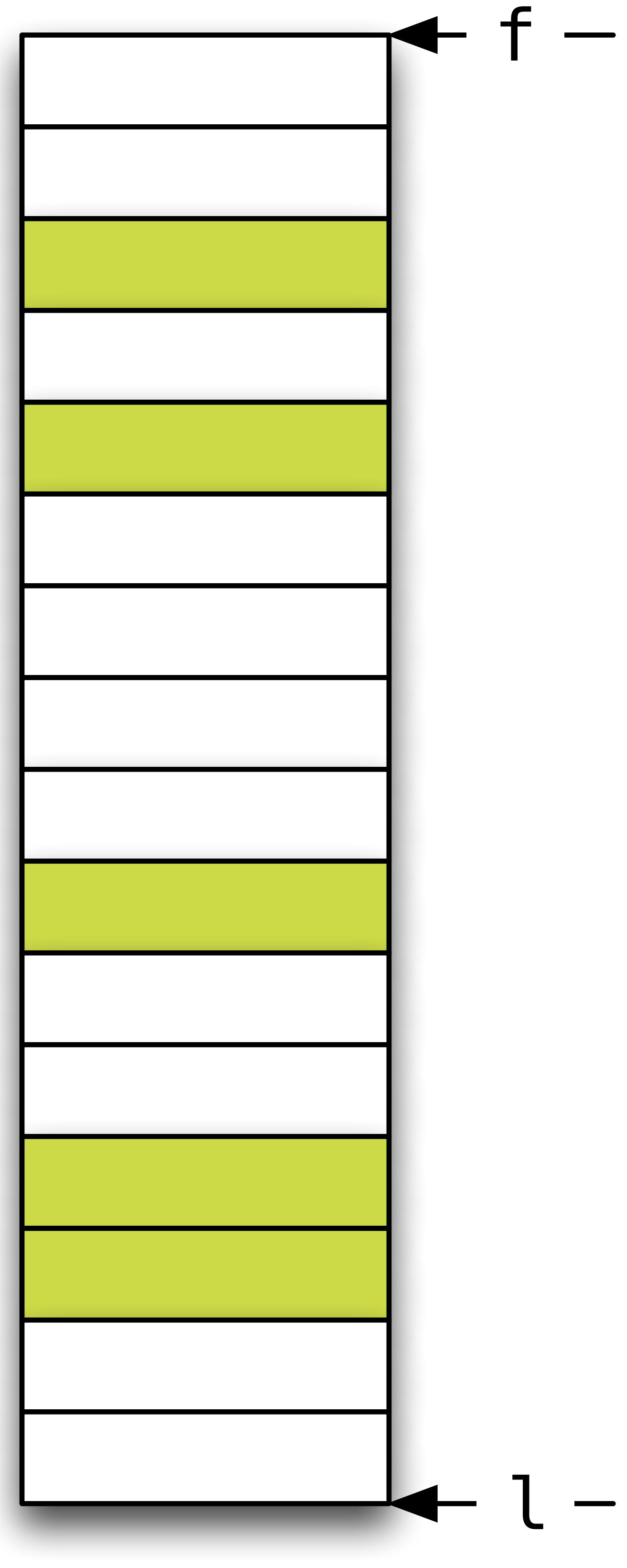
3GHz processor, from Chandler Carruth talk - Credit to Jeff Dean

Data Structure Performance

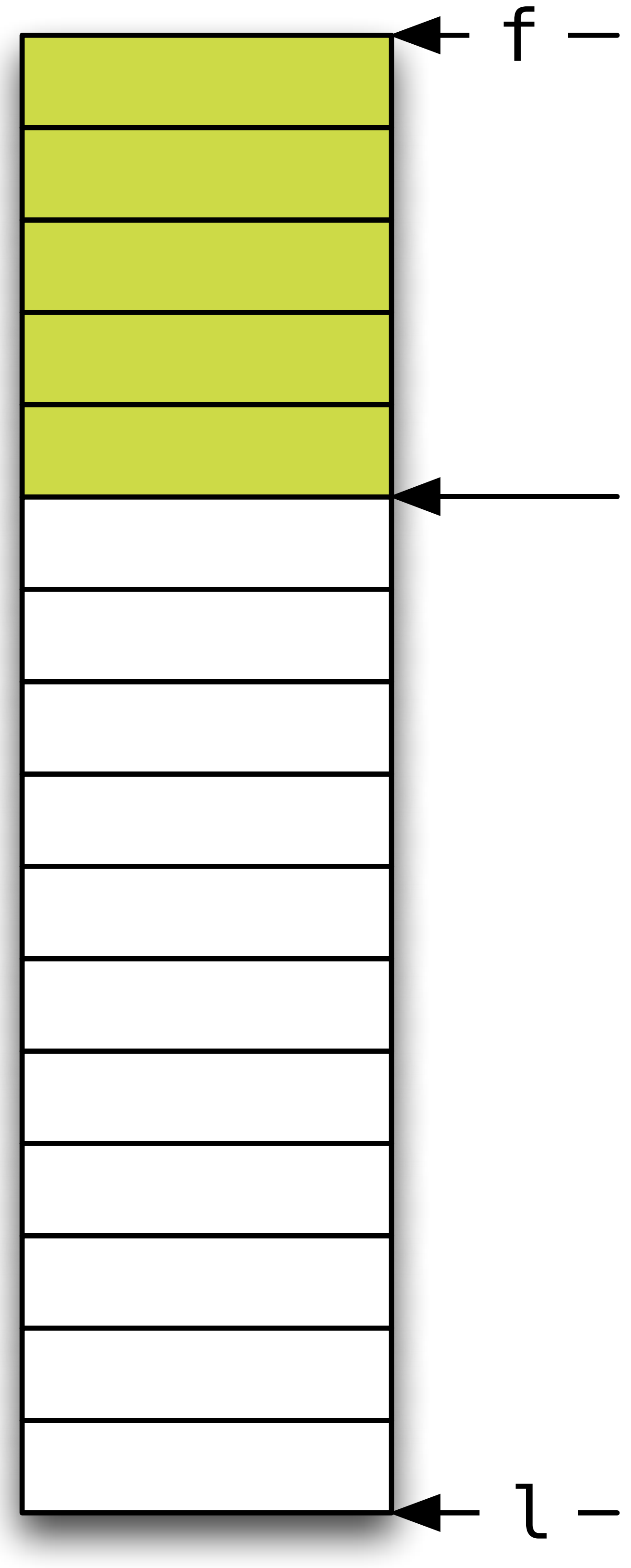
- Locality matters - use arrays or vector
  - Parallel Arrays
  - Static Lookup Tables
  - Closed Hash Maps
  - Algorithms

Example: Parallel Array & Algorithms

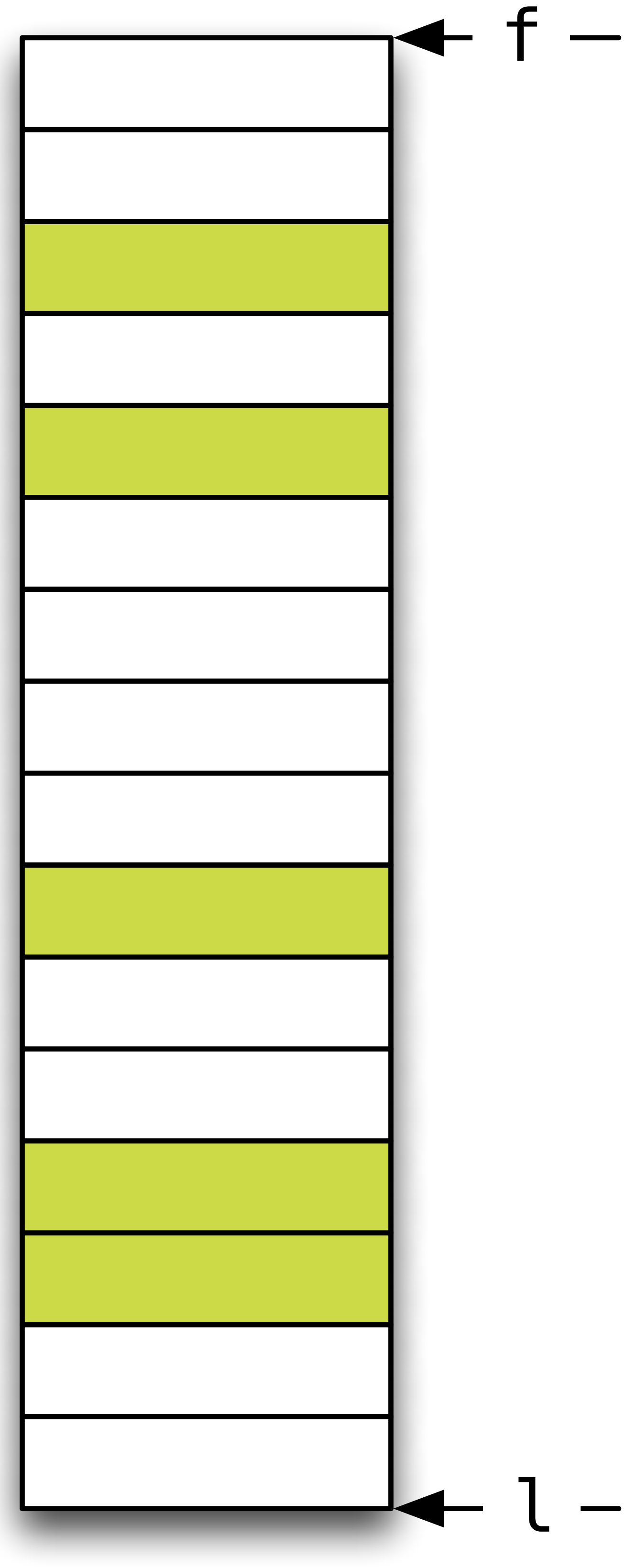
Stable Partition



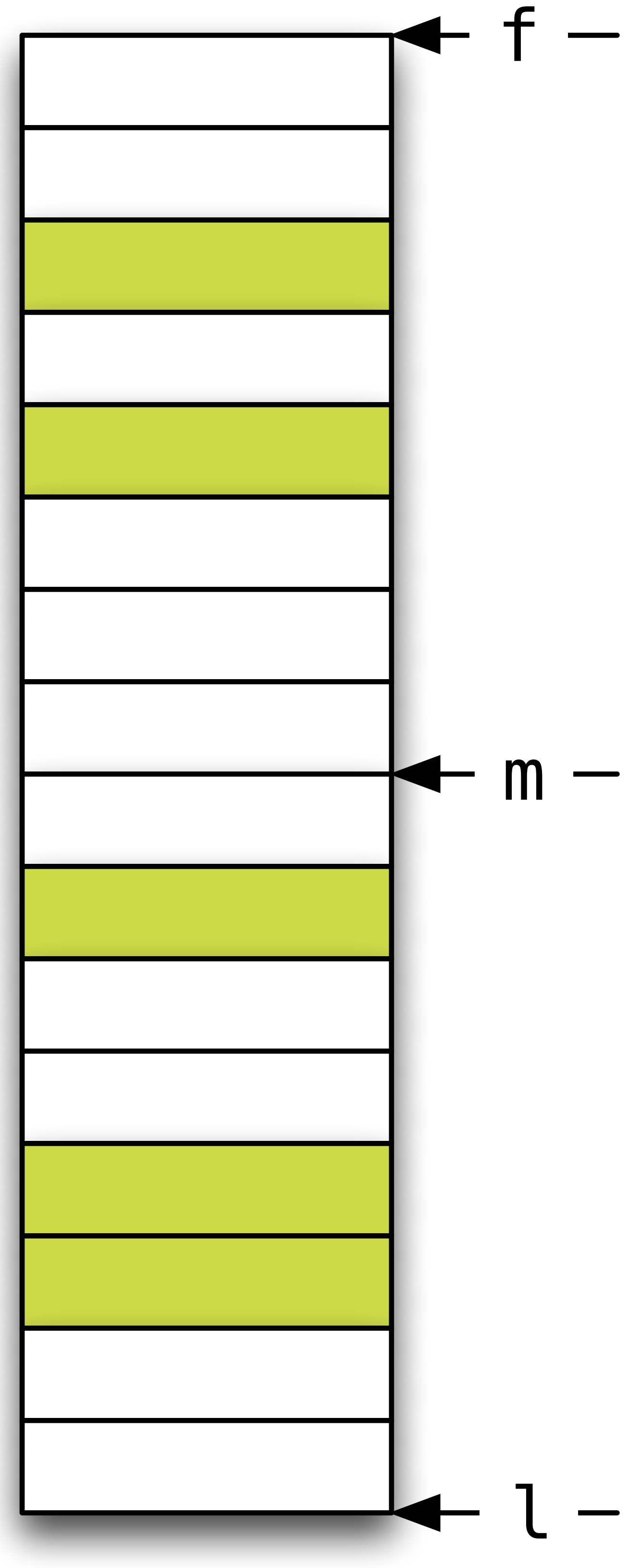
Stable Partition



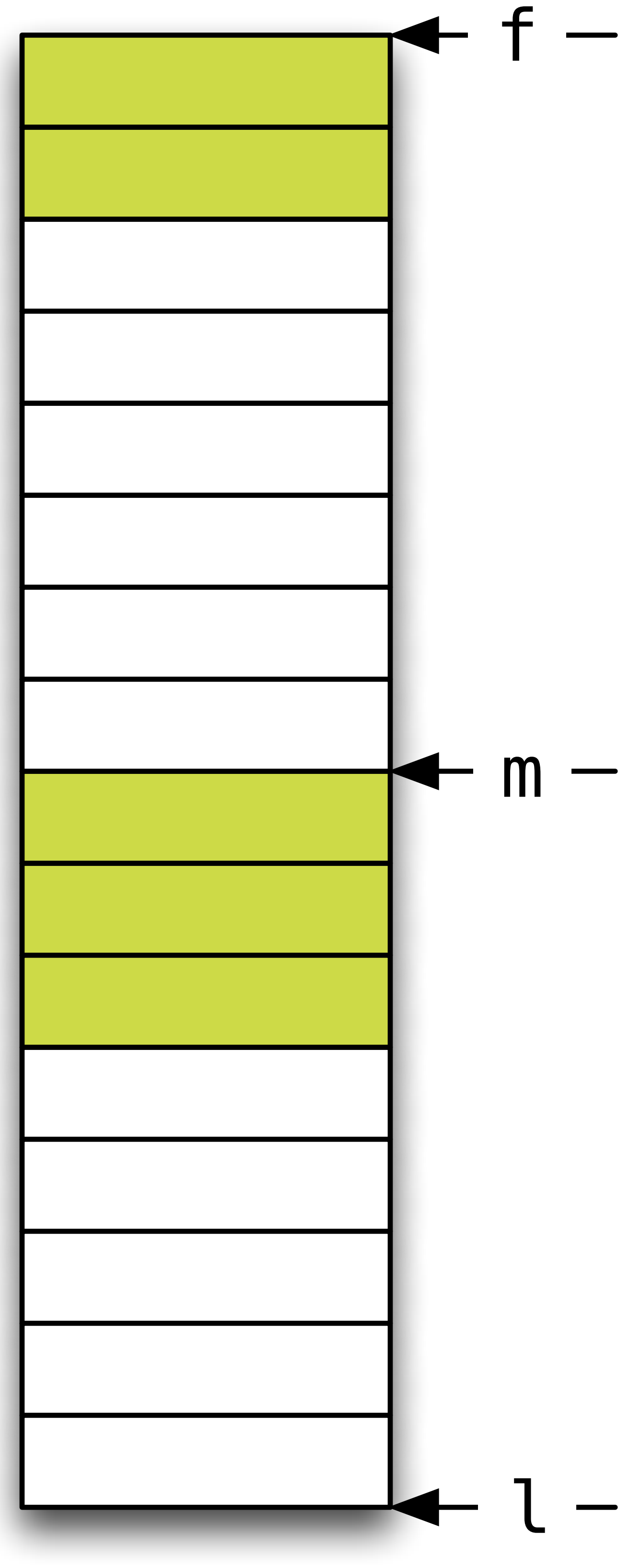
Stable Partition



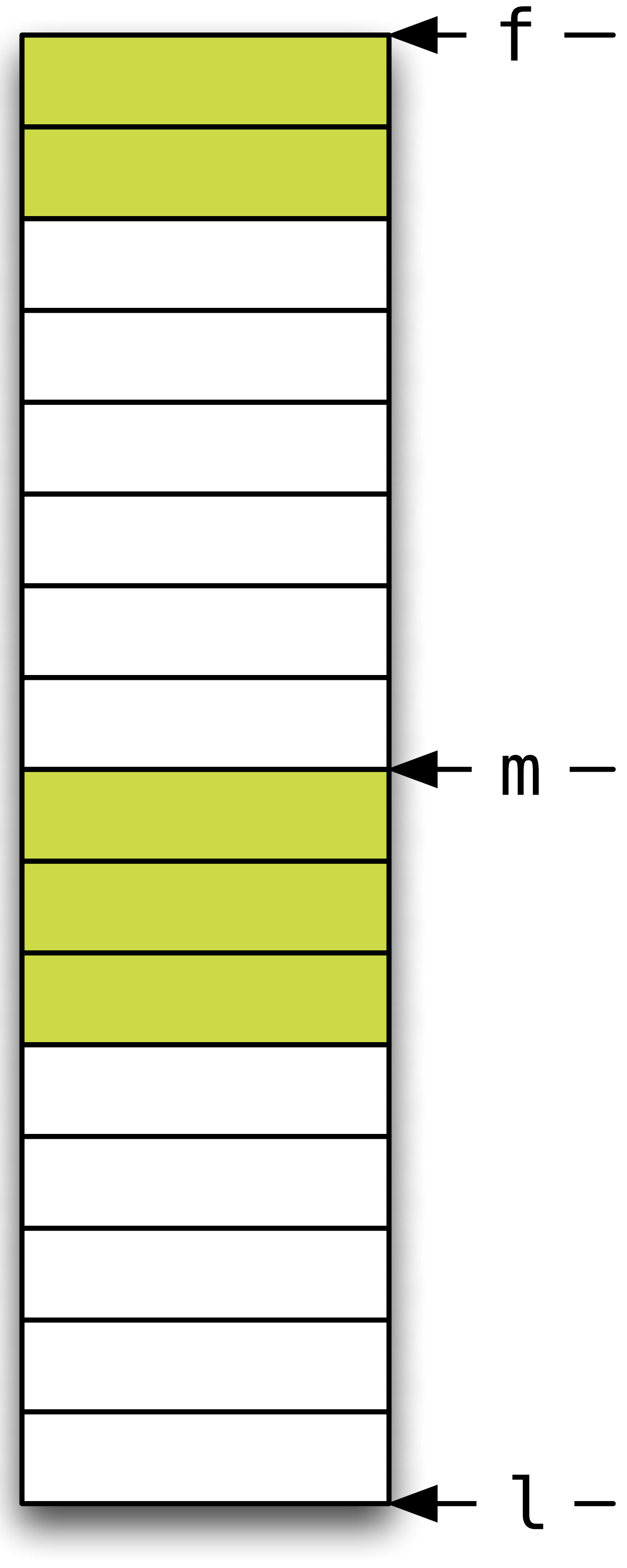
Stable Partition



Stable Partition

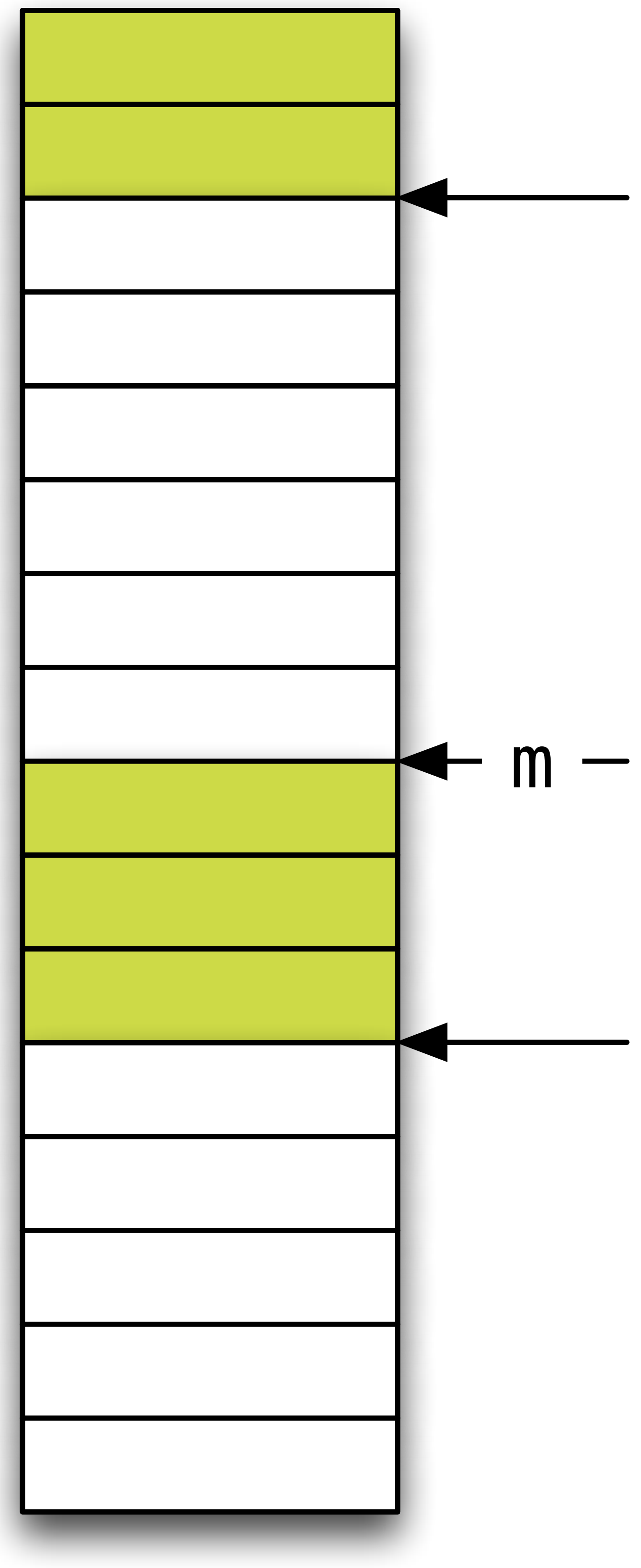


Stable Partition



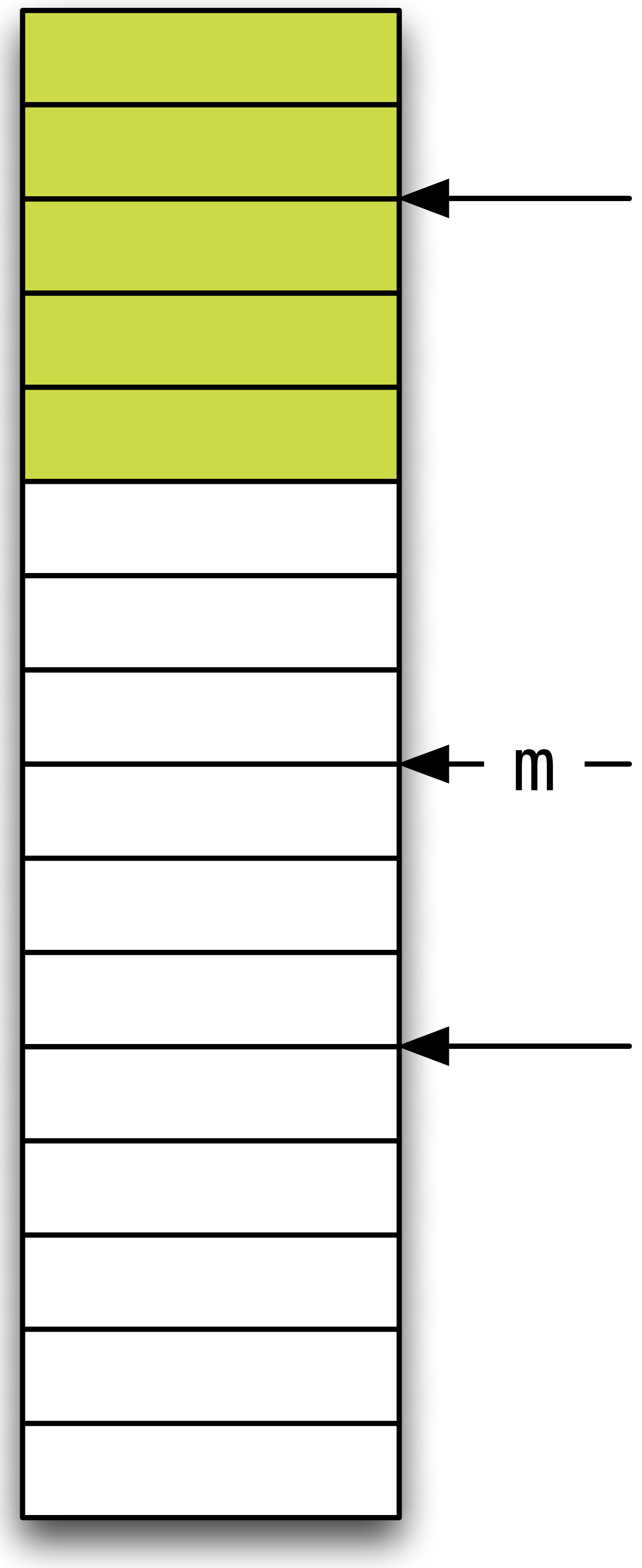
```
stable_partition(f, m, p)

stable_partition(m, l, p)
```

Stable Partition



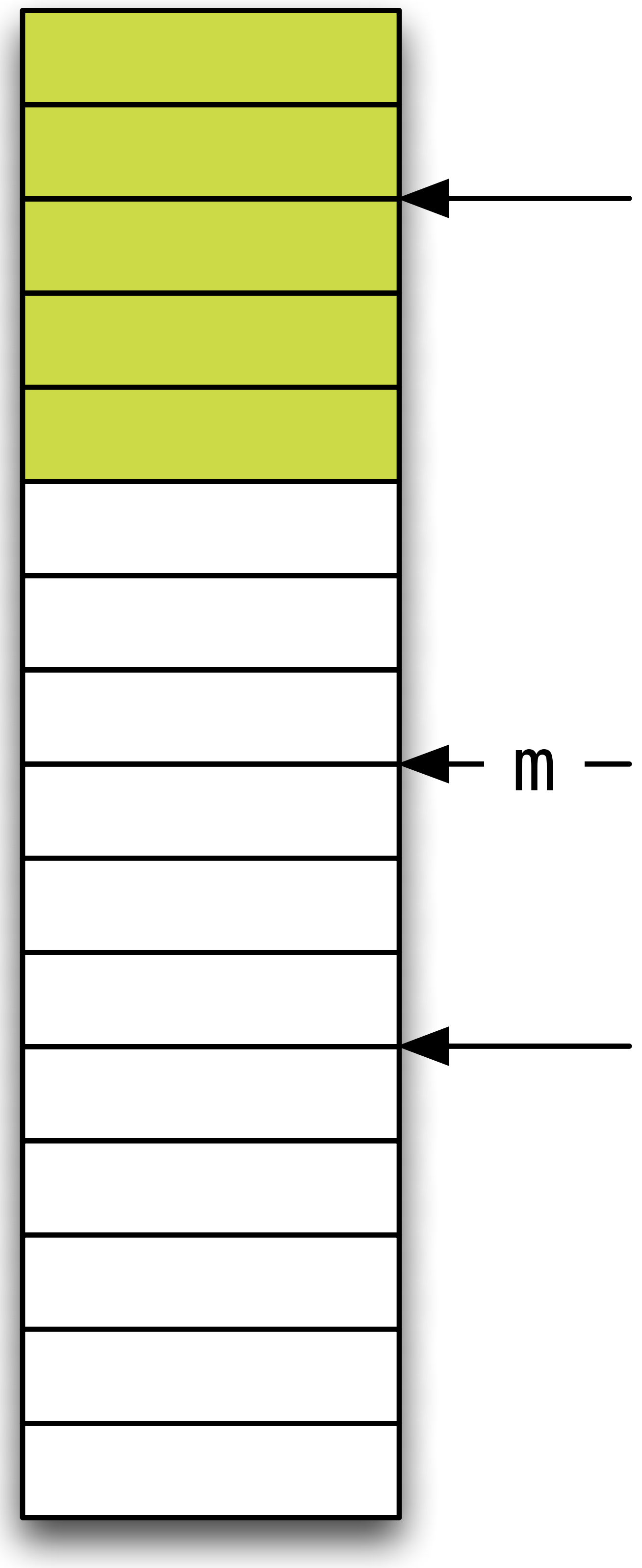
```
stable_partition(f, m, p)

stable_partition(m, l, p)
```

Stable Partition



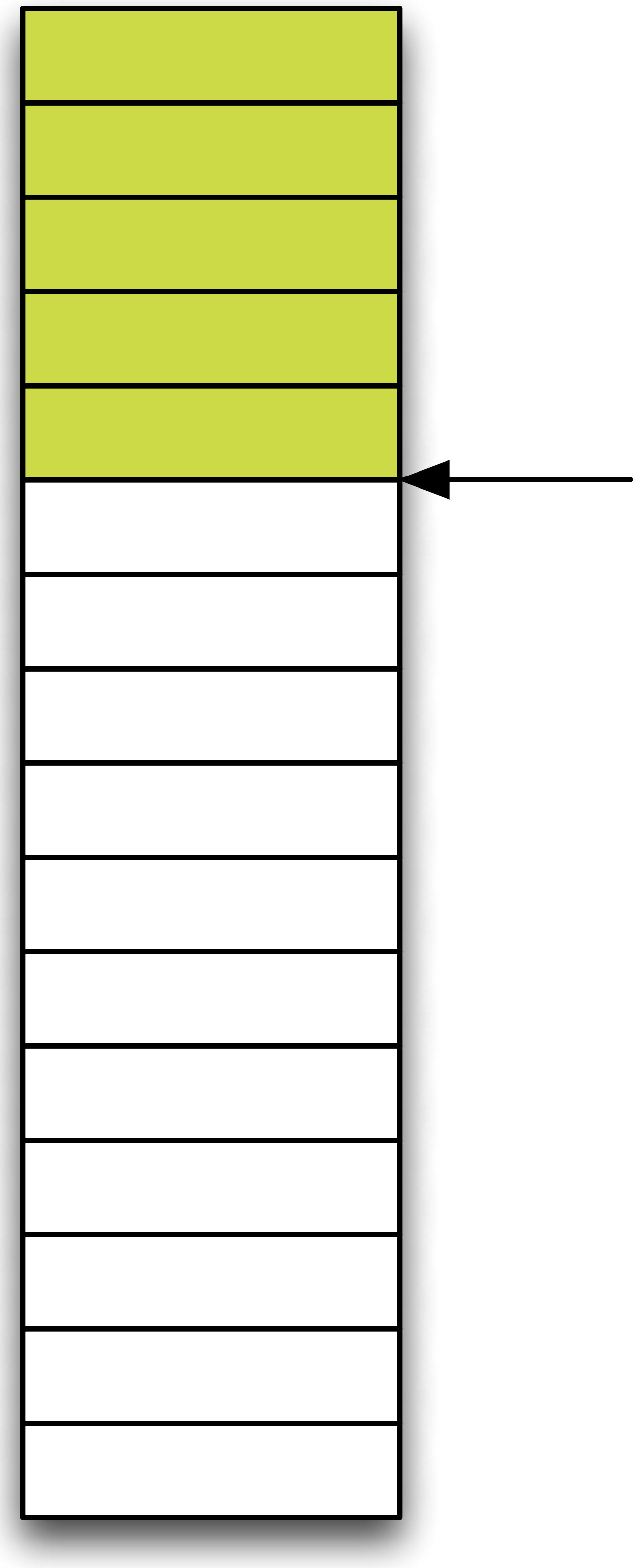
```
stable_partition(f, m, p)

stable_partition(m, l, p)
```

Stable Partition



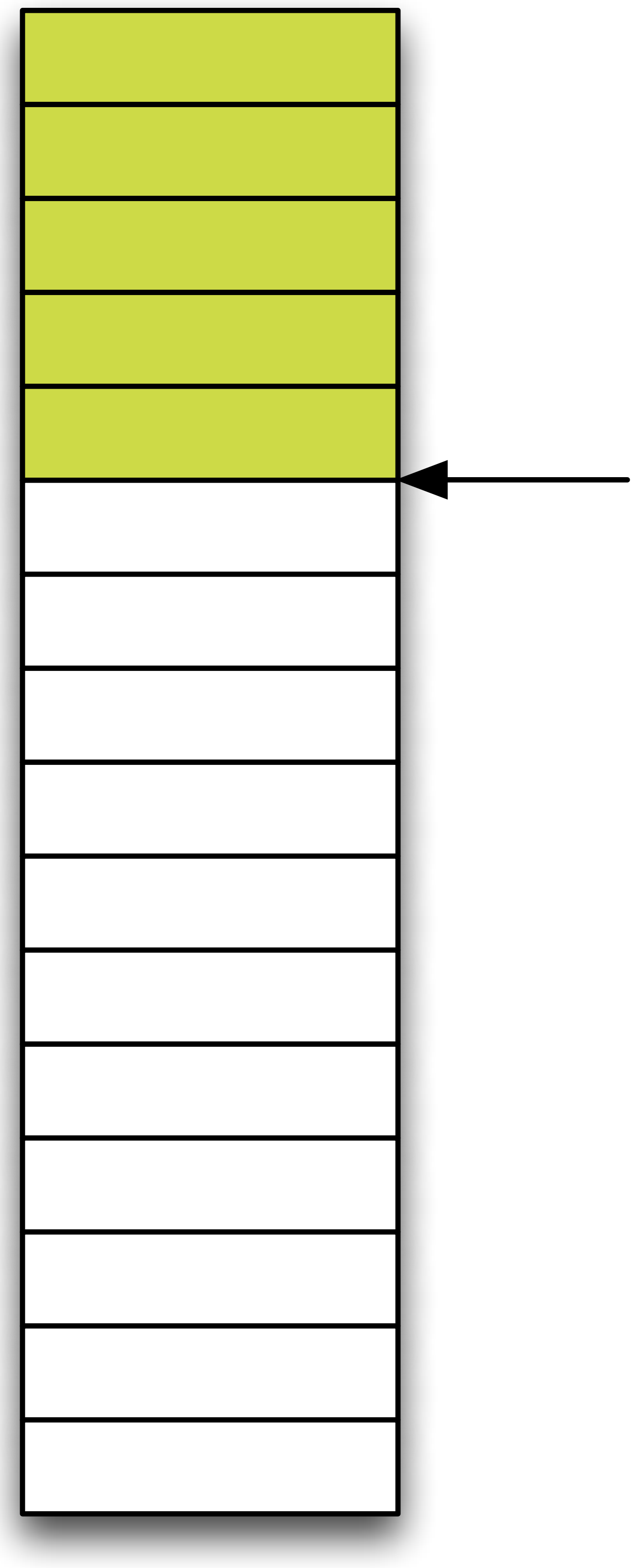
```
rotate(stable_partition(f, m, p),
       m,
       stable_partition(m, l, p));
```

Stable Partition



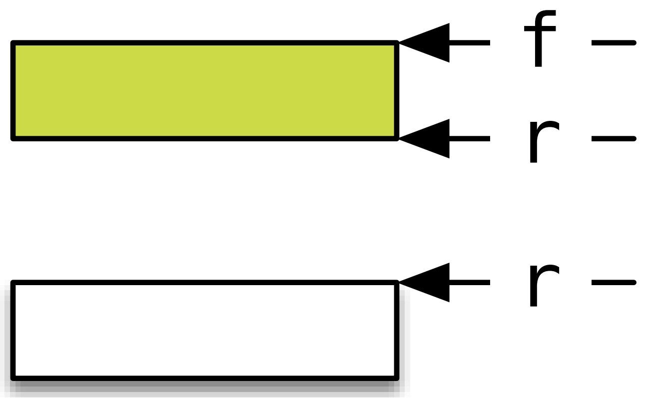
```
rotate(stable_partition(f, m, p),
       m,
       stable_partition(m, l, p));
```

Stable Partition



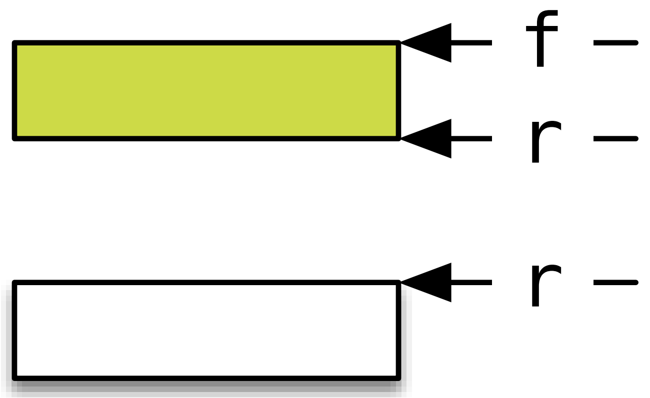
```
return rotate(stable_partition(f, m, p),
              m,
              stable_partition(m, l, p));
```

Stable Partition



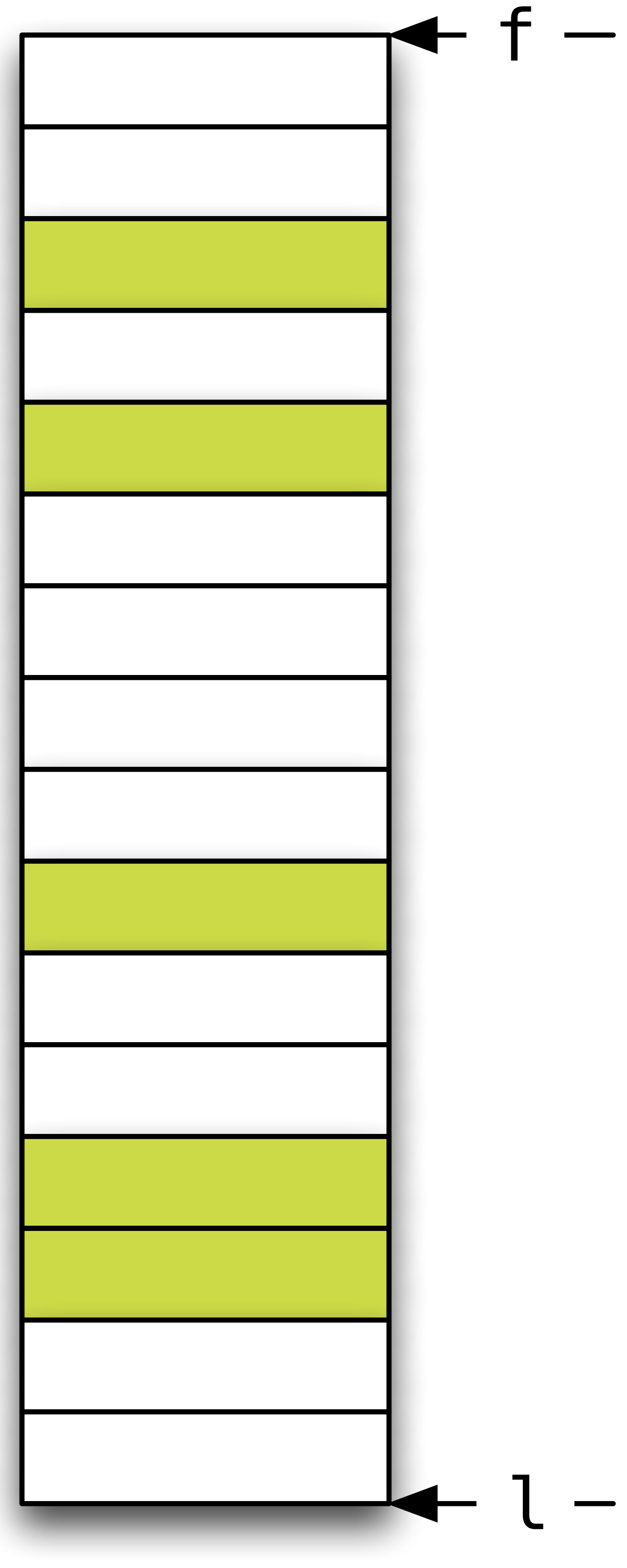
```
return rotate(stable_partition(f, m, p),
              m,
              stable_partition(m, l, p));
```

Stable Partition



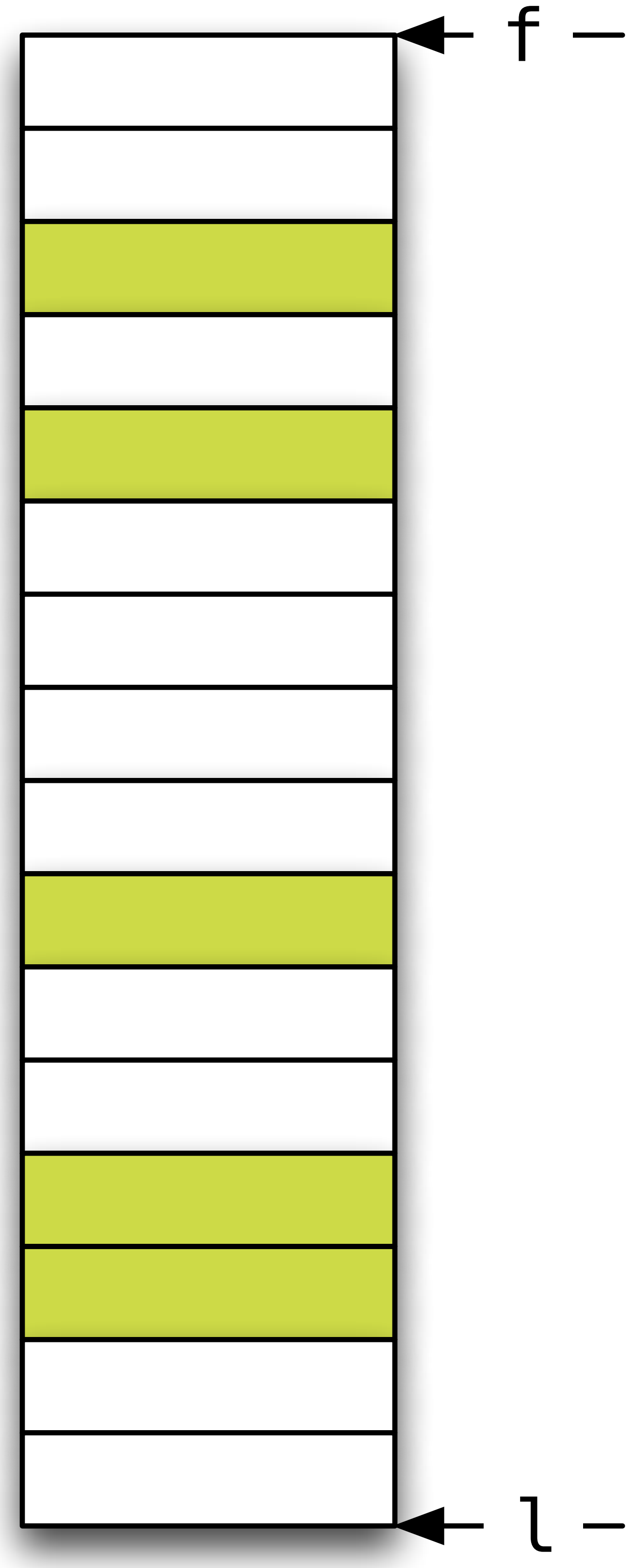
```
if (n == 1) return f + p(*f);


return rotate(stable_partition(f, m, p),
              m,
              stable_partition(m, l, p));
```

Stable Partition



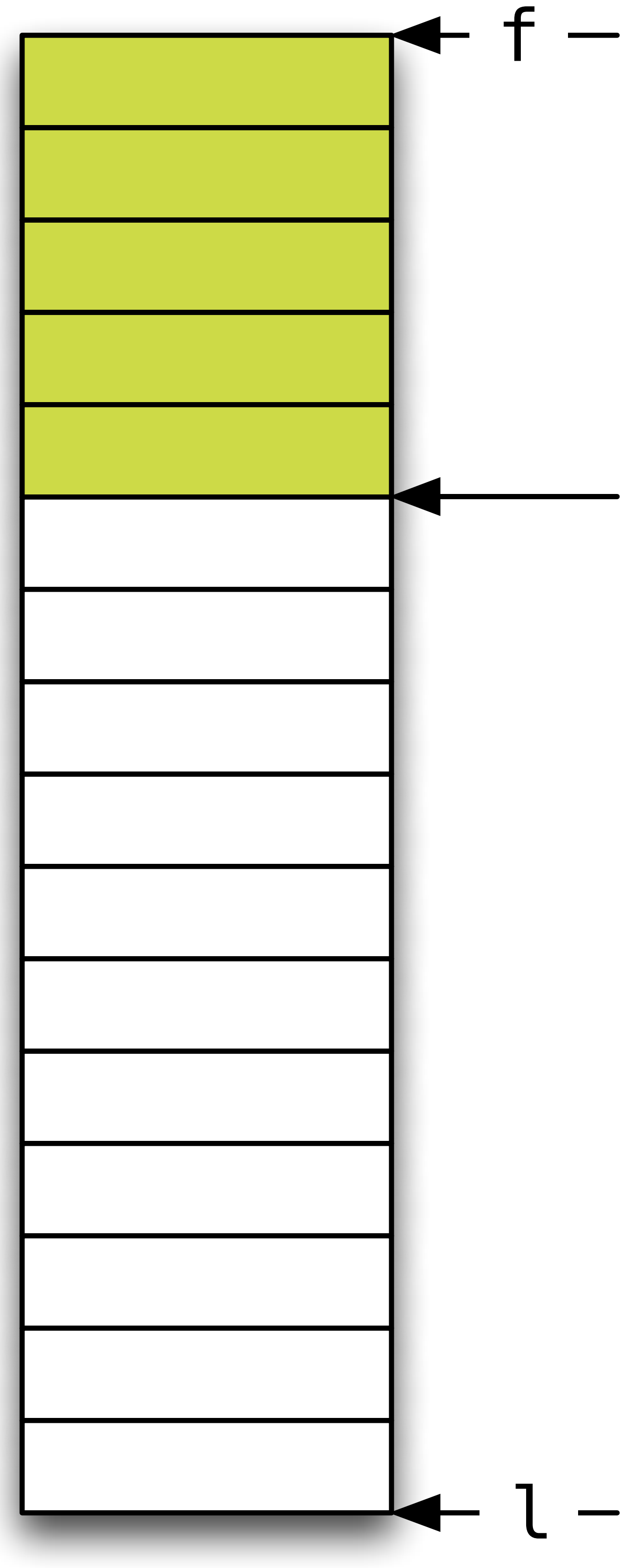
```
if (n == 1) return f + p(*f);


return rotate(stable_partition(f, m, p),
              m,
              stable_partition(m, l, p));
```

Stable Partition



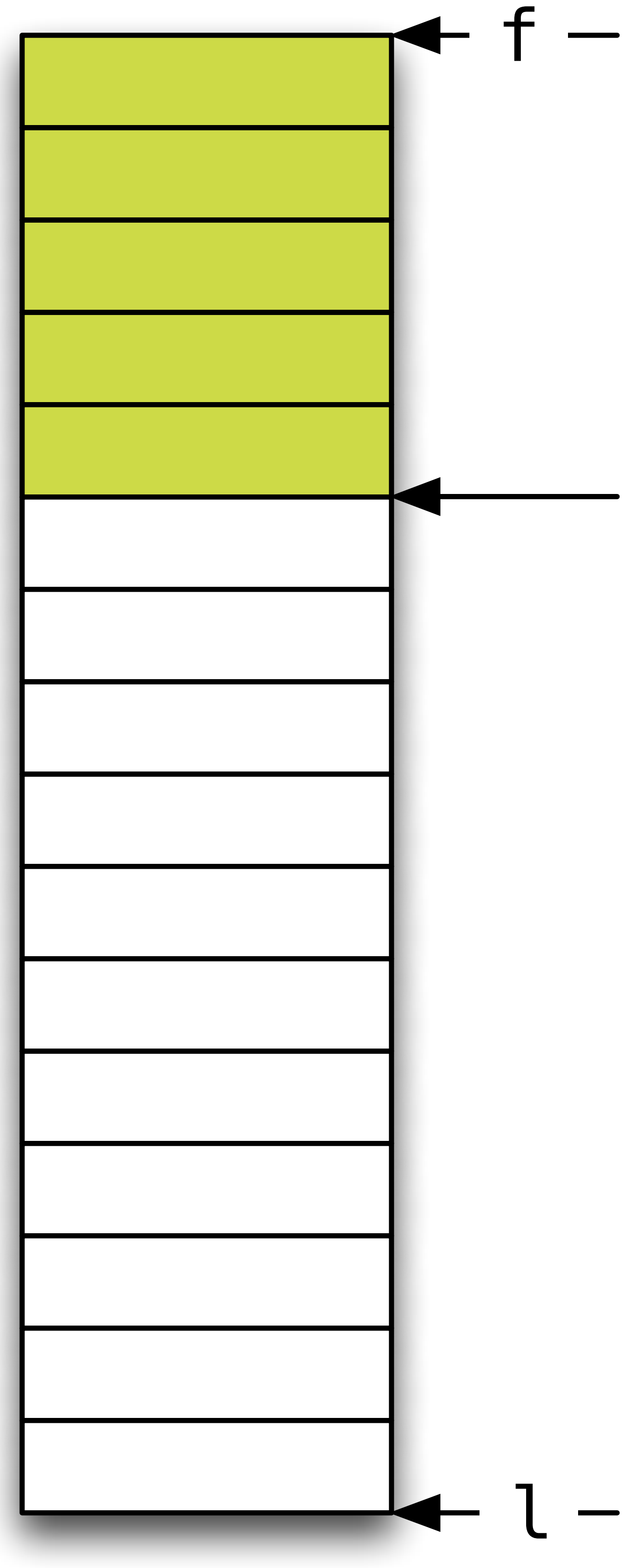
```
template <typename I,
          typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);

    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, p),
                  m,
                  stable_partition(m, l, p));
}
```

Stable Partition



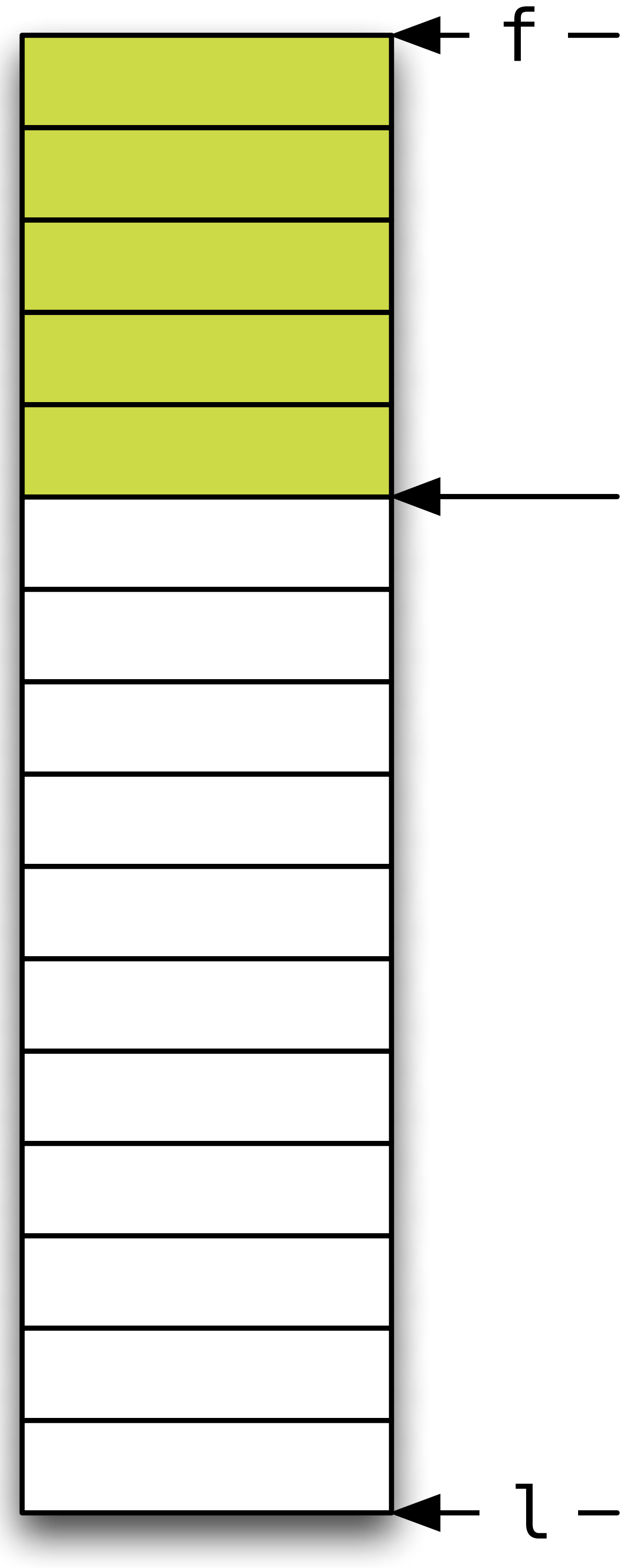
```
template <typename I,
          typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);

    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, p),
                  m,
                  stable_partition(m, l, p));
}
```

Stable Partition



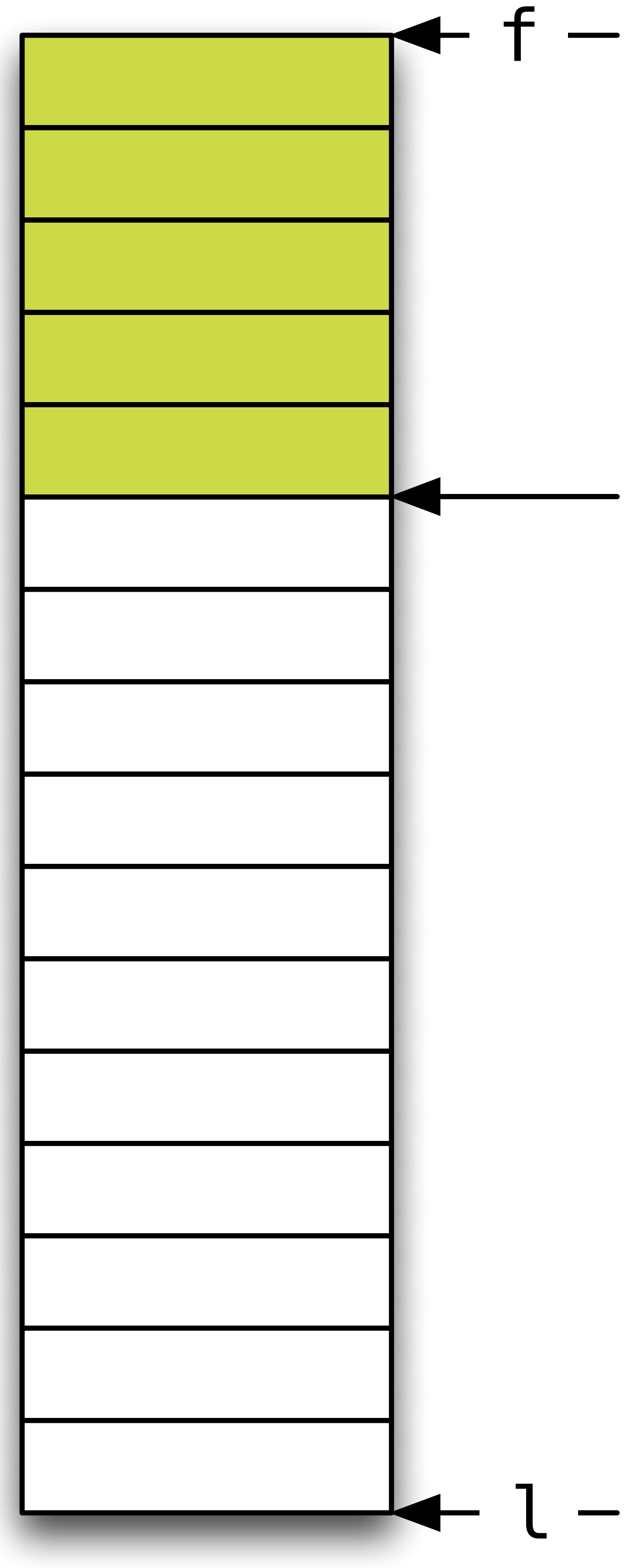
```
template <typename I,
          typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);

    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, p),
                  m,
                  stable_partition(m, l, p));
}
```

Stable Partition



```
template <typename I,
          typename P>
auto stable_partition(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(*f);

    auto m = f + (n / 2);

    return rotate(stable_partition(f, m, p),
                  m,
                  stable_partition(m, l, p));
}
```

Stable Partition



```
template <typename I,
          typename P>
auto stable_partition_position(I f, I l, P p) -> I
{
    auto n = l - f;
    if (n == 0) return f;
    if (n == 1) return f + p(f);

    auto m = f + (n / 2);

    return rotate(stable_partition_position(f, m, p),
                  m,
                  stable_partition_position(m, l, p));
}
```

Stable Partition

```
int  a[] = { 1, 2, 3, 4, 5, 5, 4, 3, 2, 1 };
bool b[] = { 0, 1, 0, 1, 0, 0, 1, 0, 1, 0 };

auto p = stable_partition_position(begin(a), end(a), [&](auto i) {
    return *(begin(b) + (i - begin(a)));
});

for (auto f = begin(a), l = p; f != l; ++f) cout << *f << " ";
cout << "^ ";
for (auto f = p, l = end(a); f != l; ++f) cout << *f << " ";
cout << endl;
```

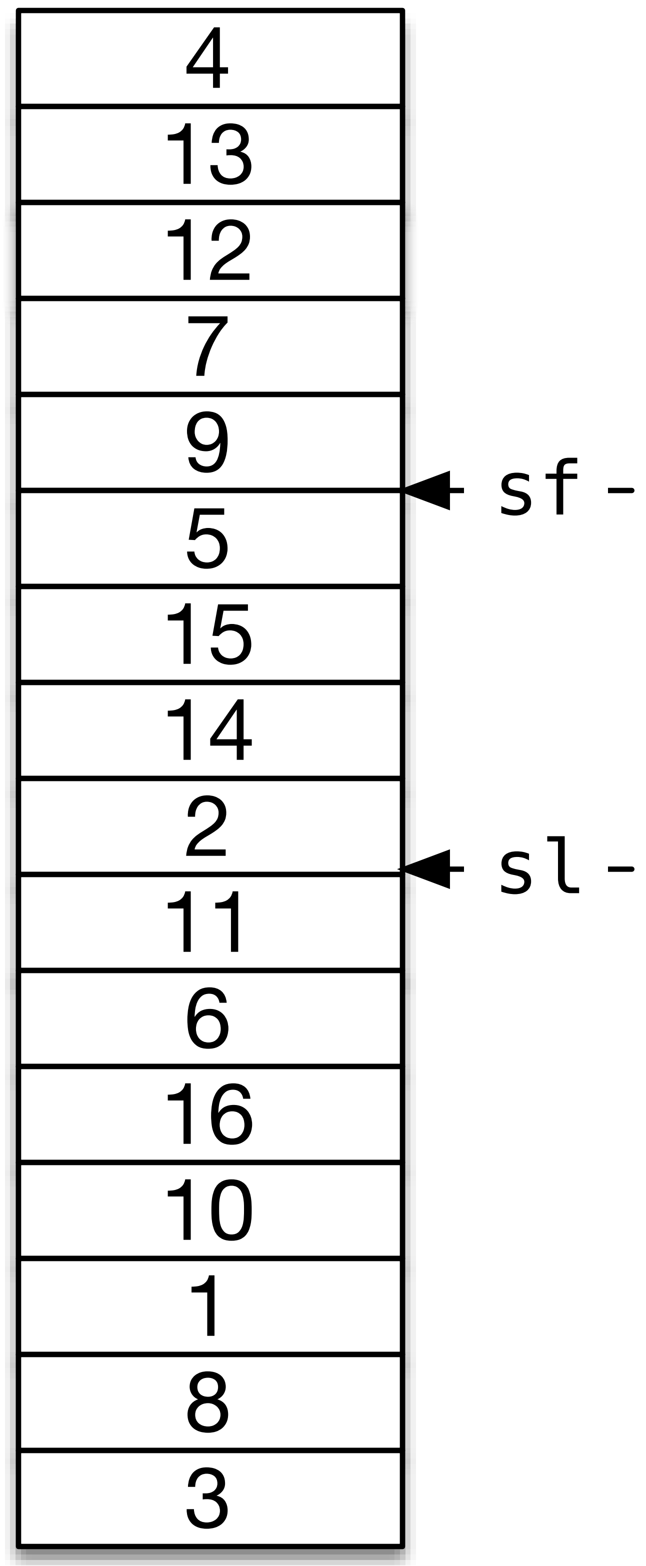
Stable Partition

```
int  a[] = { 1, 2, 3, 4, 5, 5, 4, 3, 2, 1 };
bool b[] = { 0, 1, 0, 1, 0, 0, 1, 0, 1, 0 };

auto p = stable_partition_position(begin(a), end(a), [&](auto i) {
    return *(begin(b) + (i - begin(a)));
});

for (auto f = begin(a), l = p; f != l; ++f) cout << *f << " ";
cout << "^ ";
for (auto f = p, l = end(a); f != l; ++f) cout << *f << " ";
cout << endl;
```

```
2 4 4 2 ^ 1 3 5 5 3 1
```

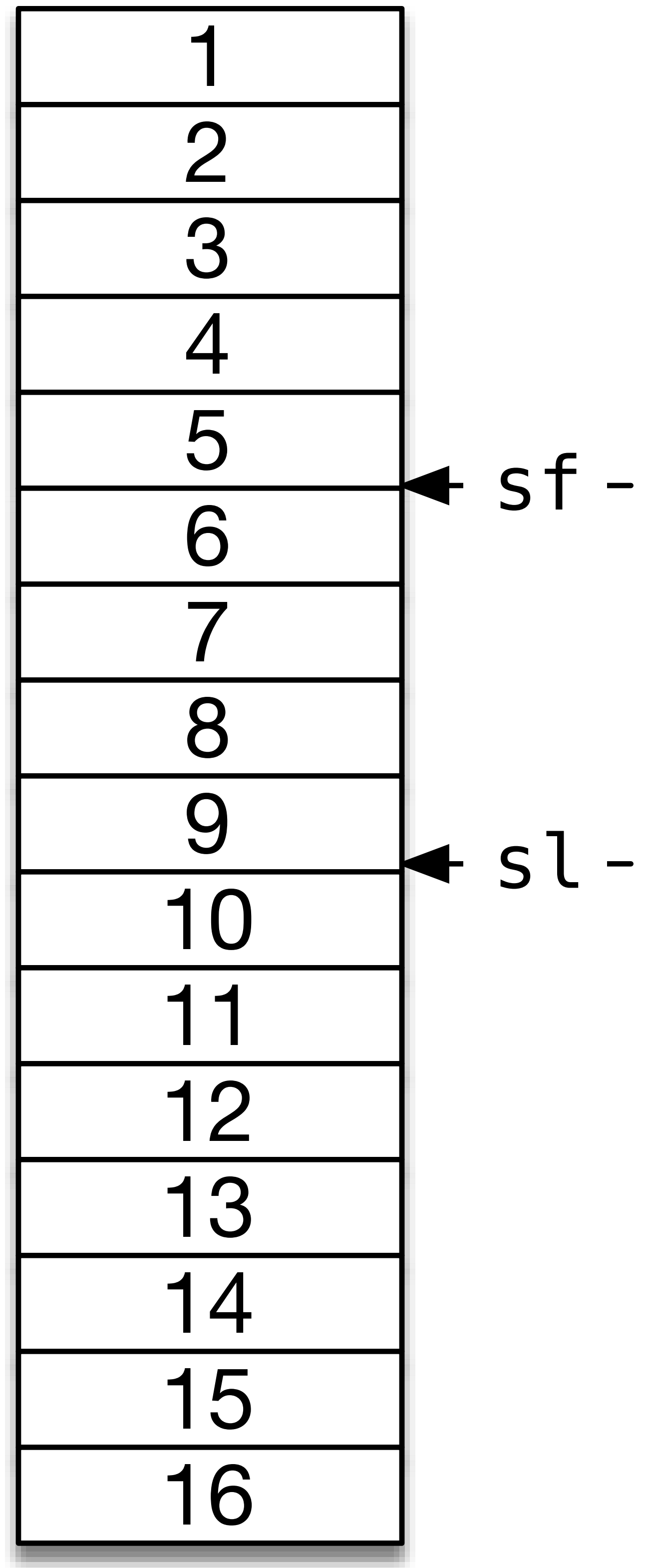
Example: Algorithms & Minimal Work

Minimize Work

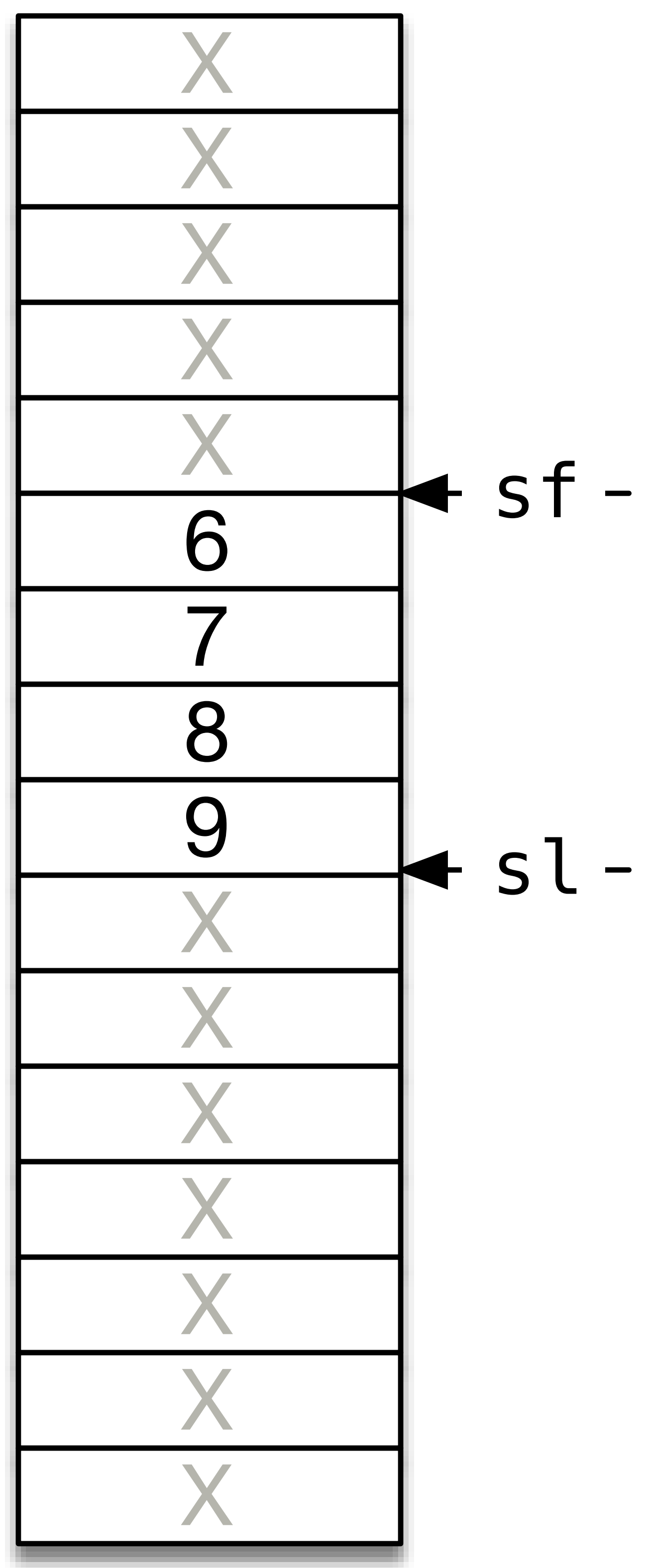


4
13
12
7
9
5
15
14
2
11
6
16
10
1
8
3

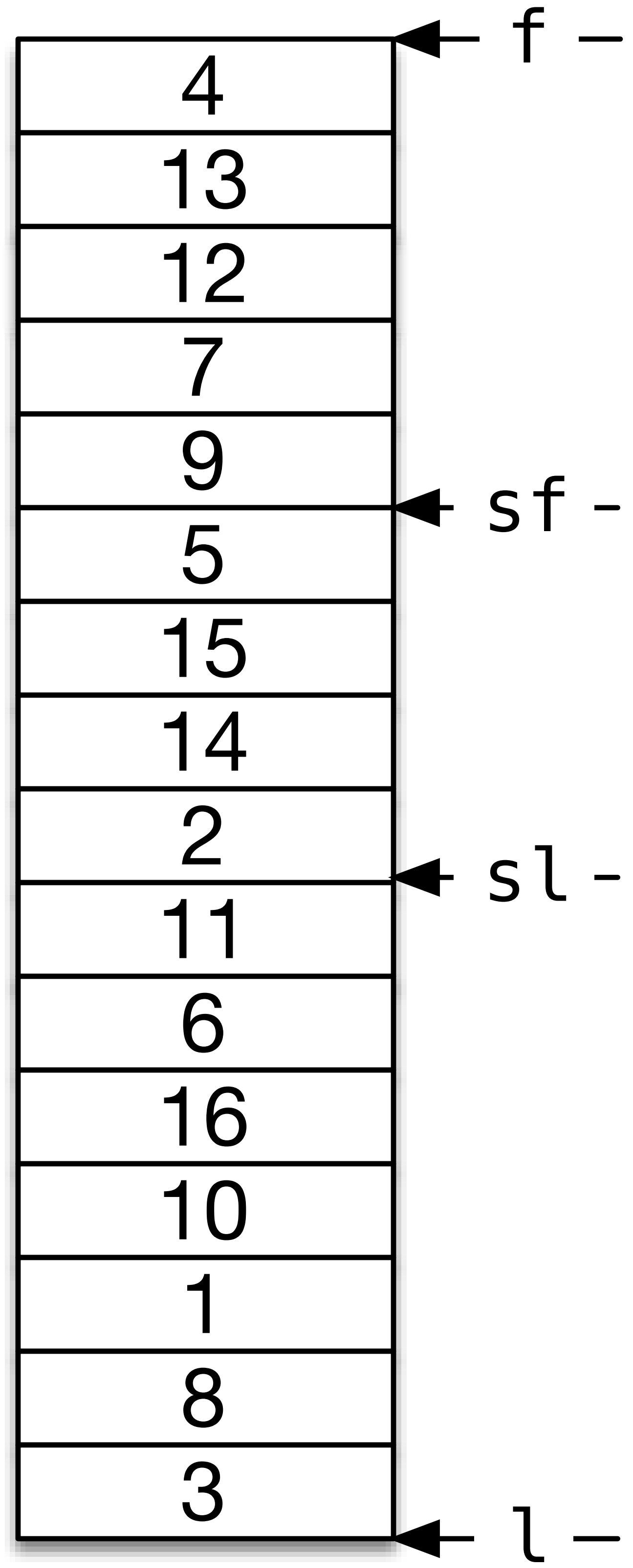
Minimize Work



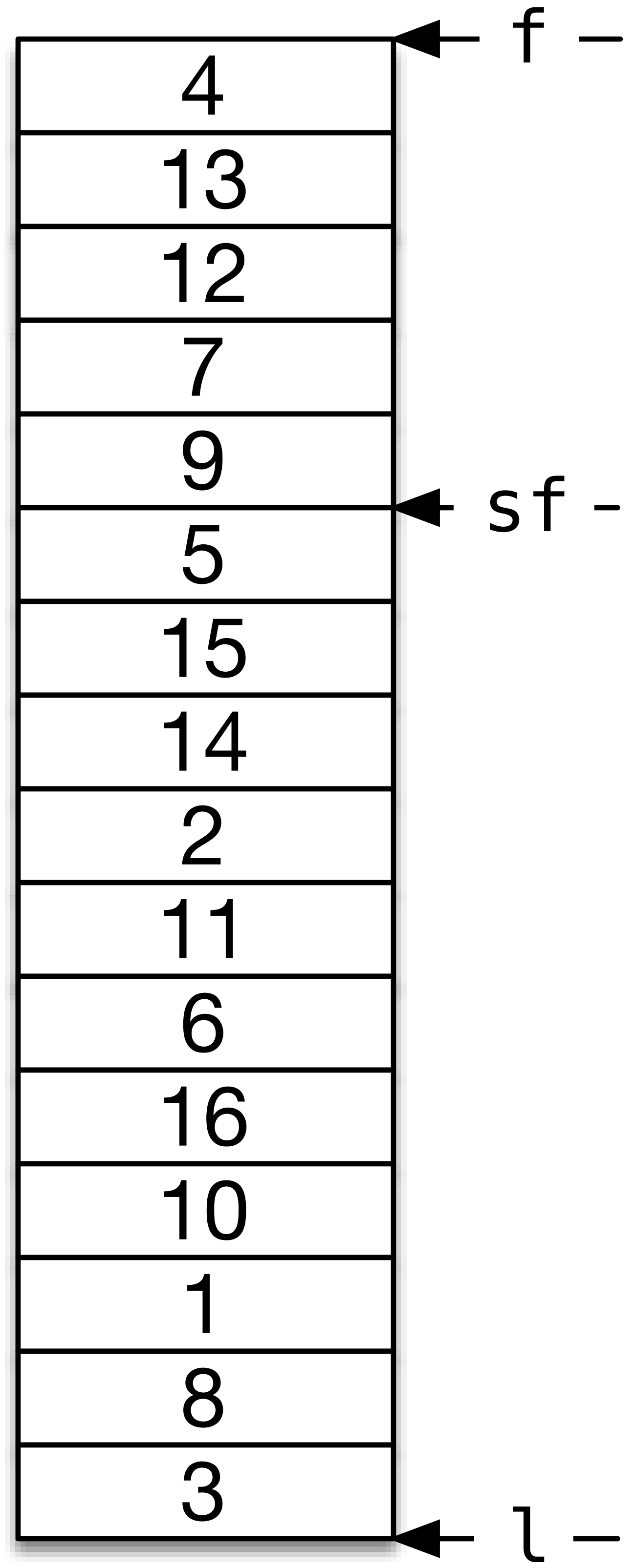
Minimize Work



Minimize Work

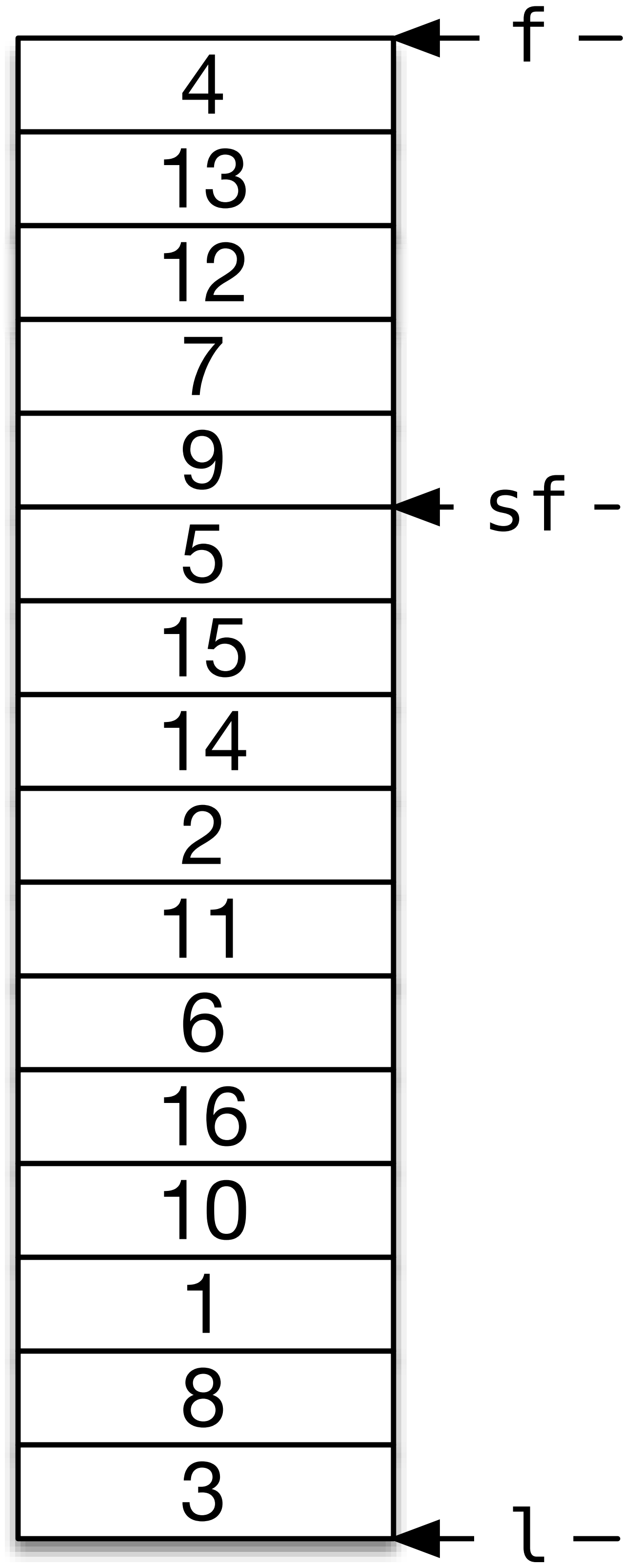


Minimize Work



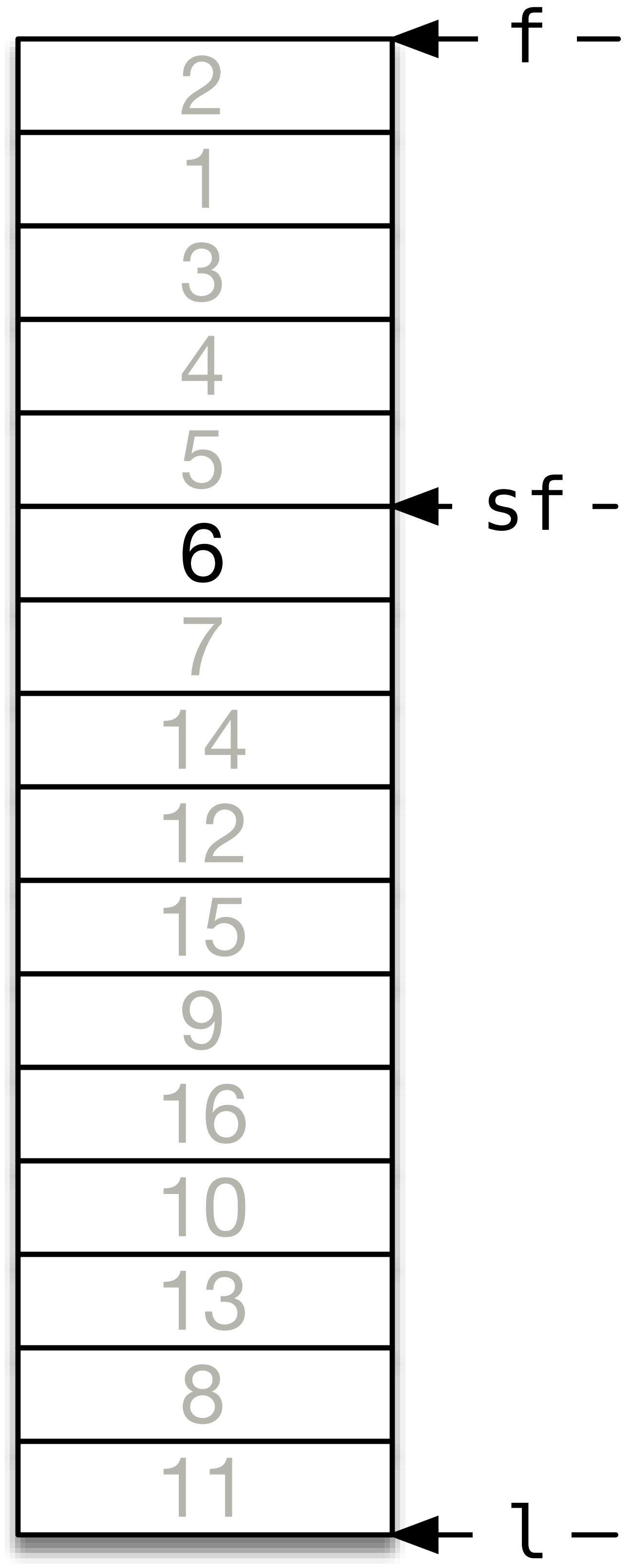
4
13
12
7
9
5
15
14
2
11
6
16
10
1
8
3

Minimize Work



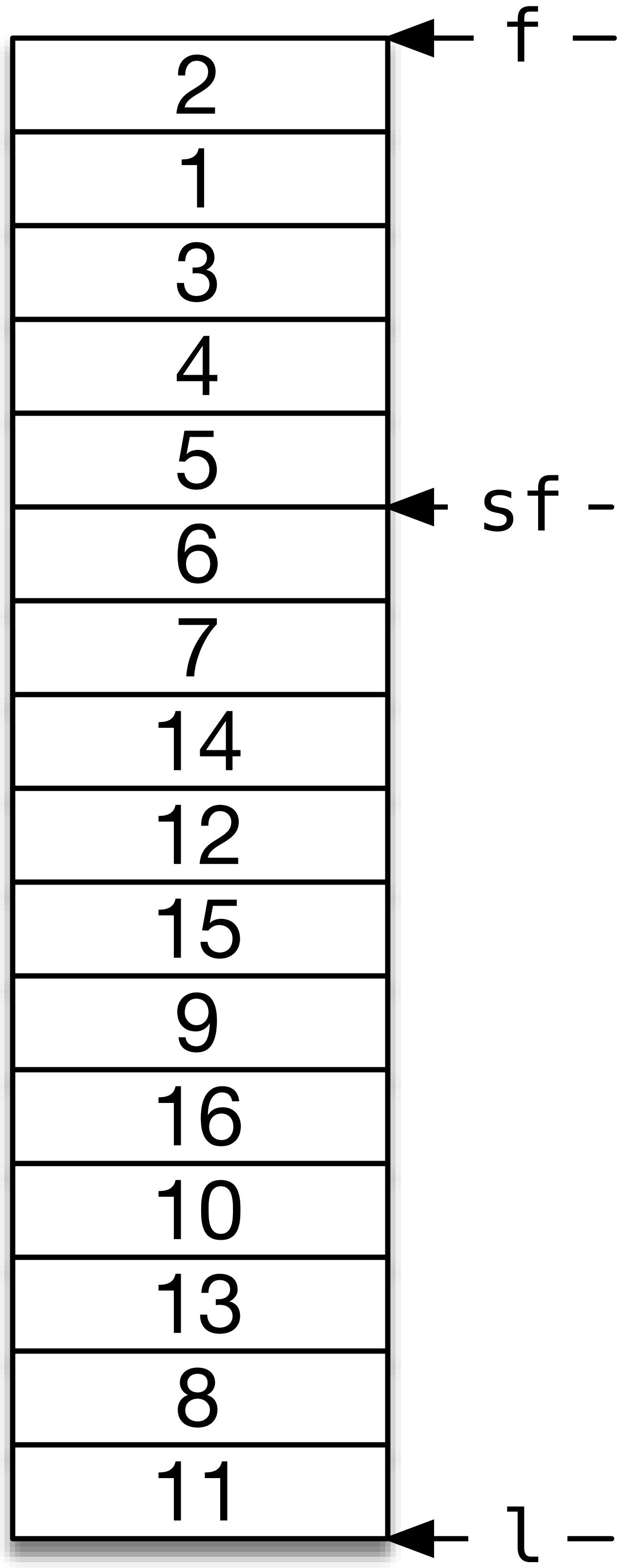
```
nth_element(f, sf, l);
```

Minimize Work


f
2
1
3
4
5
sf
6
7
14
12
15
9
16
10
13
8
11
l

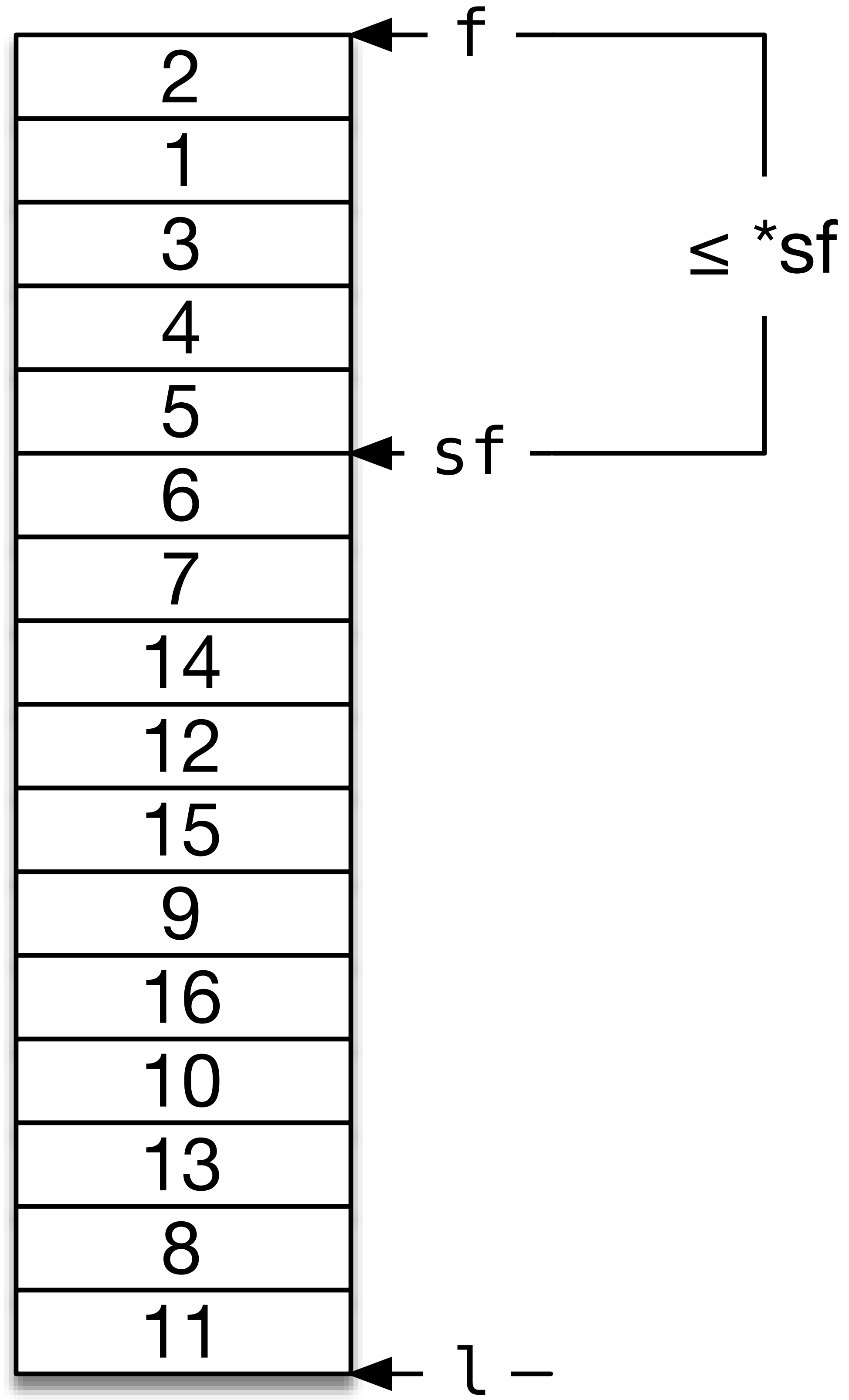
```
nth_element(f, sf, l);
```

Minimize Work



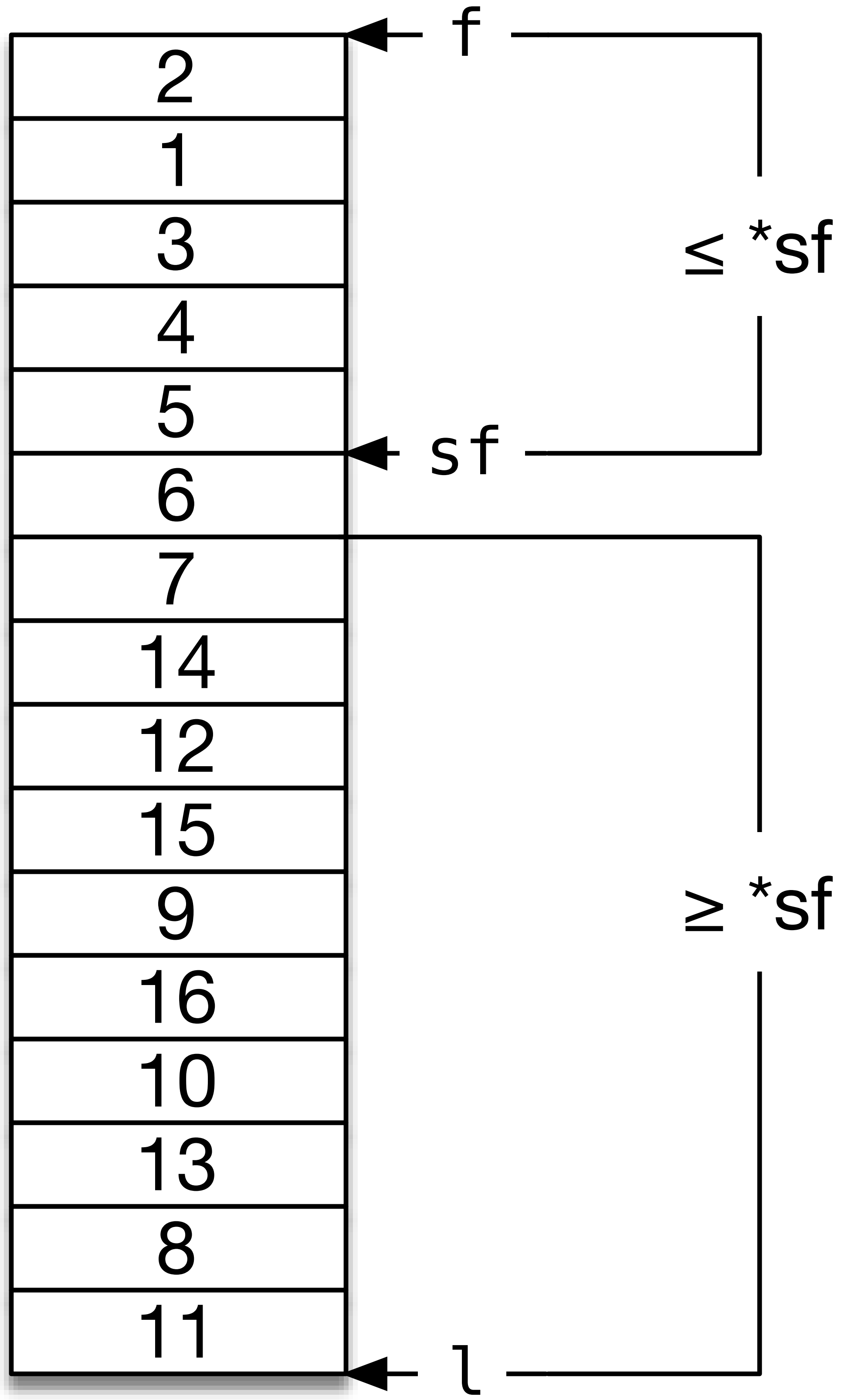
```
nth_element(f, sf, l);
```

Minimize Work



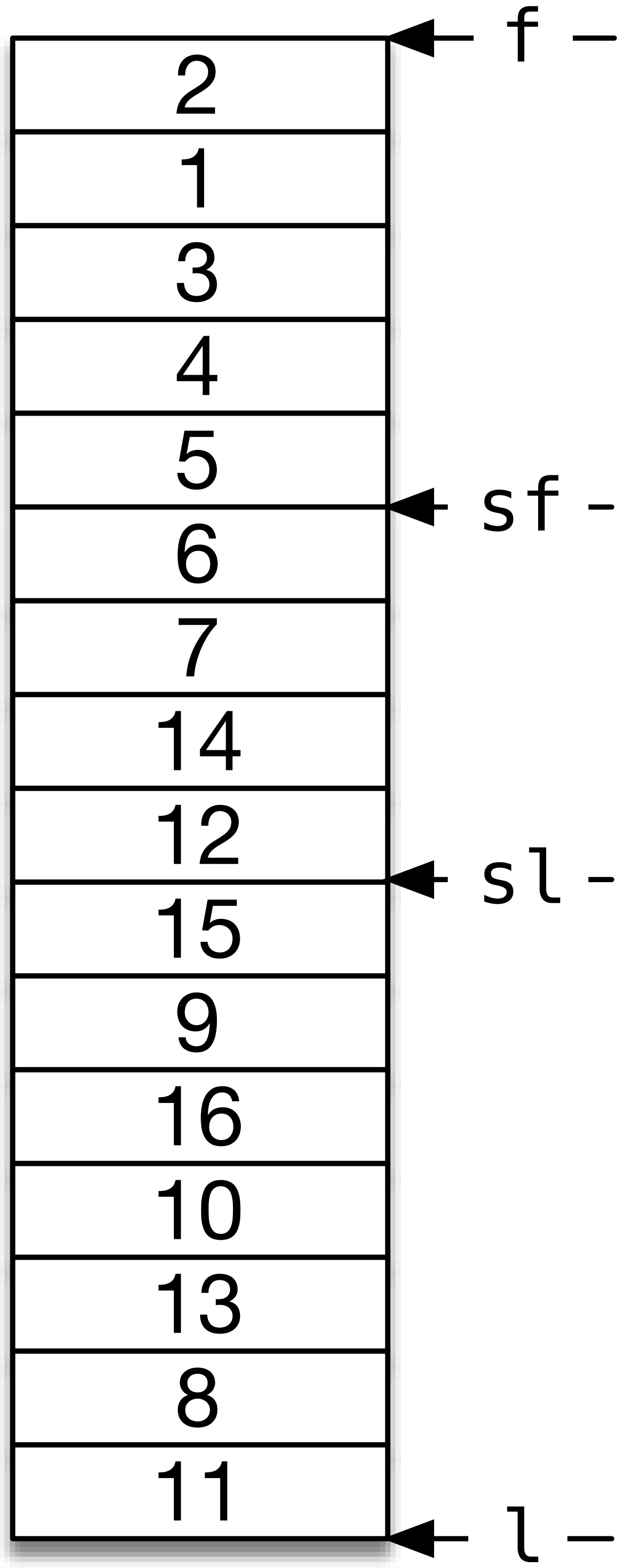
```
nth_element(f, sf, l);
```

Minimize Work



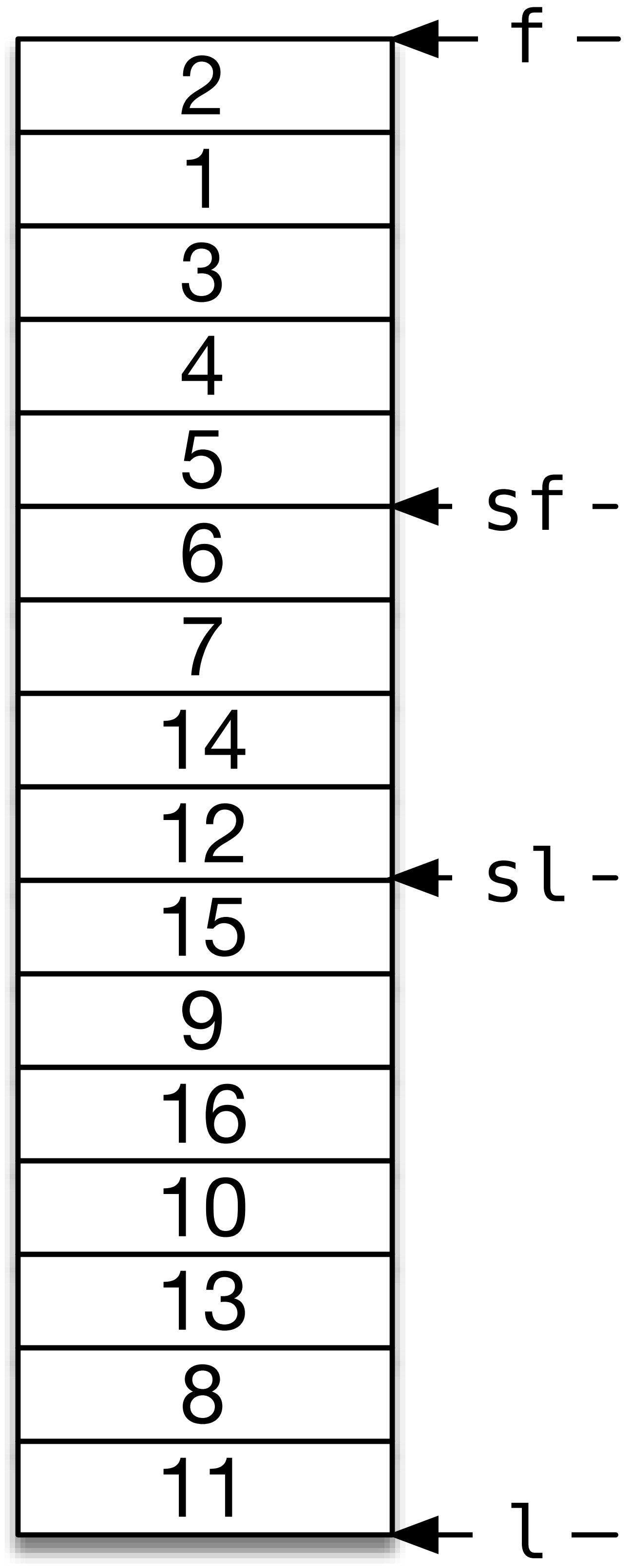
```
nth_element(f, sf, l);
```

Minimize Work



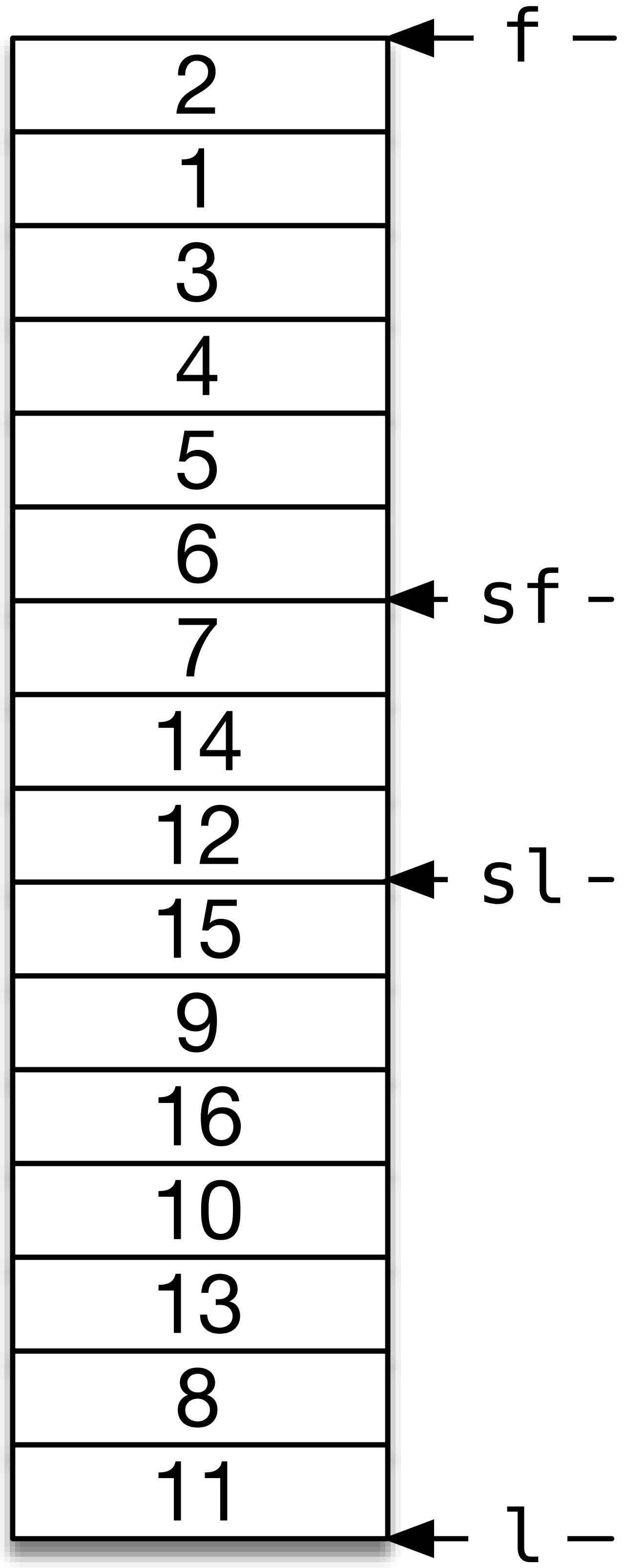
```
nth_element(f, sf, l);
```

Minimize Work



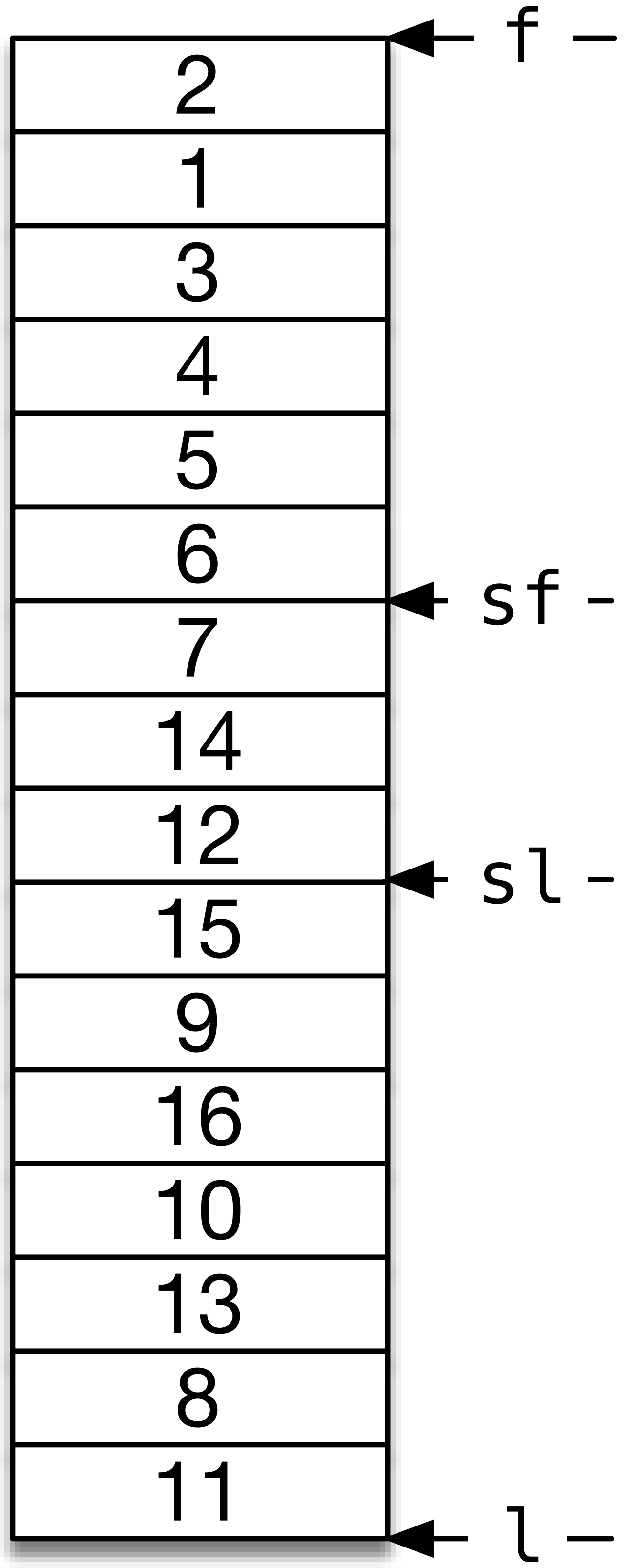
```
nth_element(f, sf, l);
++sf;
```

Minimize Work



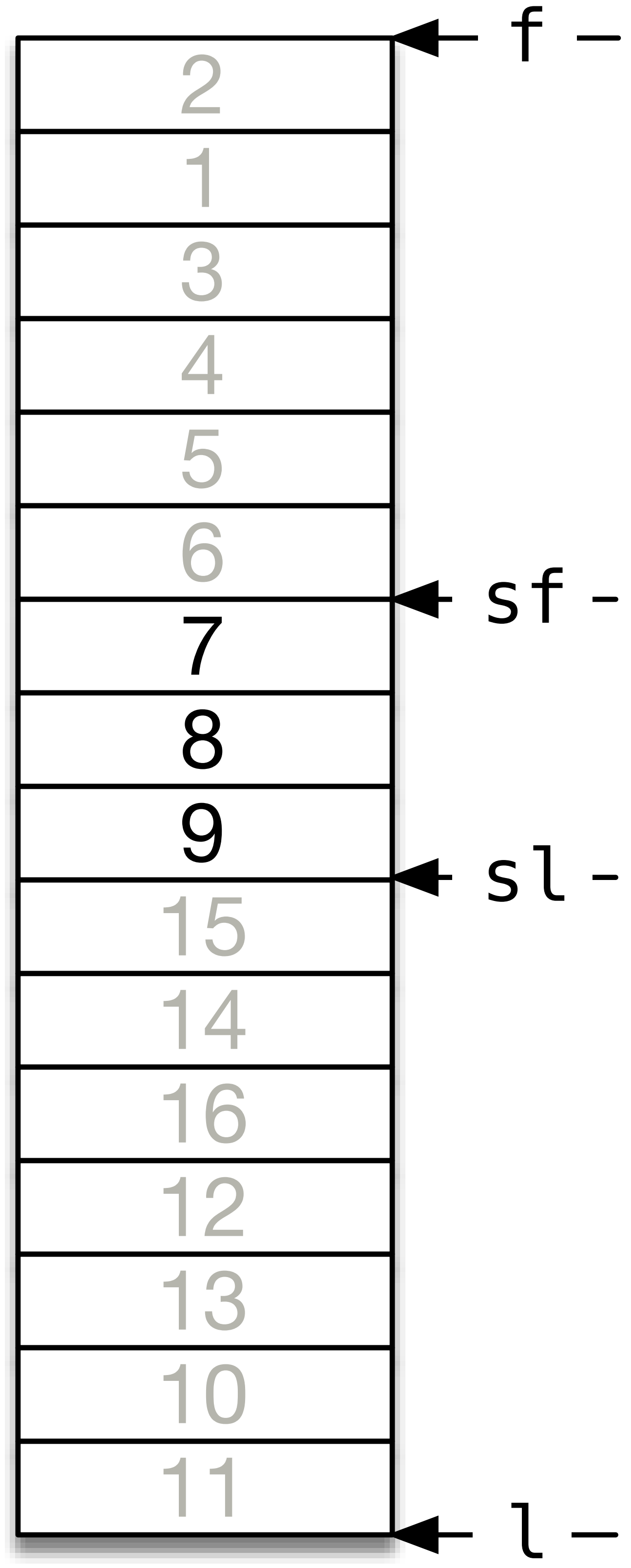
```
nth_element(f, sf, l);
++sf;
```

Minimize Work



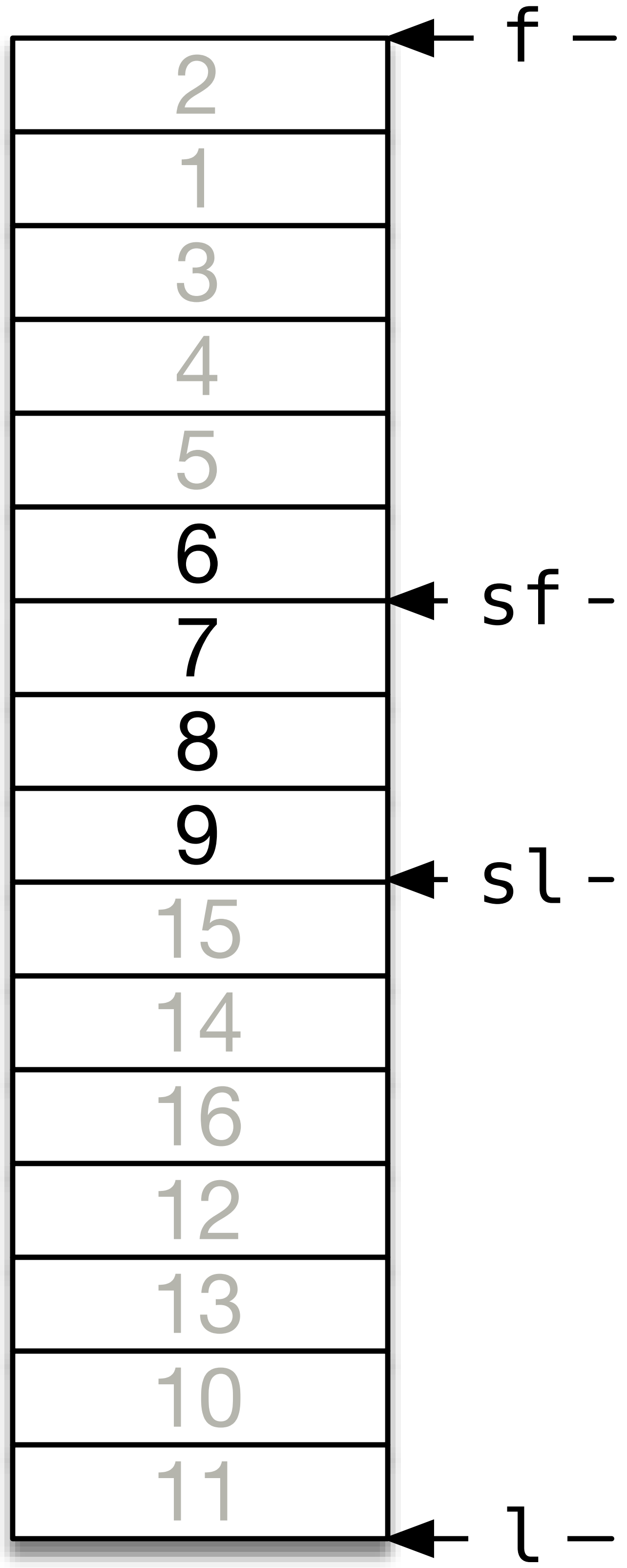
```
    nth_element(f, sf, l);
    ++sf;

partial_sort(sf, sl, l);
```

Minimize Work



```
    nth_element(f, sf, l);
    ++sf;

partial_sort(sf, sl, l);
```

Minimize Work



```
  nth_element(f, sf, l);
  ++sf;

partial_sort(sf, sl, l);
```

Minimize Work

```
if (sf == sl) return;

    nth_element(f, sf, l);
    ++sf;

partial_sort(sf, sl, l);
```

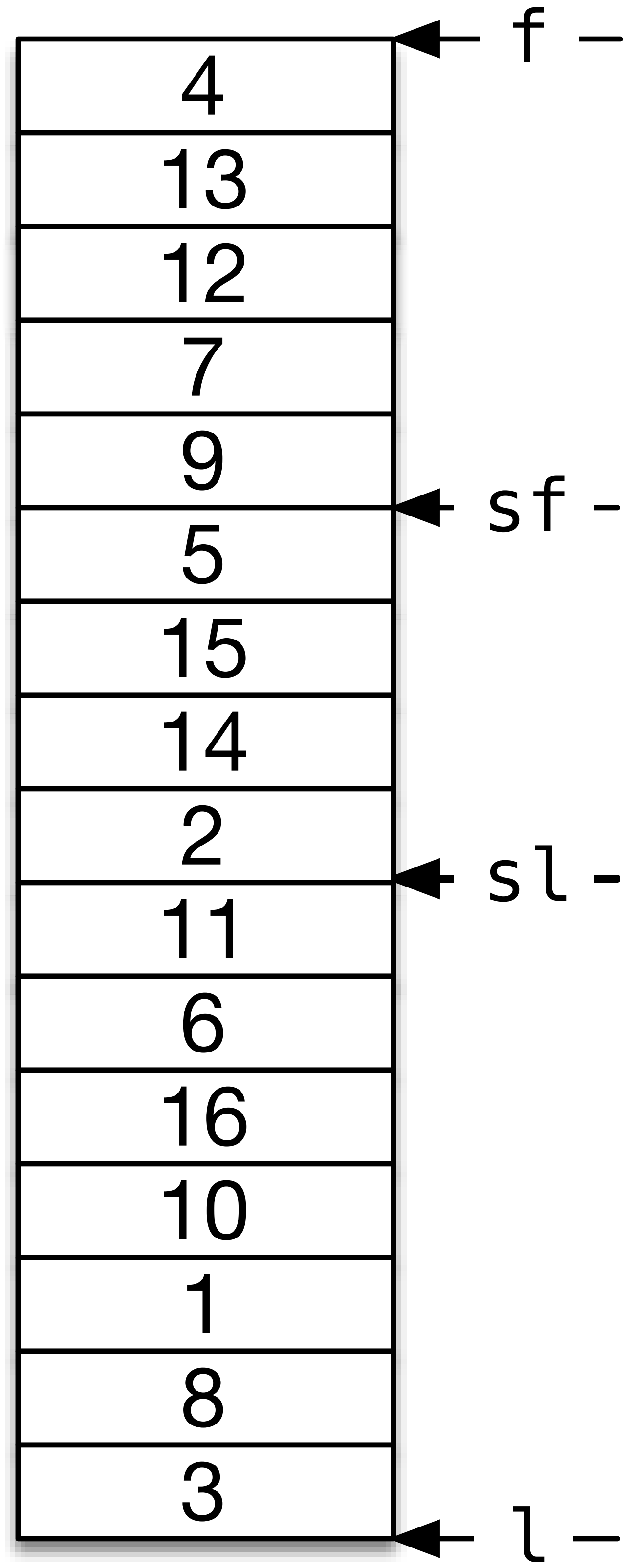
Minimize Work

```
if (sf == sl) return;
if (sf != f) {
    nth_element(f, sf, l);
    ++sf;
}
partial_sort(sf, sl, l);
```

Minimize Work

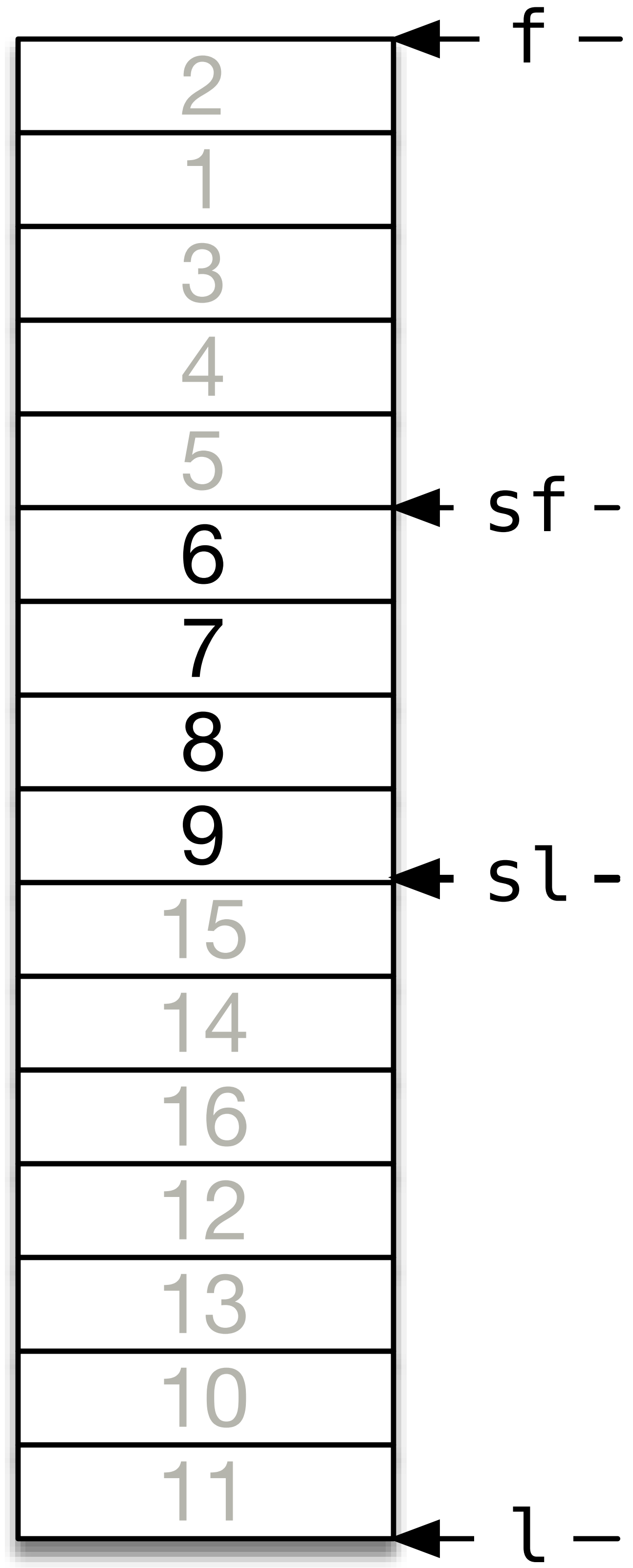
```
template <typename I> // I models RandomAccessIterator
void sort_subrange(I f, I l, I sf, I sl)
{
    if (sf == sl) return;
    if (sf != f) {
        nth_element(f, sf, l);
        ++sf;
    }
    partial_sort(sf, sl, l);
}
```

Minimize Work



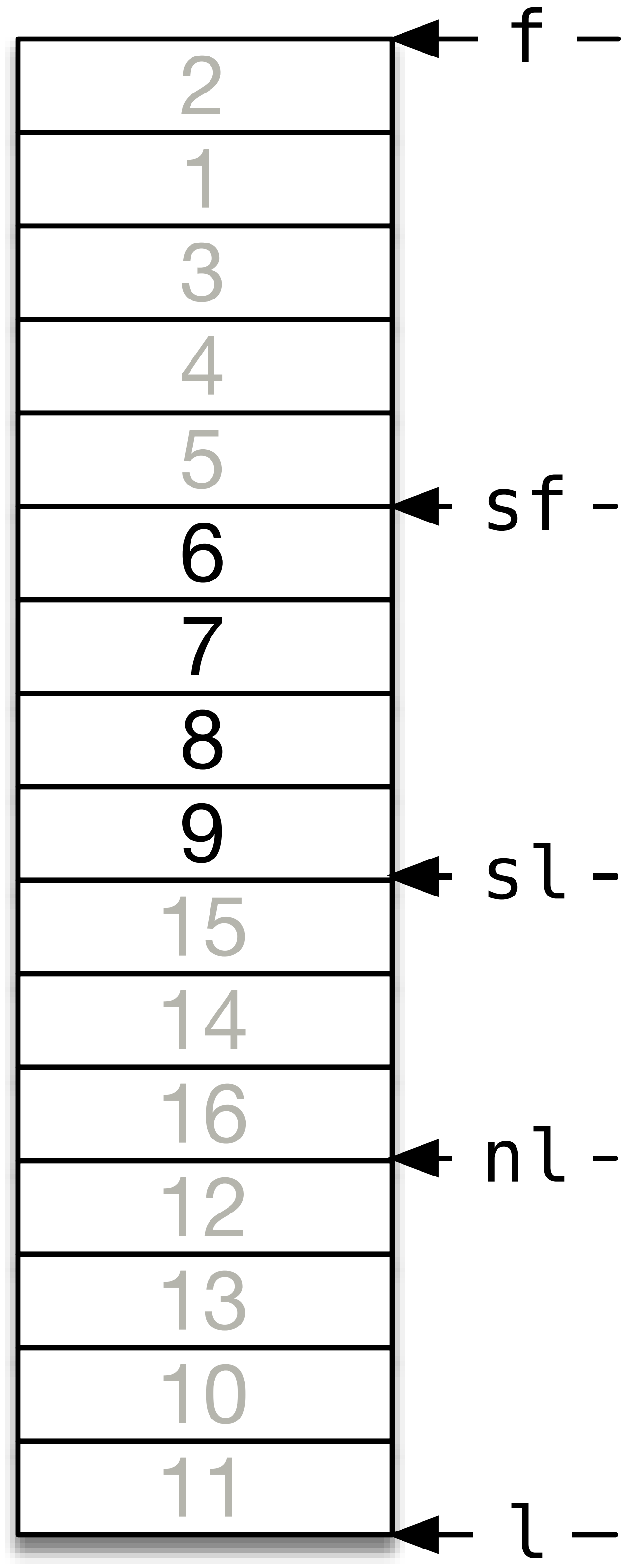
```
sort_subrange(f, l, sf, sl);
```

Minimize Work



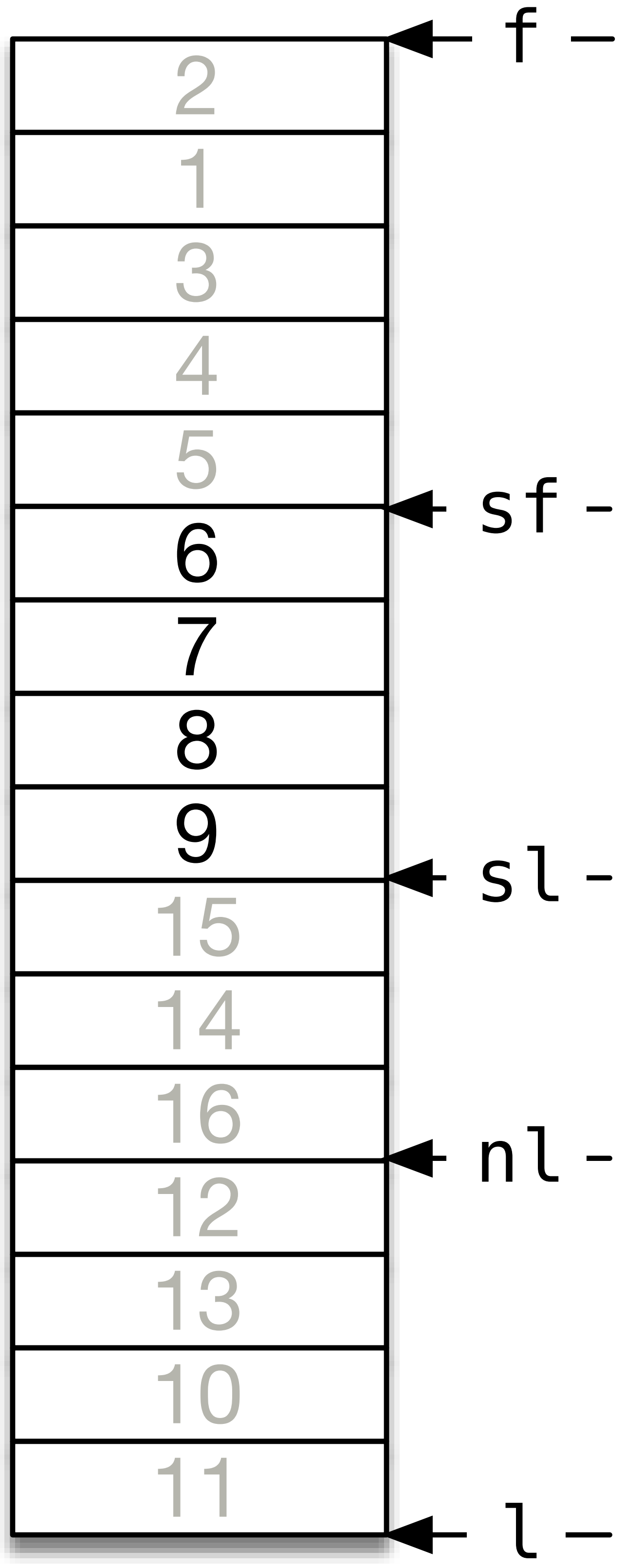
```
sort_subrange(f, l, sf, sl);
```

Minimize Work



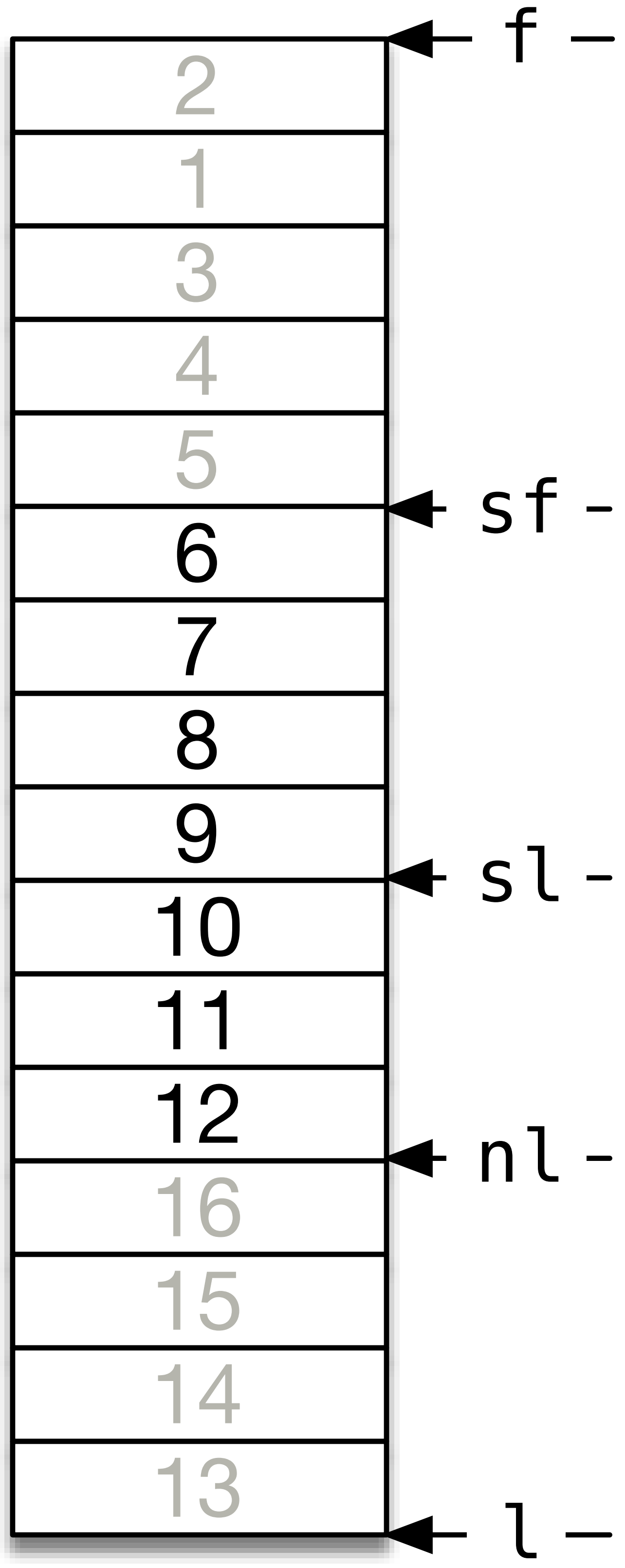
```
sort_subrange(f, l, sf, sl);
```

Minimize Work



```
sort_subrange(f, l, sf, sl);
partial_sort(sl, nl, l);
```

Minimize Work



```
sort_subrange(f, l, sf, sl);
partial_sort(sl, nl, l);
```

What is an *incidental* data structure?

What is an *incidental* data structure?

Definition: An incidental data structure is a data structure that occurs within a system when there is no object representing the structure as a whole.

What is an *incidental* data structure?

Definition: An incidental data structure is a data structure that occurs within a system when there is no object representing the structure as a whole.

Structures formed in the absence of a whole/part relationship

Why no incidental data structures?

- They cause ambiguities and break our ability to reason about code locally

Incidental Data Structures

Incidental Data Structures

- Delegates

Incidental Data Structures

- Delegates
- Message handlers

Incidental Data Structures

Incidental Data Structures

- Self-referential interface

Incidental Data Structures

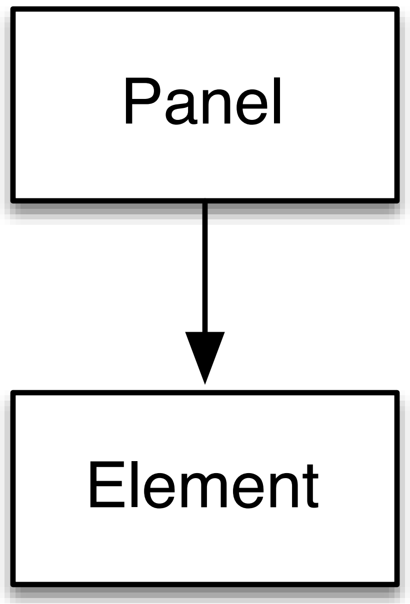
- Self-referential interface

```
class UIElement { };

class UIElementCollection {
  public:
    void Add(shared_ptr<UIElement>);
};

class Panel : public UIElement {
  public:
    shared_ptr<UIElementCollection> Children() const;
};
```

Incidental Data Structures

- Self-referential interface

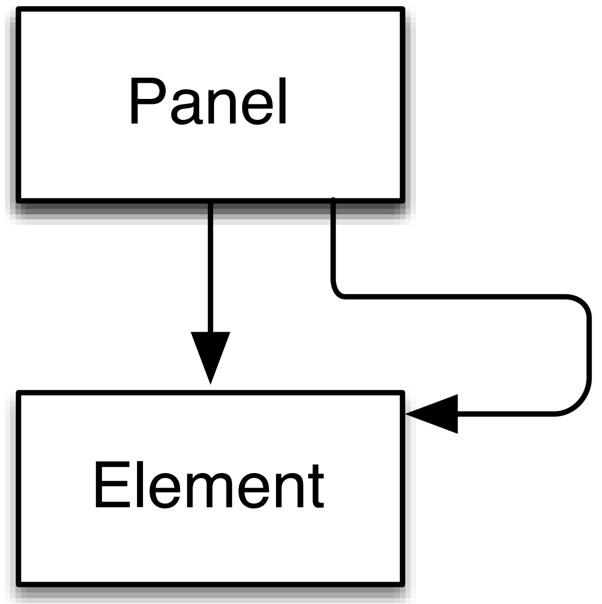
```
class UIElement { };

class UIElementCollection {
  public:
    void Add(shared_ptr<UIElement>);
};

class Panel : public UIElement {
  public:
    shared_ptr<UIElementCollection> Children() const;
};

panel->Children()->Add(element);
```



Incidental Data Structures

- Self-referential interface

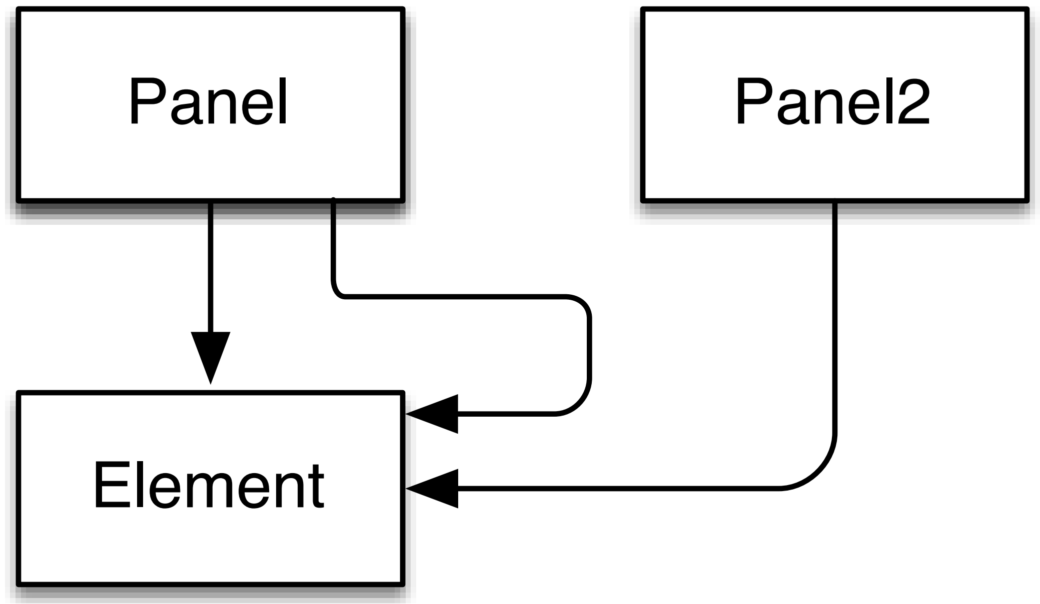
```
class UIElement { };

class UIElementCollection {
  public:
    void Add(shared_ptr<UIElement>);
};

class Panel : public UIElement {
  public:
    shared_ptr<UIElementCollection> Children() const;
};

panel->Children()->Add(element);
panel->Children()->Add(element);
```



Incidental Data Structures

- Self-referential interface

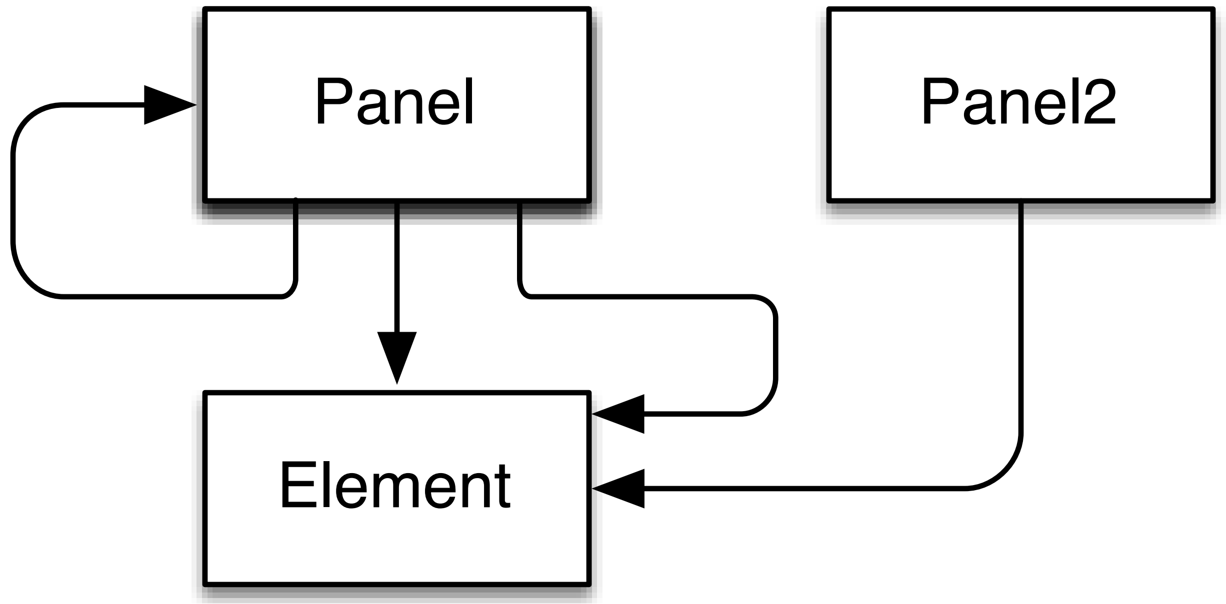
```
class UIElement { };

class UIElementCollection {
  public:
    void Add(shared_ptr<UIElement>);
};

class Panel : public UIElement {
  public:
    shared_ptr<UIElementCollection> Children() const;
};

panel->Children()->Add(element);
panel->Children()->Add(element);
panel2->Children()->Add(element);
```



Incidental Data Structures

- Self-referential interface

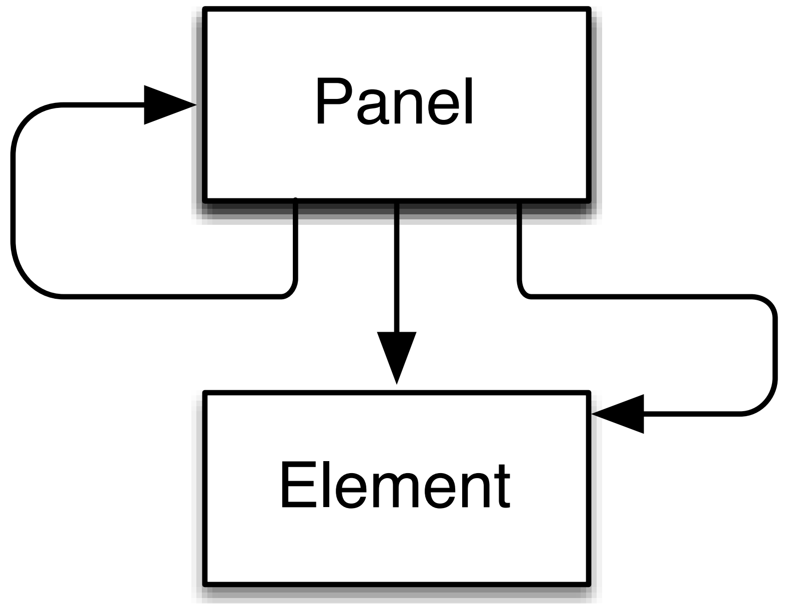
```
class UIElement { };

class UIElementCollection {
  public:
    void Add(shared_ptr<UIElement>);
};

class Panel : public UIElement {
  public:
    shared_ptr<UIElementCollection> Children() const;
};

panel->Children()->Add(element);
panel->Children()->Add(element);
panel2->Children()->Add(element);
panel->Children()->Add(panel);
```



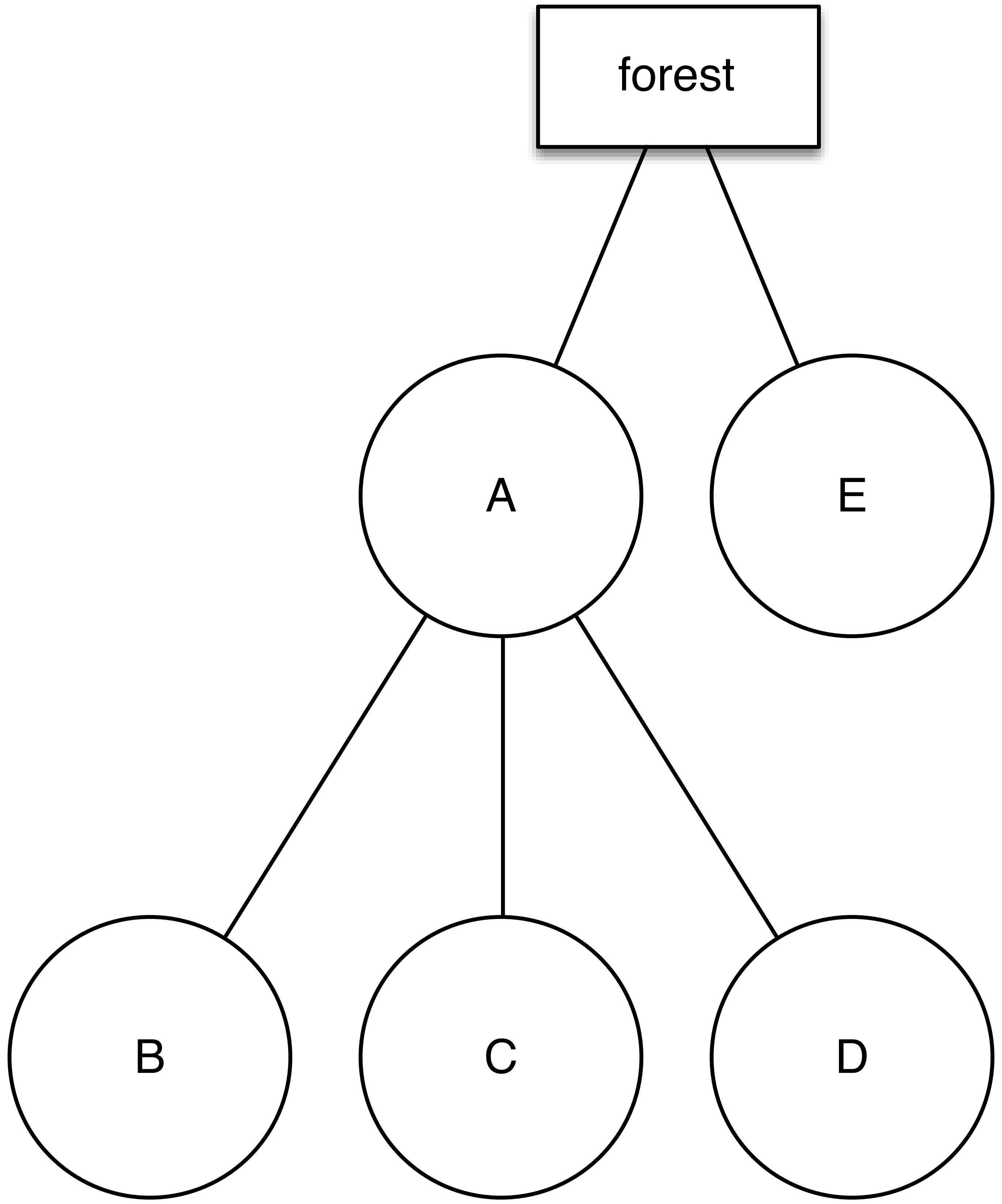
Incidental Data Structures

- Self-referential interface

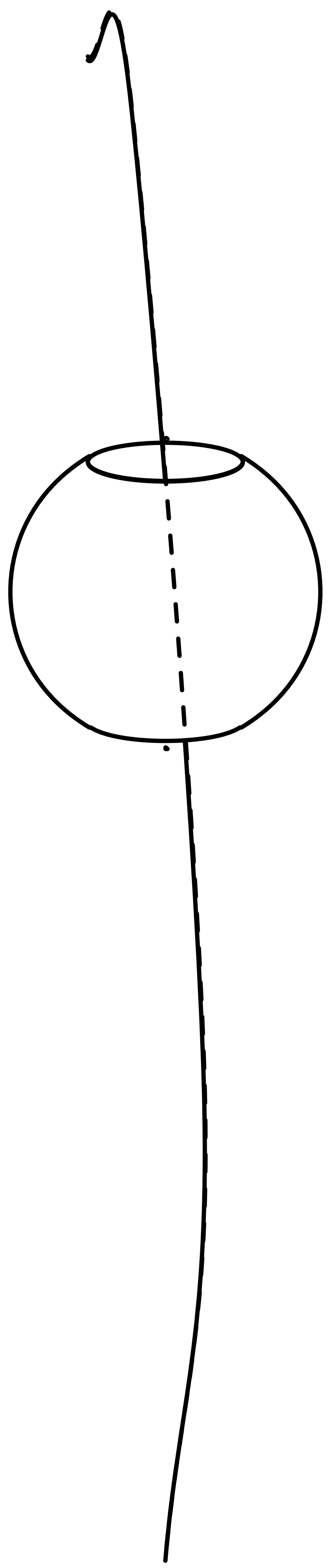
```
class UIElement { };

class UIElementCollection {
  public:
    void Add(shared_ptr<UIElement>);
};

class Panel : public UIElement {
  public:
    shared_ptr<UIElementCollection> Children() const;
};

panel->Children()->Add(element);
panel->Children()->Add(element);
panel2->Children()->Add(element);
panel->Children()->Add(panel);
```



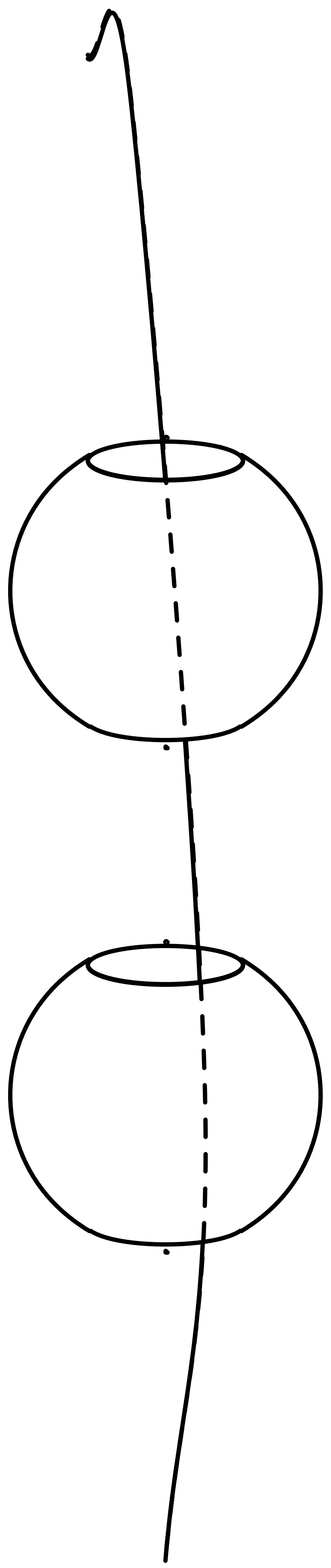
Hierarchies



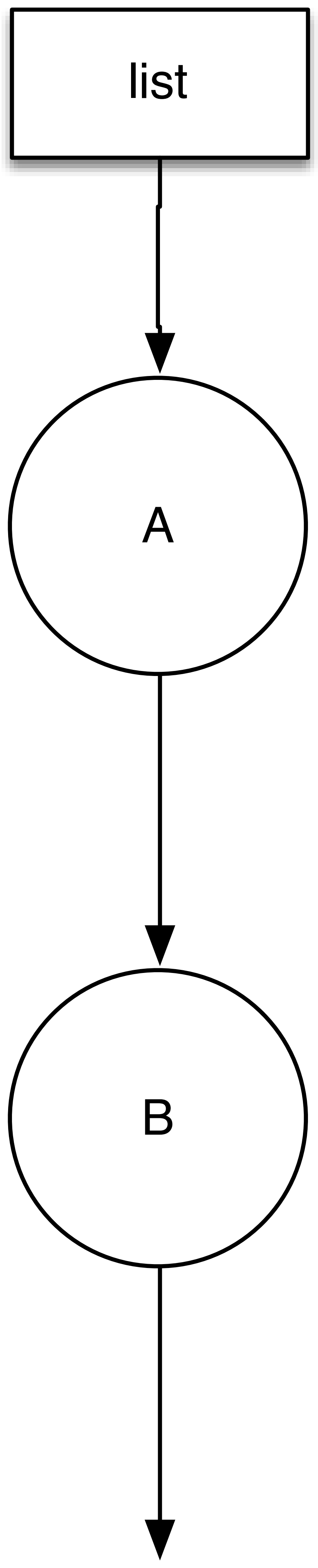
Hierarchies



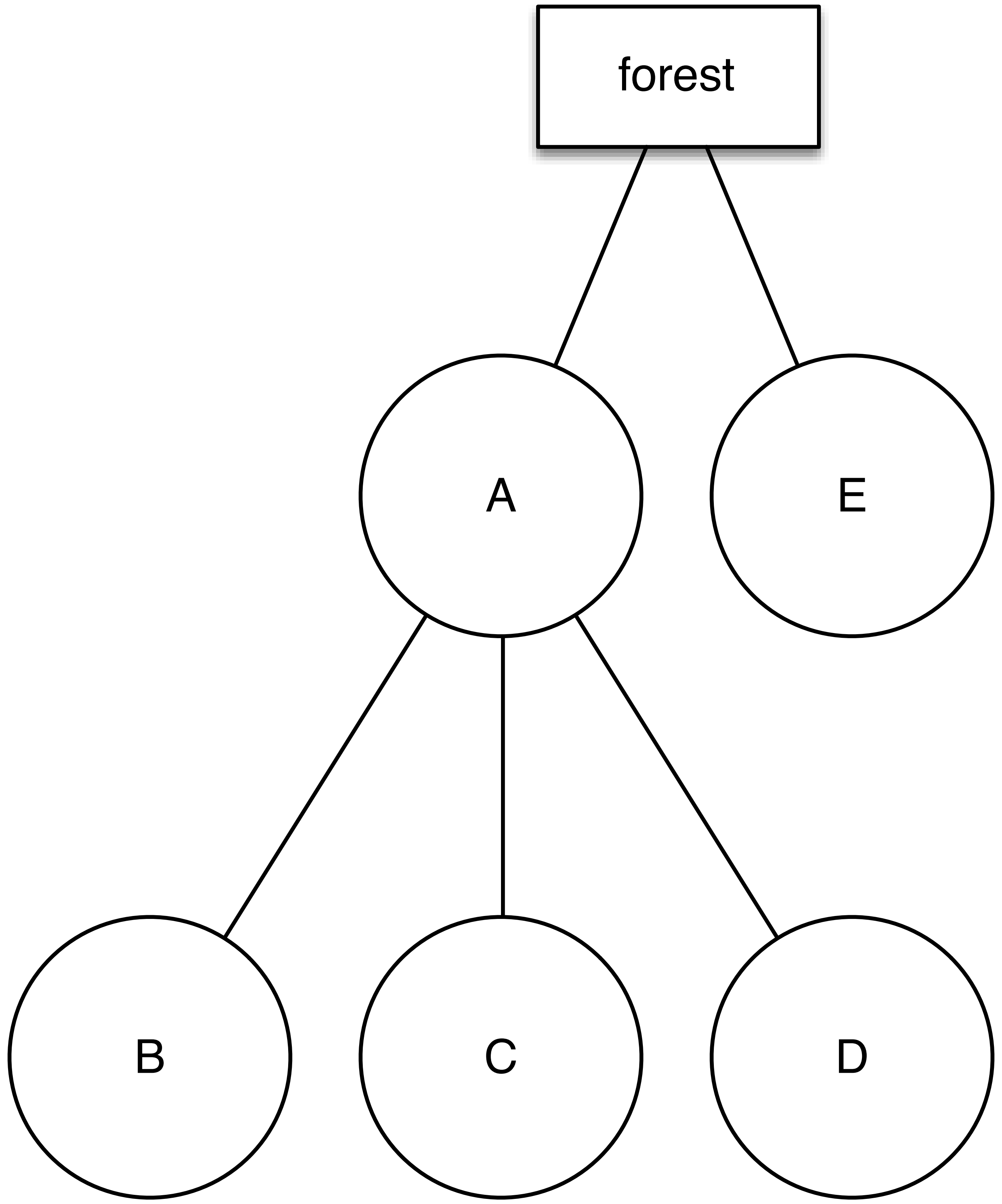
Hierarchies



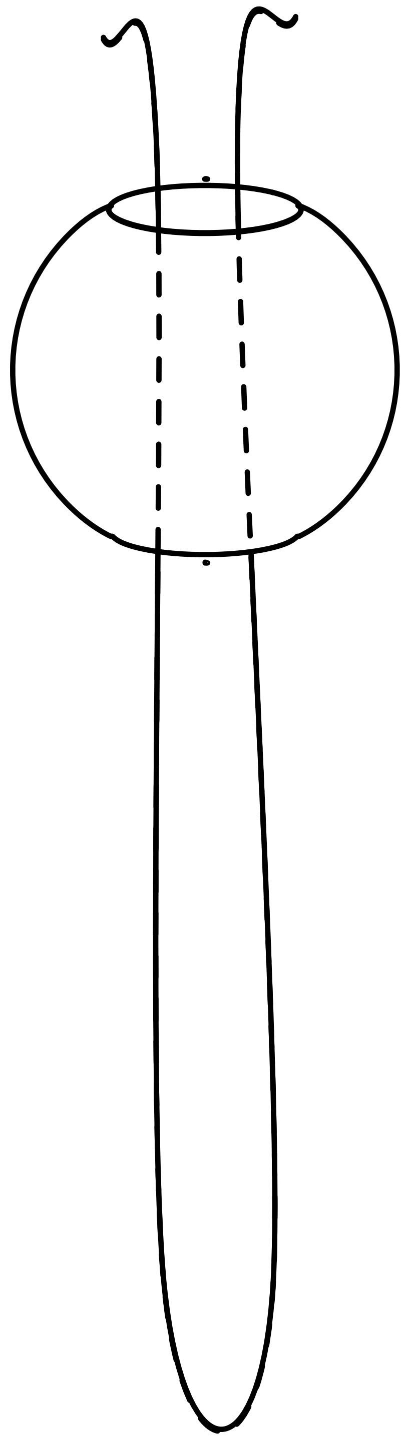
Hierarchies



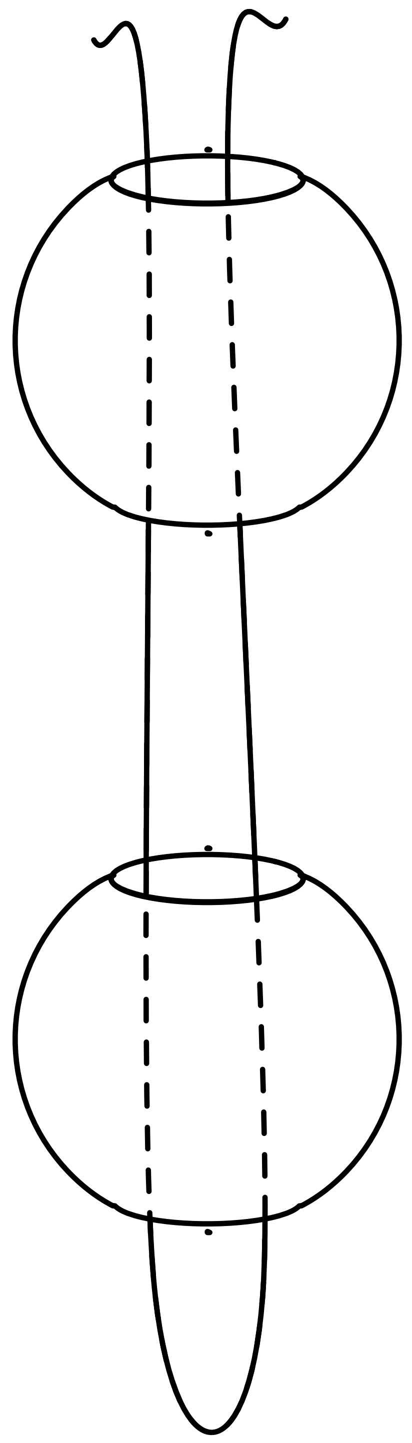
Hierarchies



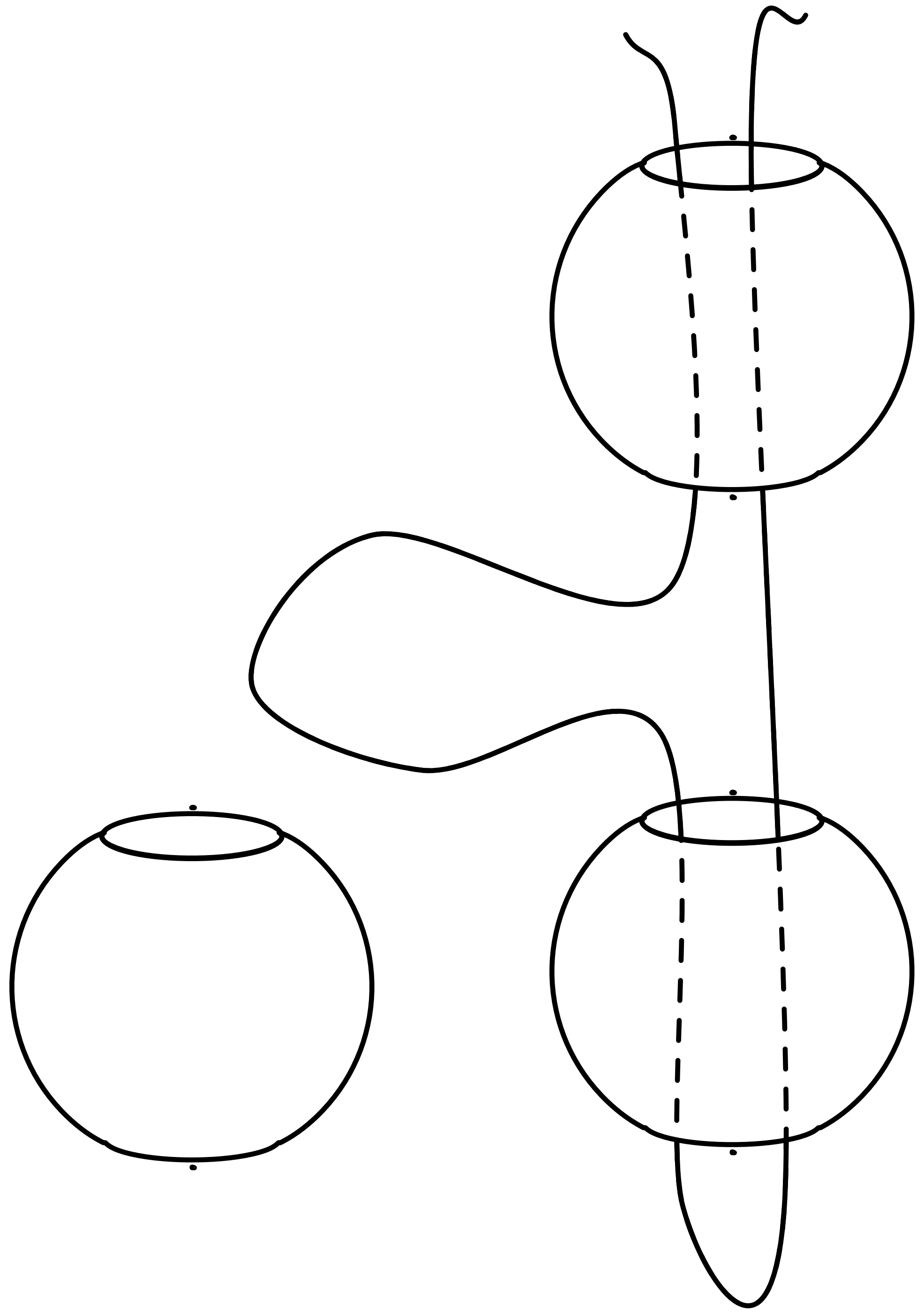
Hierarchies



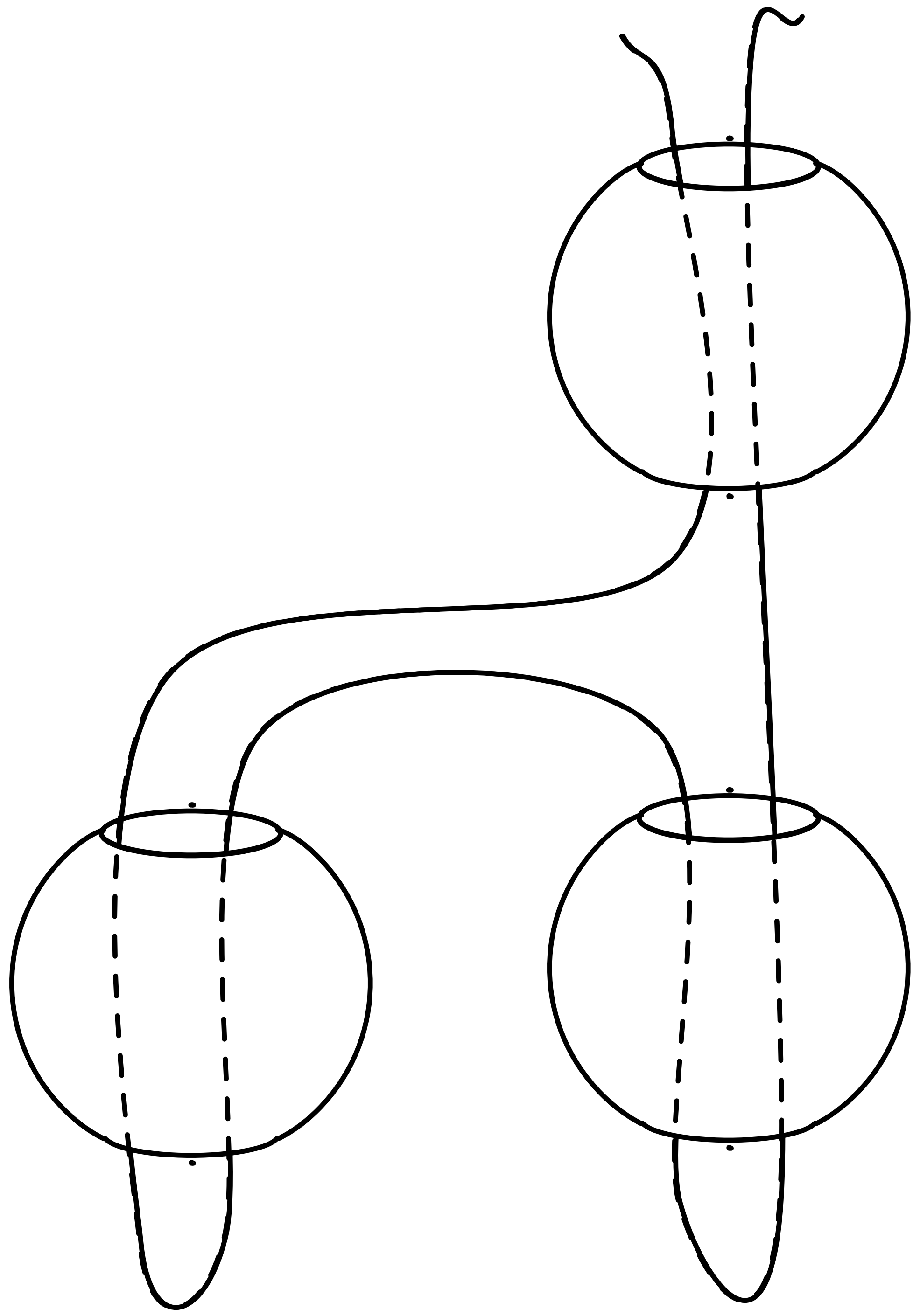
Hierarchies



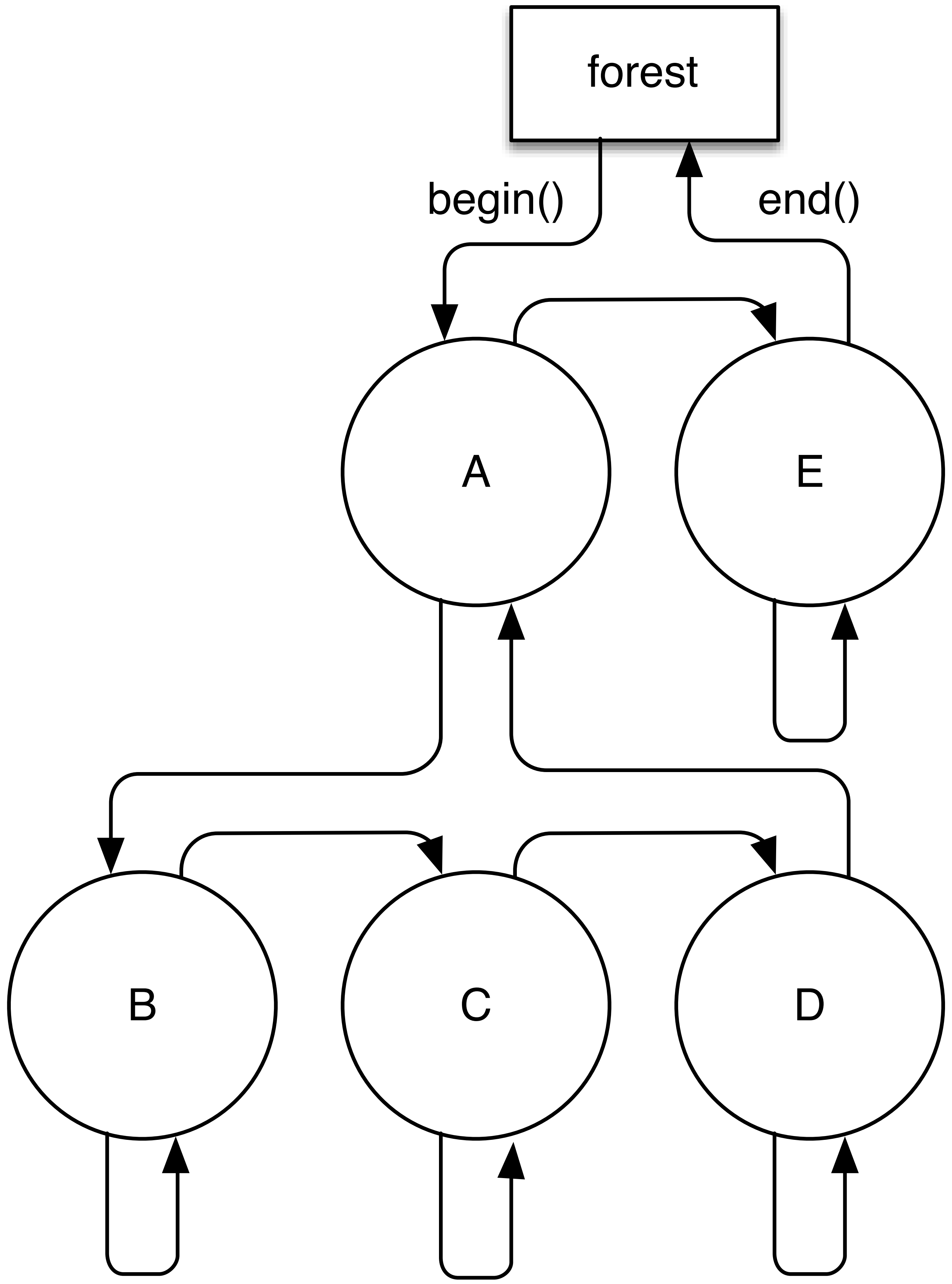
Hierarchies



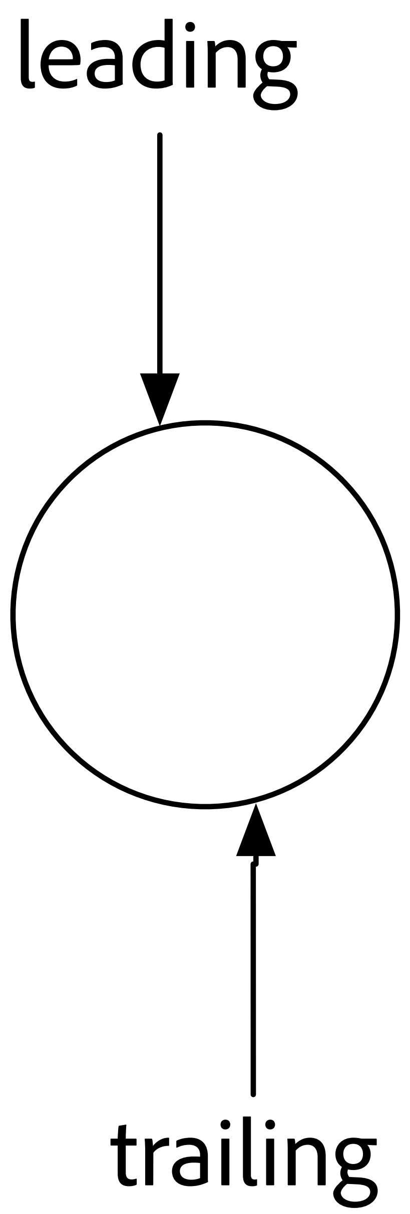
Hierarchies



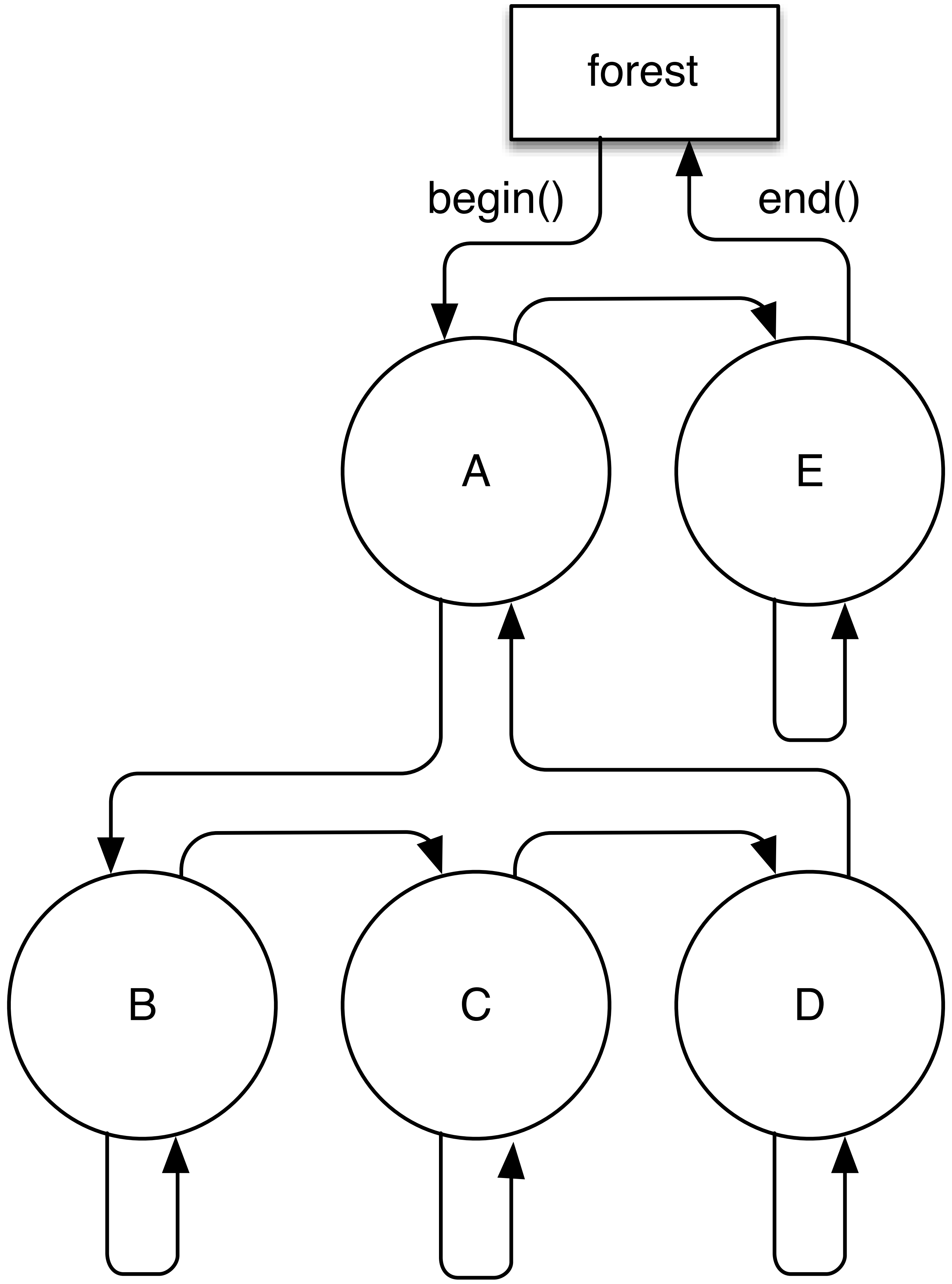
Hierarchies



Hierarchies

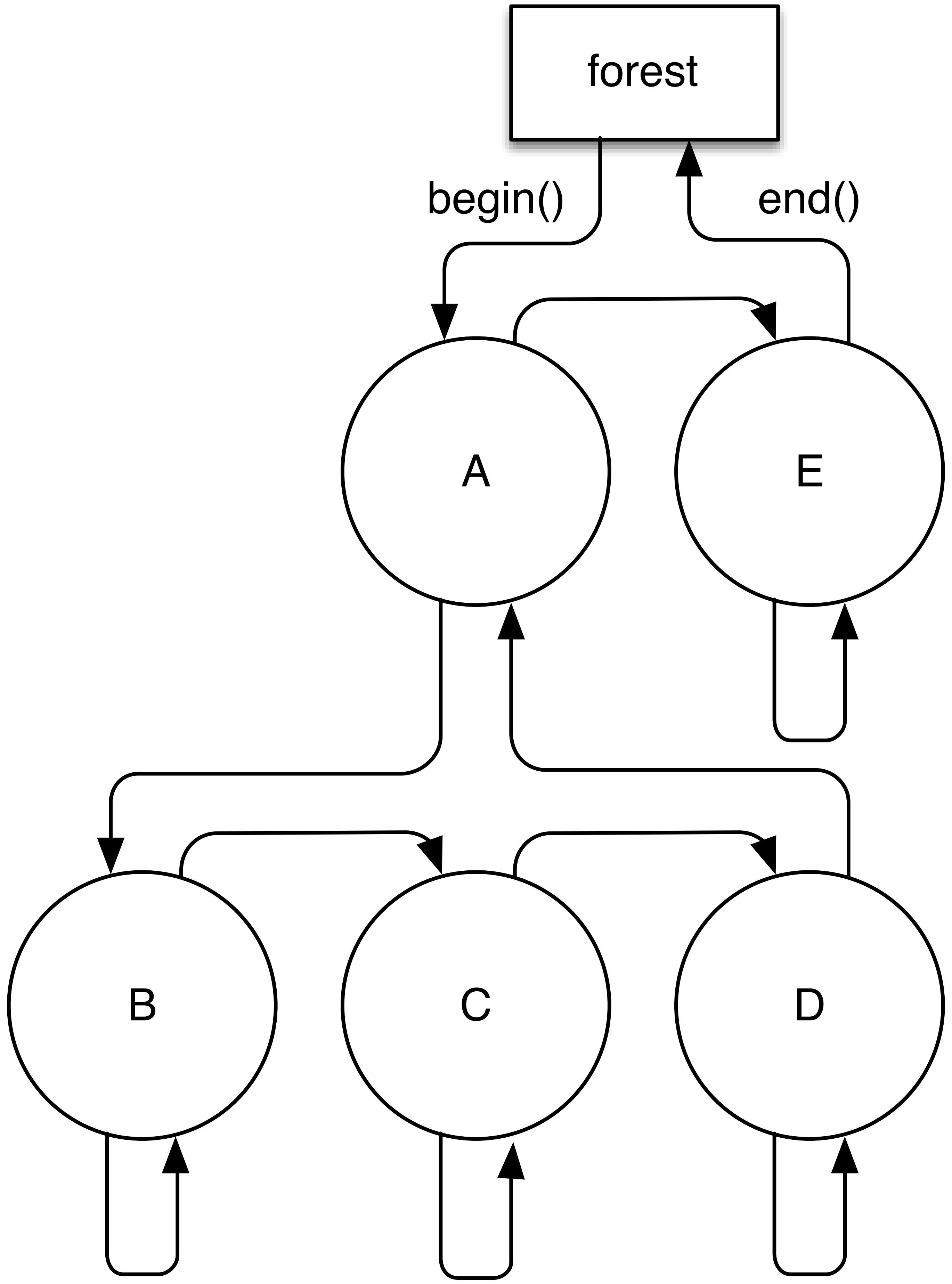


Hierarchies



Hierarchies

```
forest<string> f;

f.insert(end(f), "A");
f.insert(end(f), "E");

auto a = trailing_of(begin(f));
f.insert(a, "B");
f.insert(a, "C");
f.insert(a, "D");
```



Conclusions

- Understand the structures created by relationships
- Encapsulate structure invariants in composite types
- Learn to use the tools at your disposal
  - And how to create new ones

No incidental data structures

No incidental data structures

Composite Types

No incidental data structures

Composite Types

Better Code

Adobe