Better Code: Data Structures

Sean Parent | Principal Scientist

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?

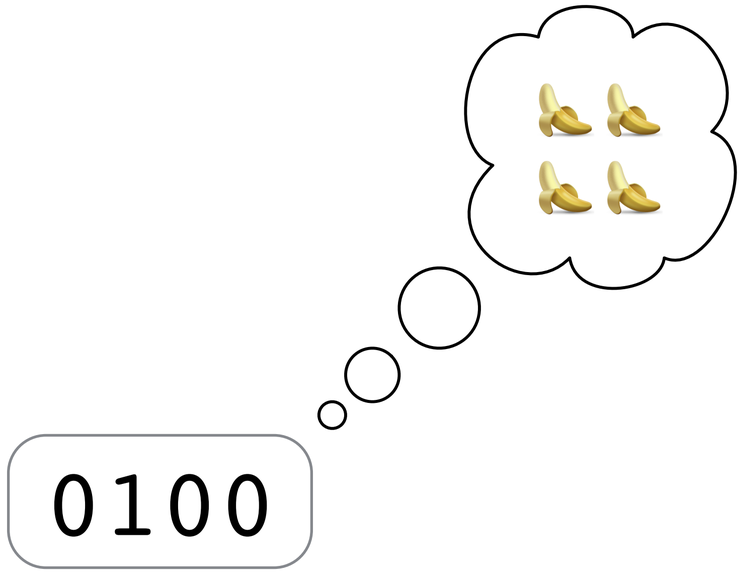
Definition: A data structure is a format for organizing and storing data.

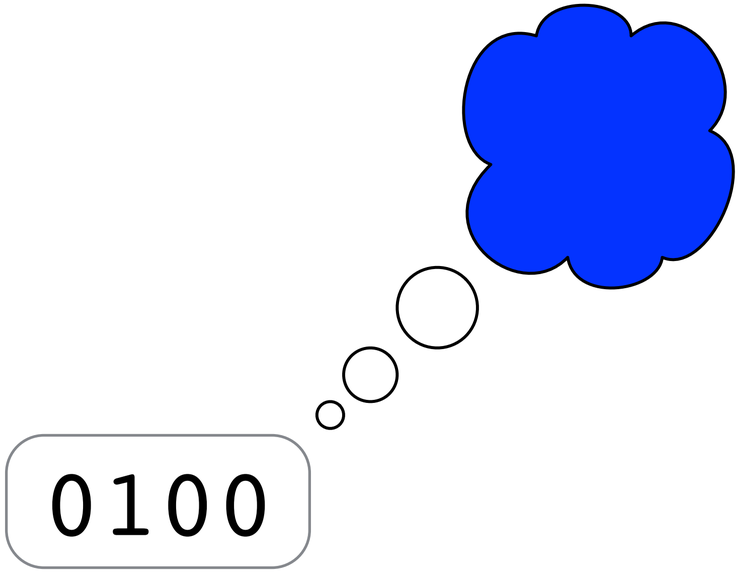
What is a structure?

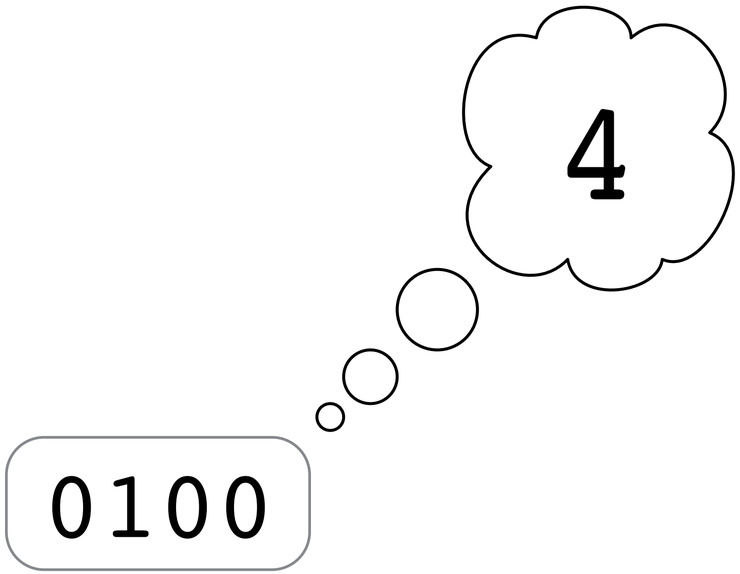
Definition: A structure on a set consists of additional entities that, in some manner, relate to the set, endowing the collection with meaning or significance.

[This slide intentionally left `void`]

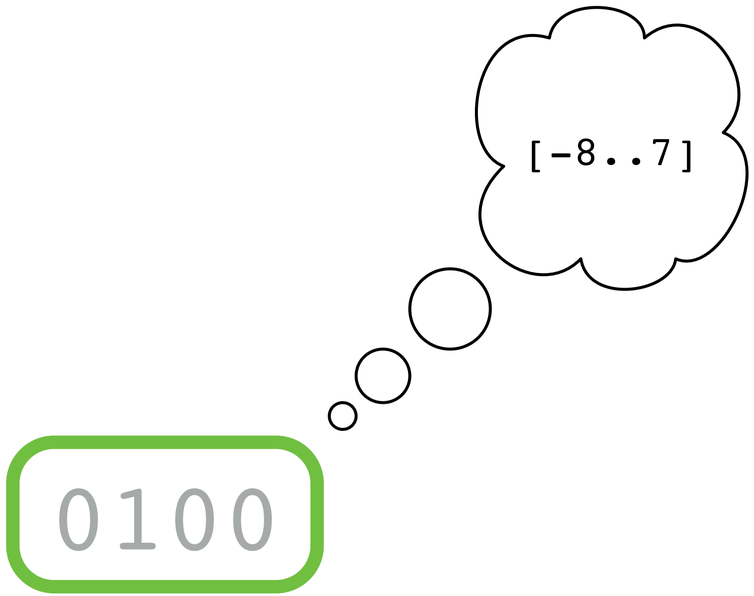
0100

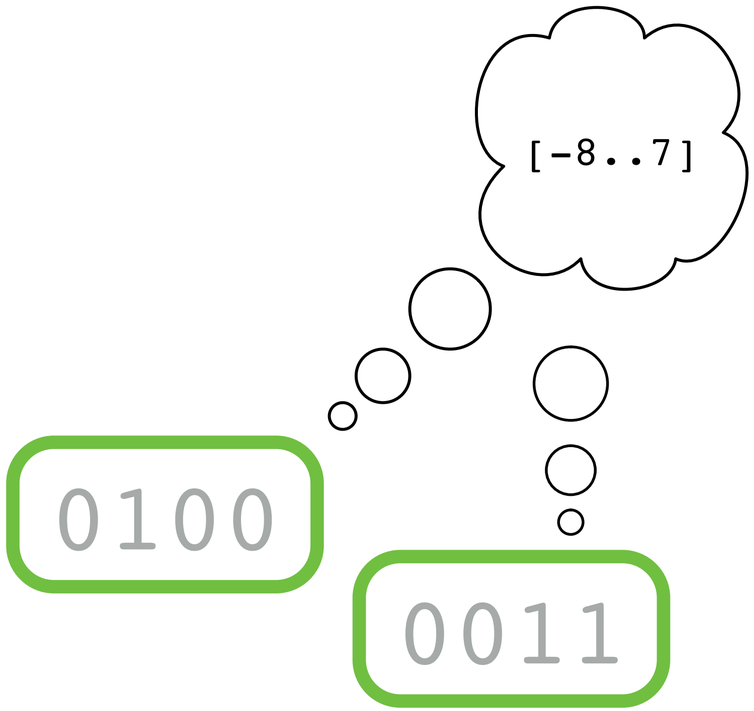

0100

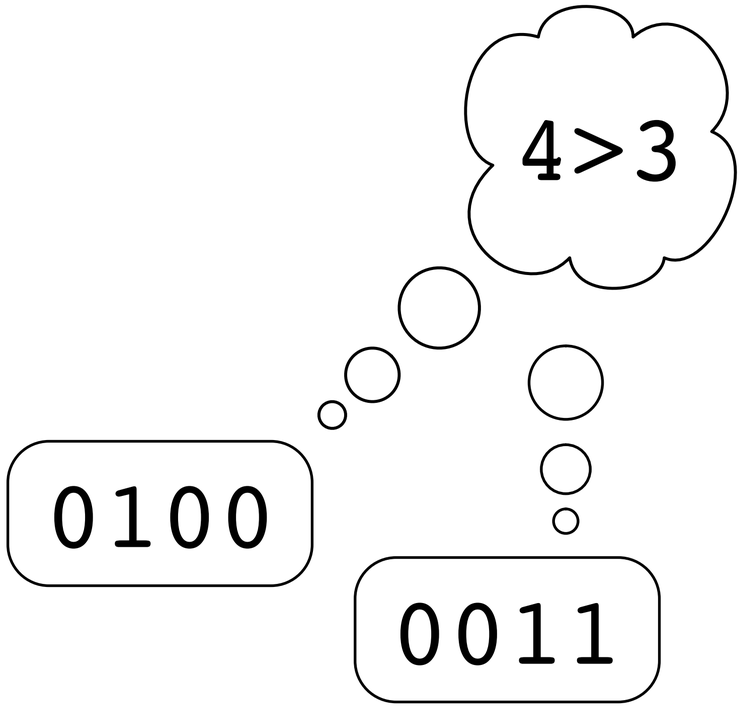

0100

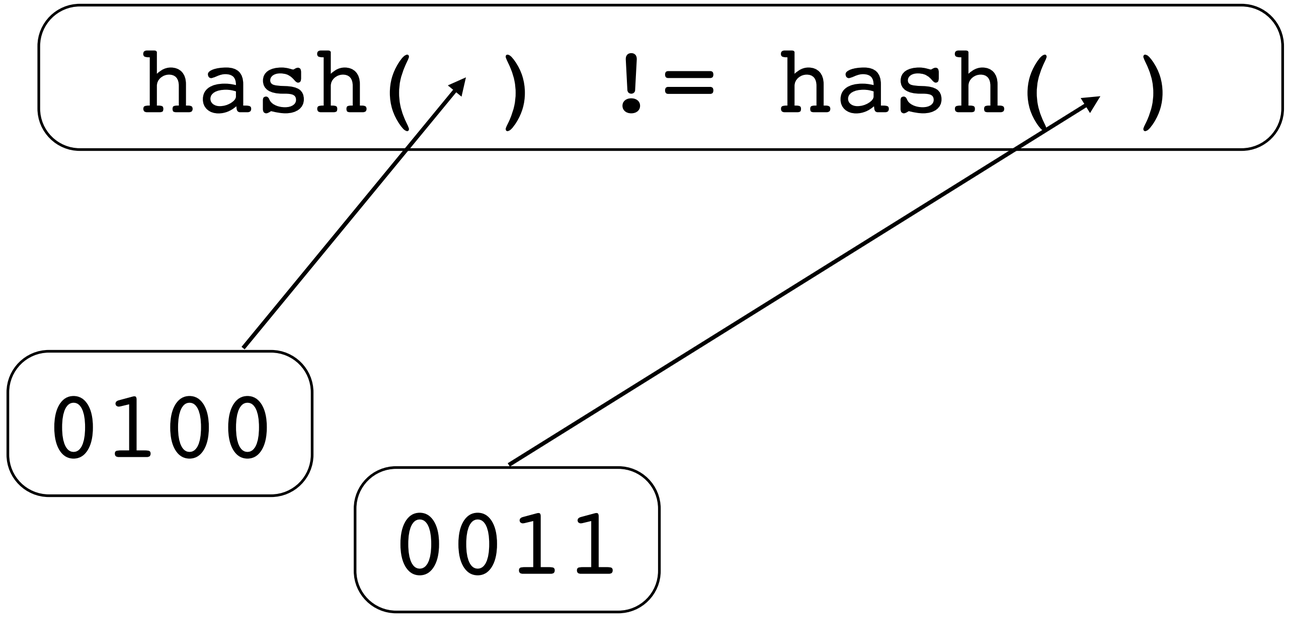

4
0100

0100

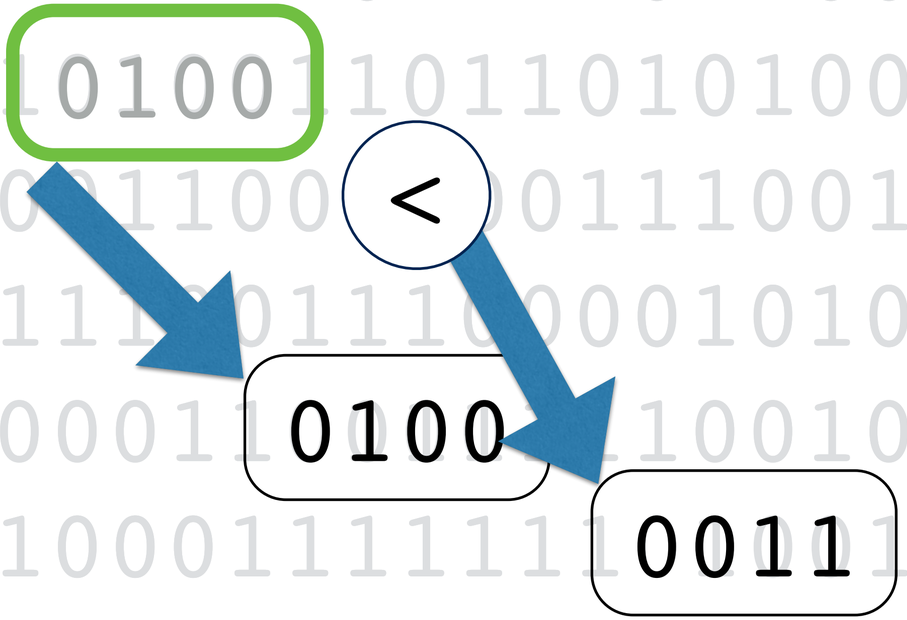

[-8..7]
0100


[-8..7]
0100
0011

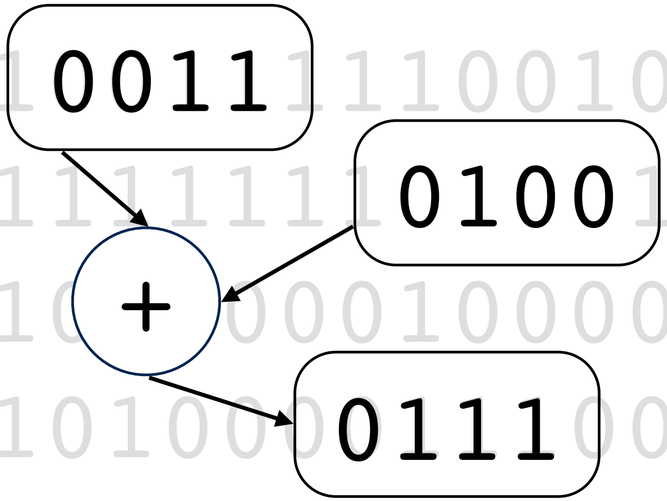

4>3
0100
0011


hash( ) != hash( )
0100
0011

Memory Space



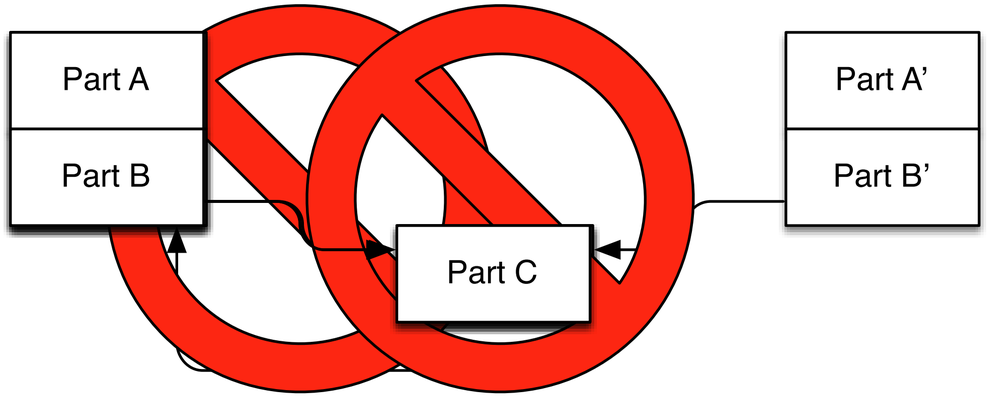
Memory Space



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?

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

- 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

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

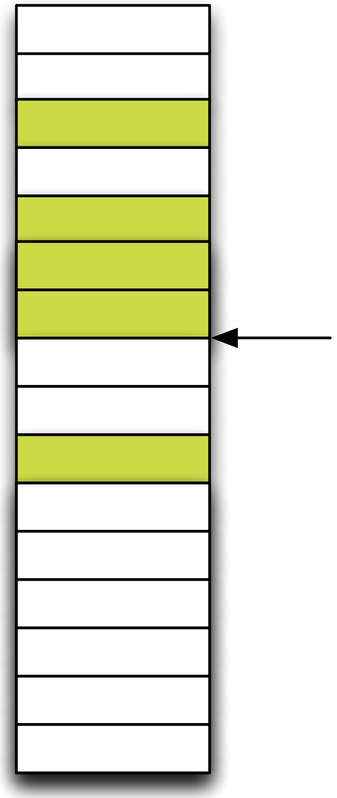
3GHz processor, from Chandler Carruth talk - Credit to Jeff Dean

Data Structure Performance

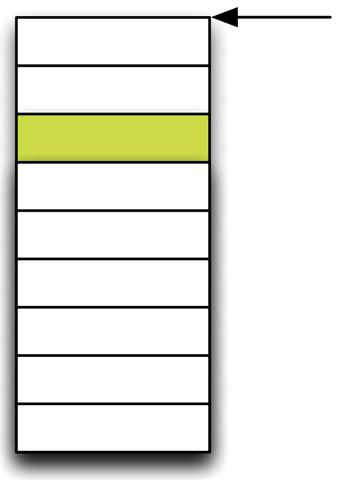
- Locality matters - use arrays or vector
  - Parallel Arrays
  - Static Lookup Tables
  - Closed Hash Maps
  - Algorithms

Example: Parallel Array & Algorithms

Gather

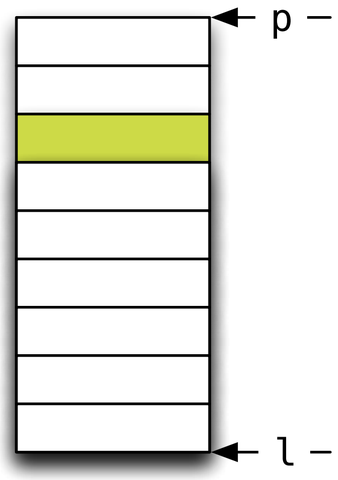


Gather

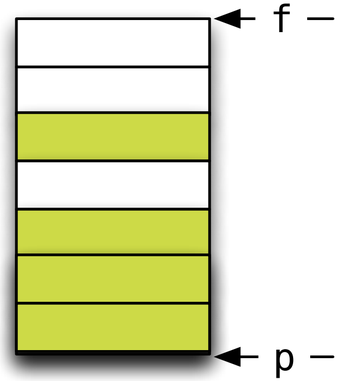


Gather

```
stable_partition(p, l, s)
```

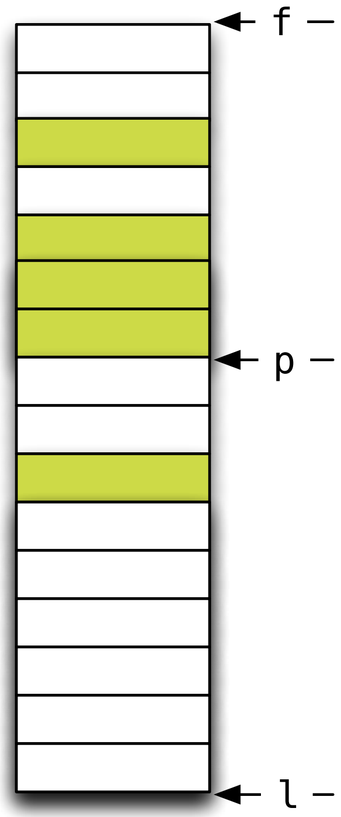


Gather



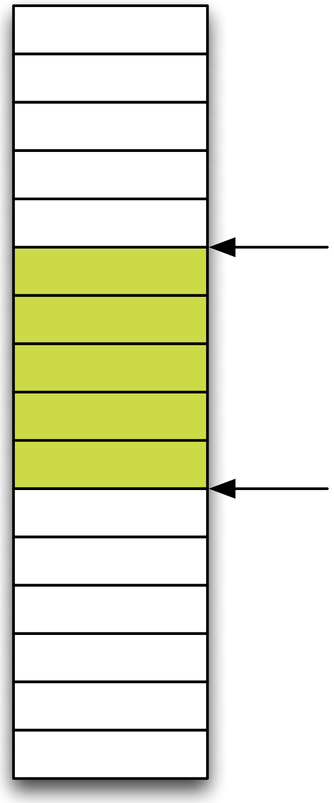

```
stable_partition(f, p, not1(s))
```

Gather



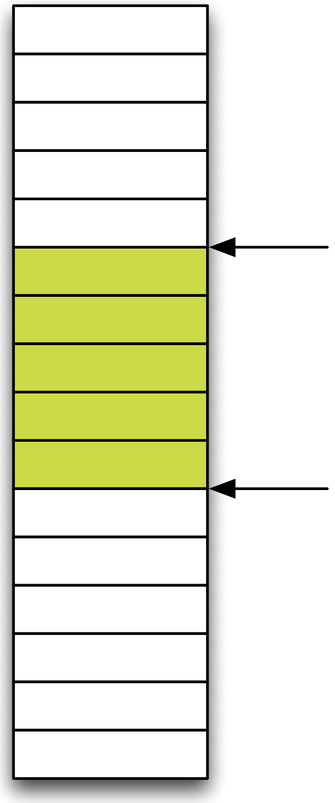

```
stable_partition(f, p, not1(s))
stable_partition(p, l, s)
```

Gather



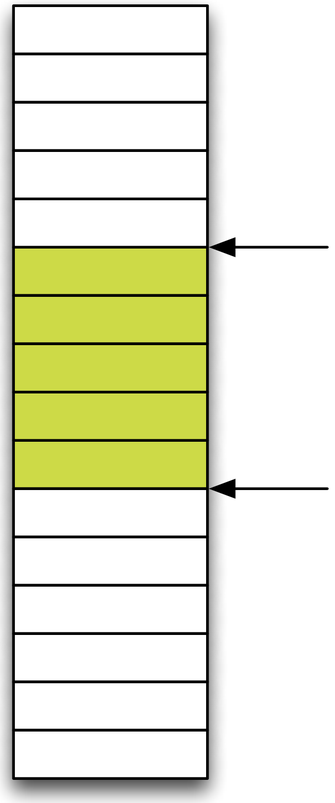
```
stable_partition(f, p, not1(s))
stable_partition(p, l, s)
```

Gather



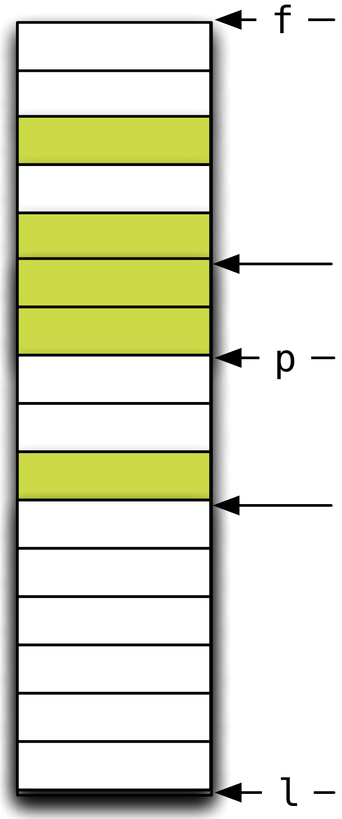
```
return { stable_partition(f, p, not1(s)),
         stable_partition(p, l, s) };
```

Gather



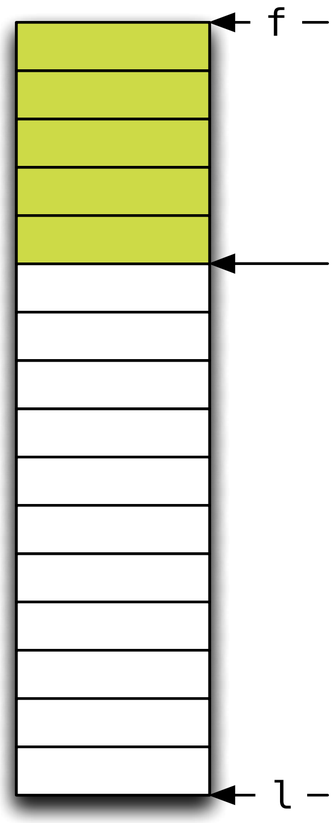
```
template <typename I, // I models BidirectionalIterator
          typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
}
```

Gather

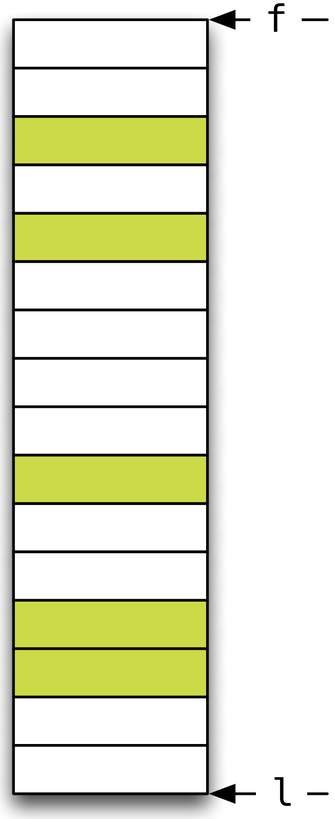



```
template <typename I, // I models BidirectionalIterator
          typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
}
```

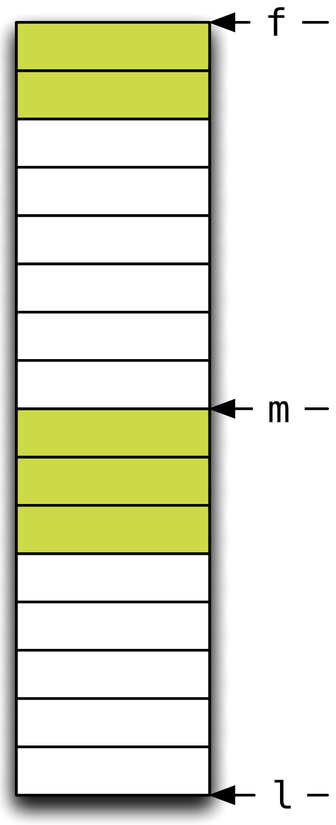
Stable Partition



Stable Partition



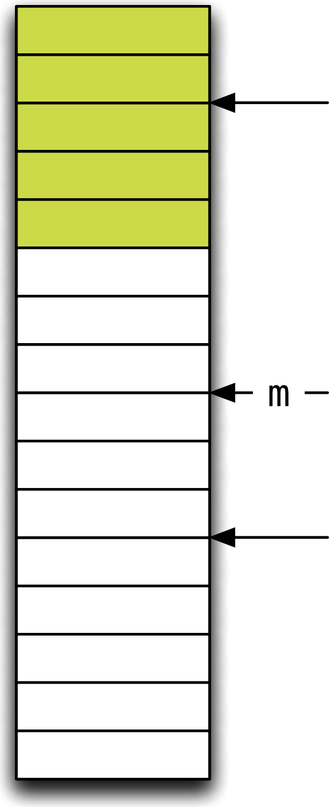
Stable Partition




```
stable_partition(f, m, p)

stable_partition(m, l, p)
```

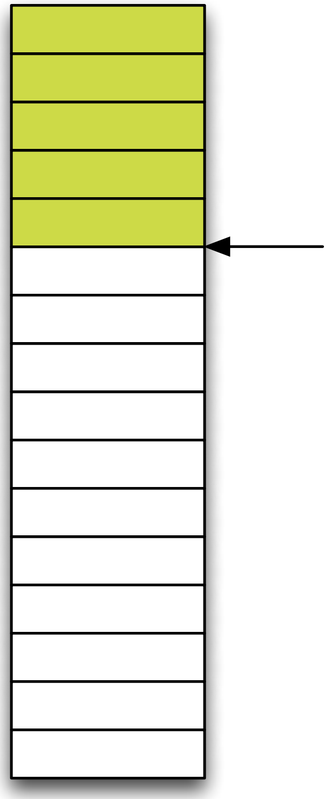

Adobe

Stable Partition



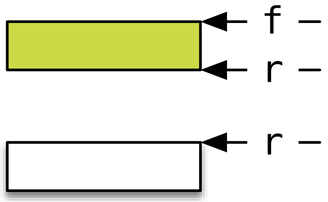

```
rotate(stable_partition(f, m, p),
       m,
       stable_partition(m, l, p));
```

Stable Partition



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

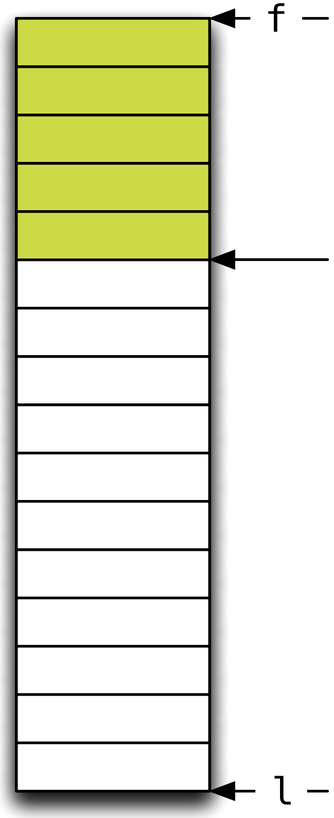
Stable Partition




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

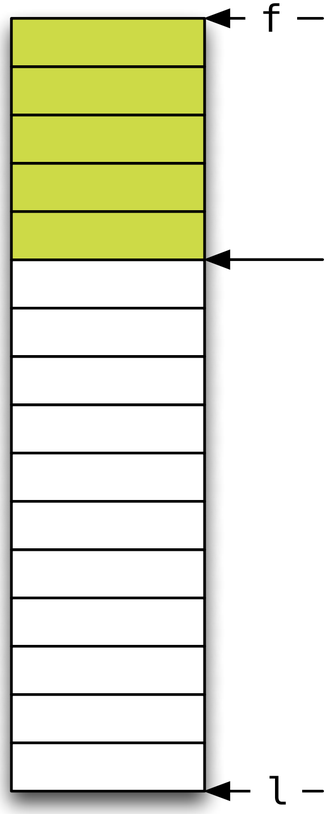
```
return rotate(stable_partition(f, m, p),
              m,
              stable_partition(m, l, p));
```

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));
}
```


Adobe

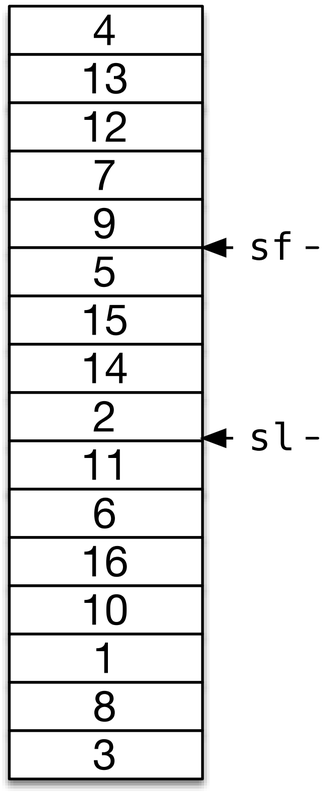
Stable Partition

```
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;
```

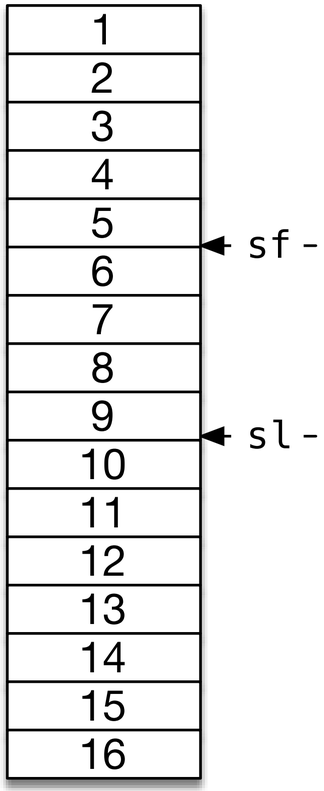
2 4 4 2 ^ 1 3 5 5 3 1

Example: Algorithms & Minimal Work

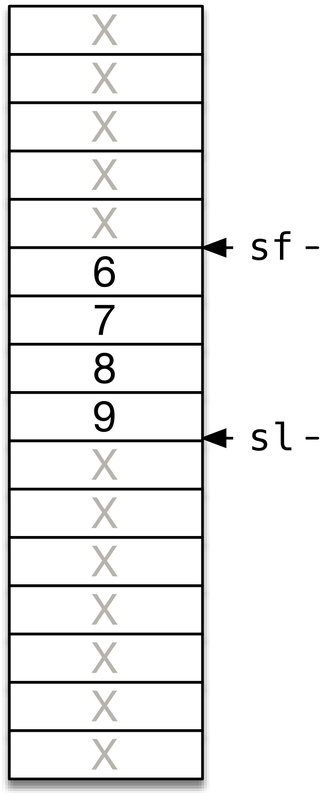
Minimize Work



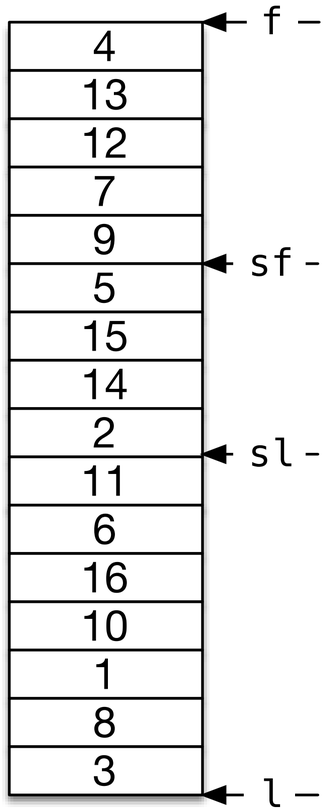
Minimize Work



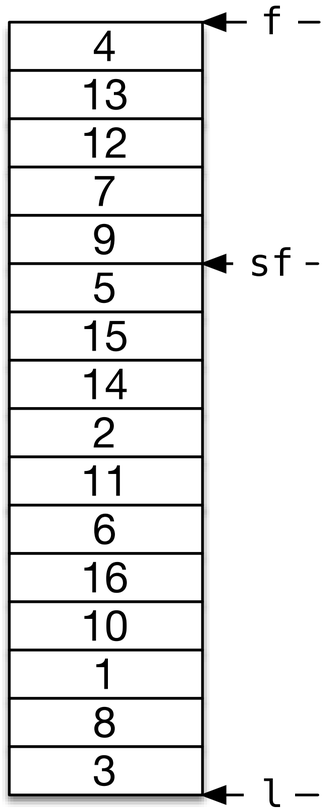
Minimize Work



Minimize Work

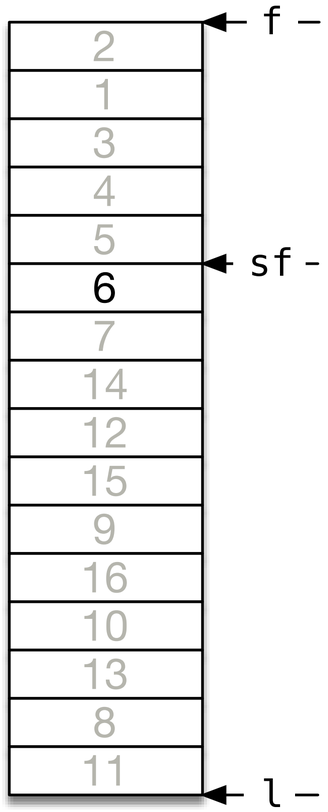


Minimize Work



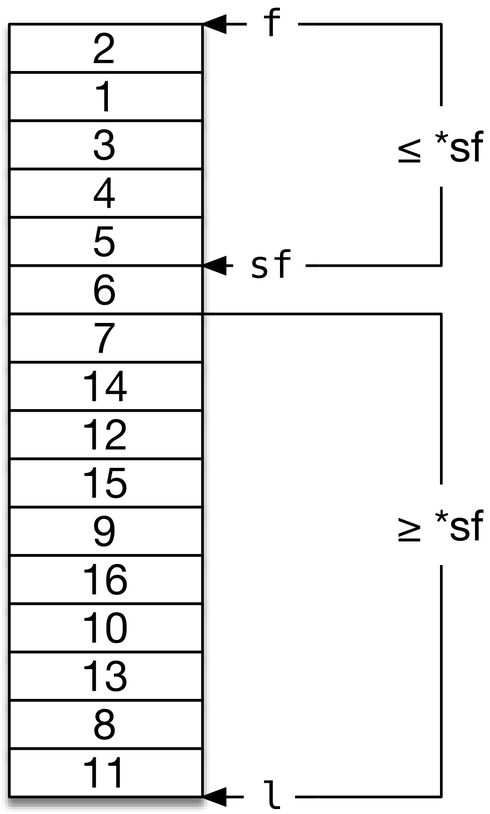

```
nth_element(f, sf, l);
```

Minimize Work



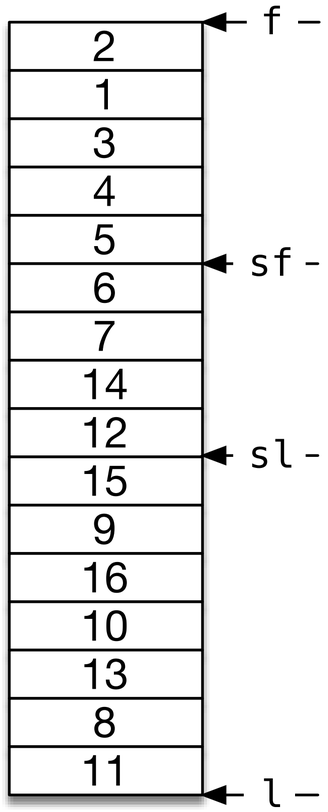

```
nth_element(f, sf, l);
```

Minimize Work



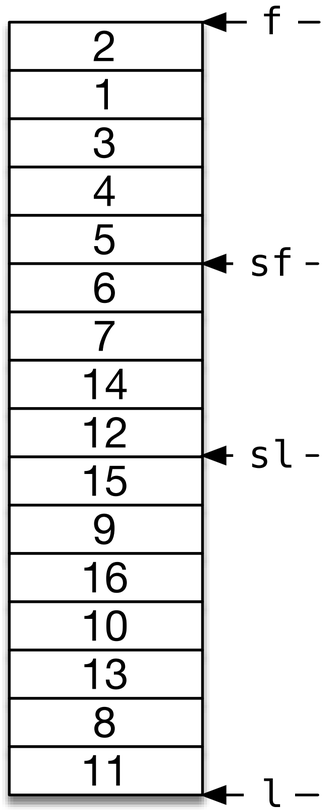

```
nth_element(f, sf, l);
```

Minimize Work



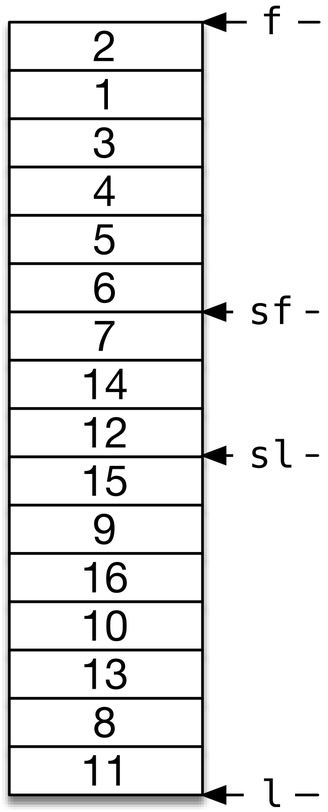

```
nth_element(f, sf, l);
```

Minimize Work



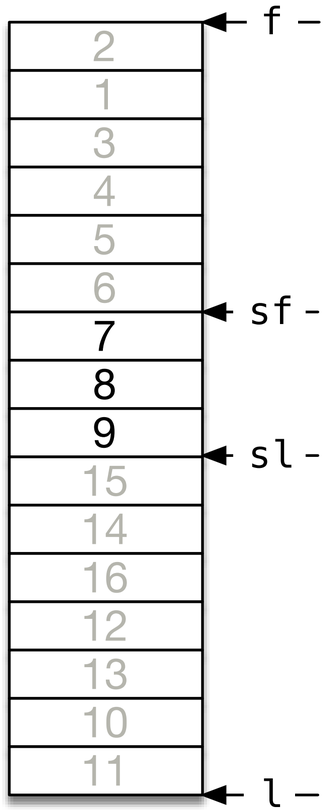

```
nth_element(f, sf, l);
++sf;
```

Minimize Work



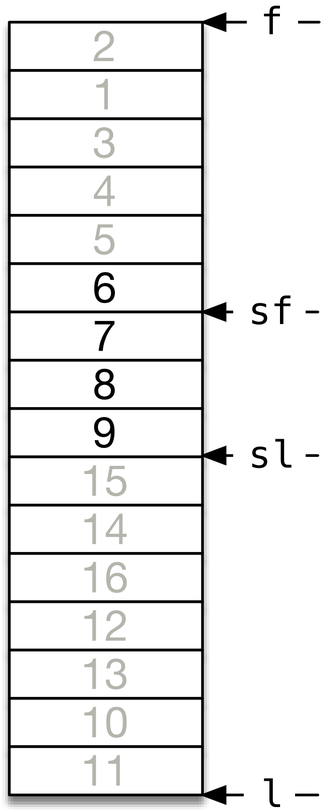

```
    nth_element(f, sf, l);
    ++sf;

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);
```

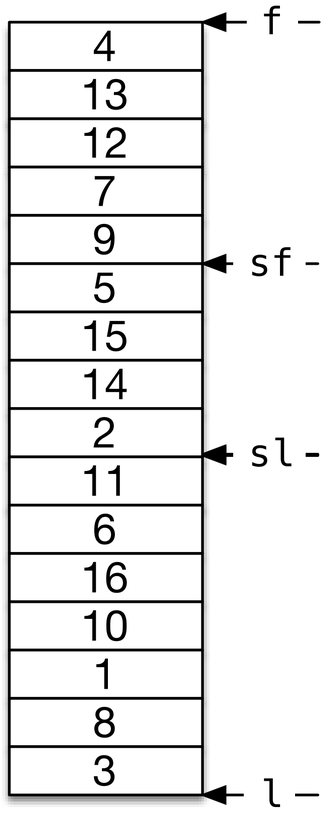
Minimize Work

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

Minimize Work

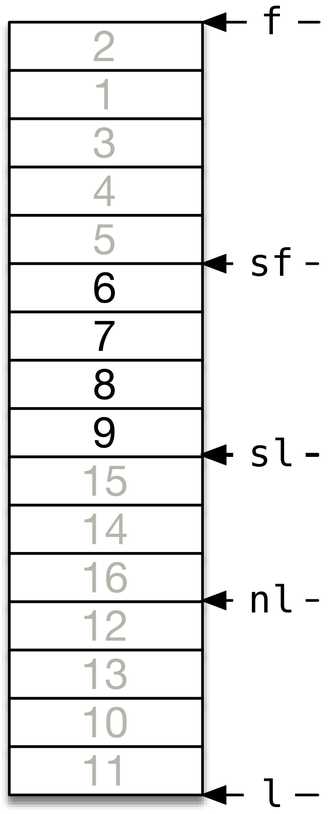
```
template <typename I> // I models RandomAccessIterator
void sort_subrange(I f, I l, I sf, I sl)
{
    if (sf == sl) return;
    if (sf != f) {
        nth_element(f, sf, l);
        ++sf;
    }
    partial_sort(sf, sl, l);
}
```

Minimize Work



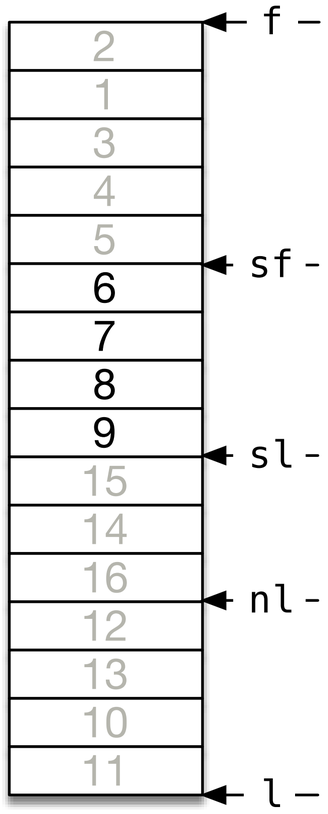

```
sort_subrange(f, l, sf, sl);
```

Minimize Work



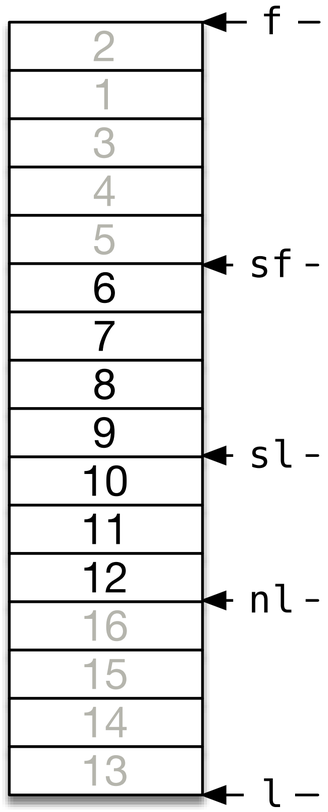

```
sort_subrange(f, l, sf, sl);
```

Minimize Work




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

Minimize Work




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

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

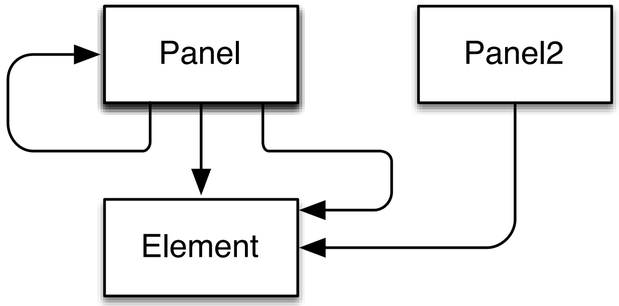
Incidental Data Structures

- Delegates
- Message handlers
- Any pointer or reference stored in an object which refers to another object which is not a part

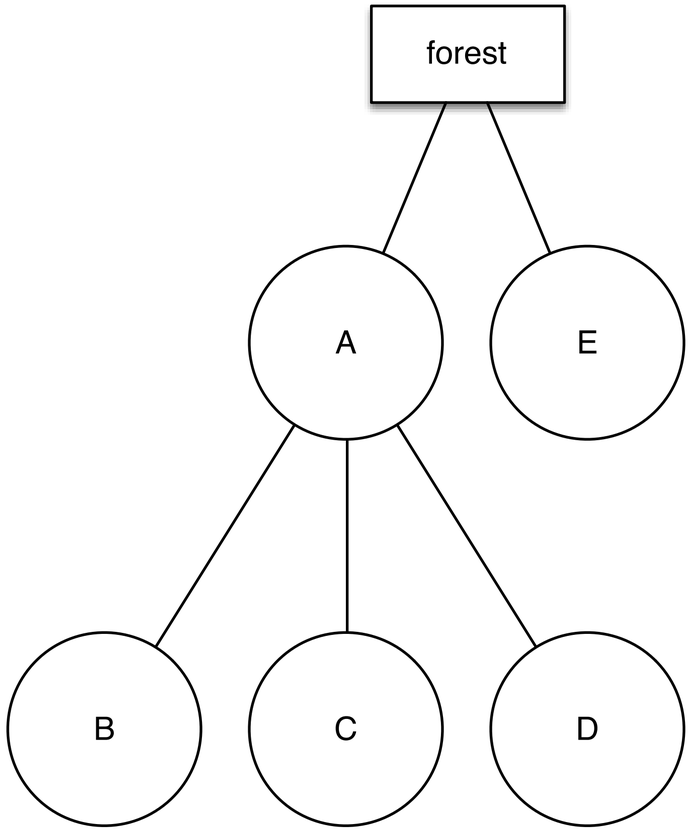
Incidental Data Structures

- Self-referential interface

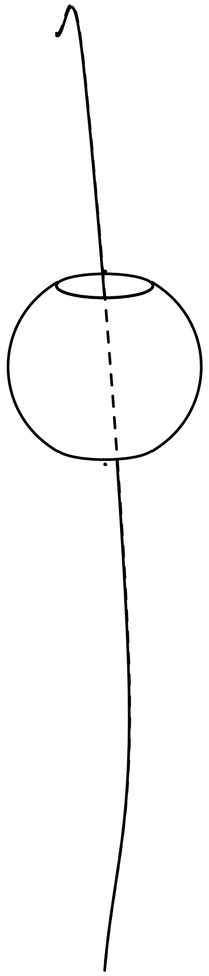
```
class UIElement { };

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

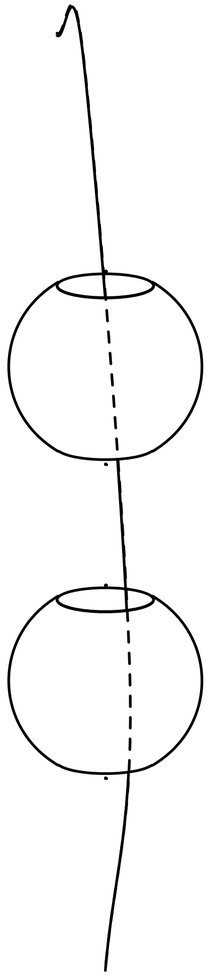


Hierarchies

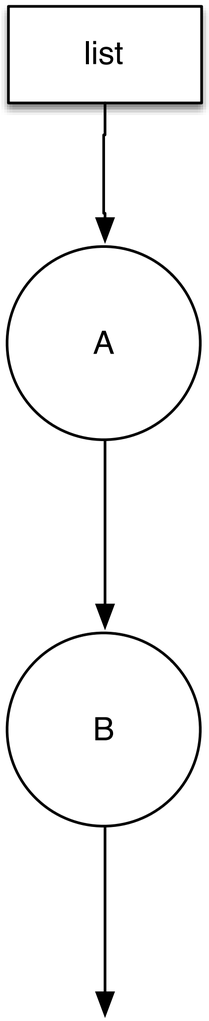


Adobe

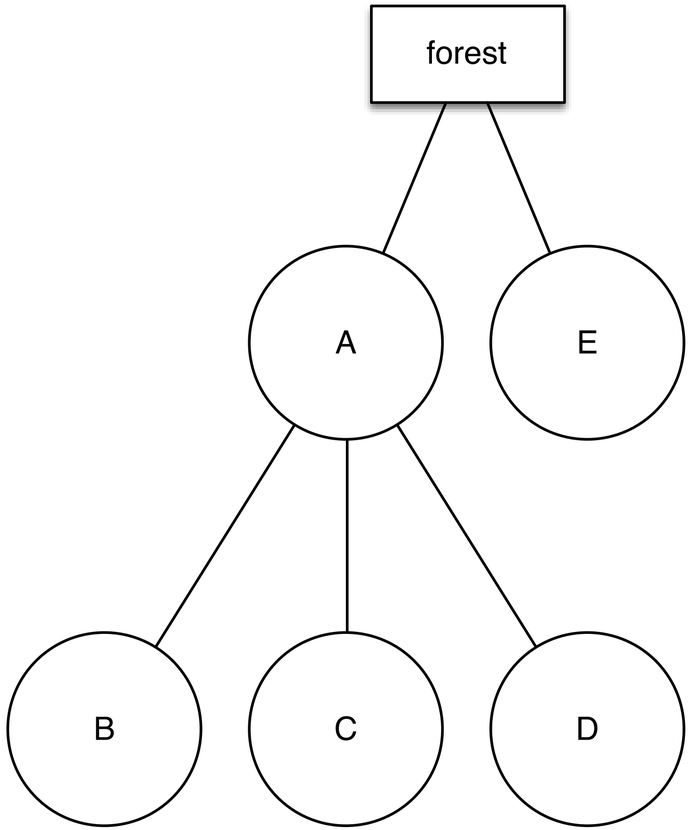
Hierarchies



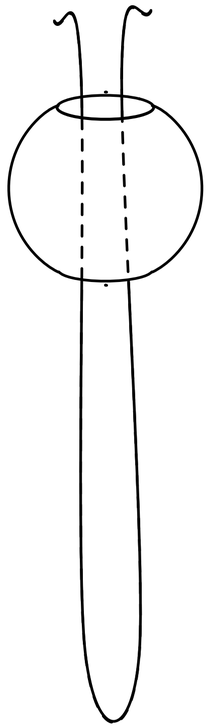
Hierarchies



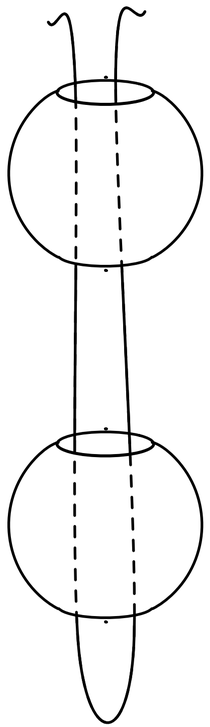
Hierarchies



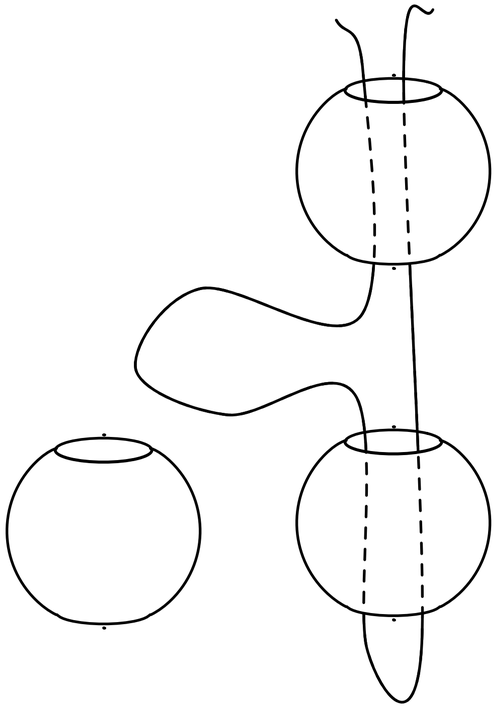
Hierarchies



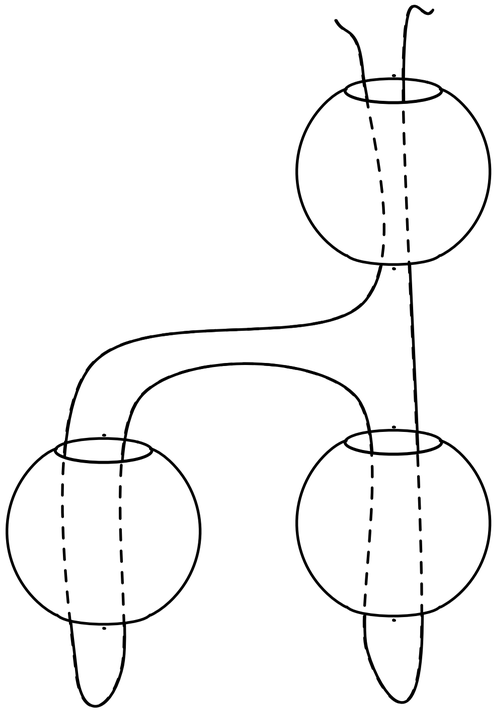
Hierarchies



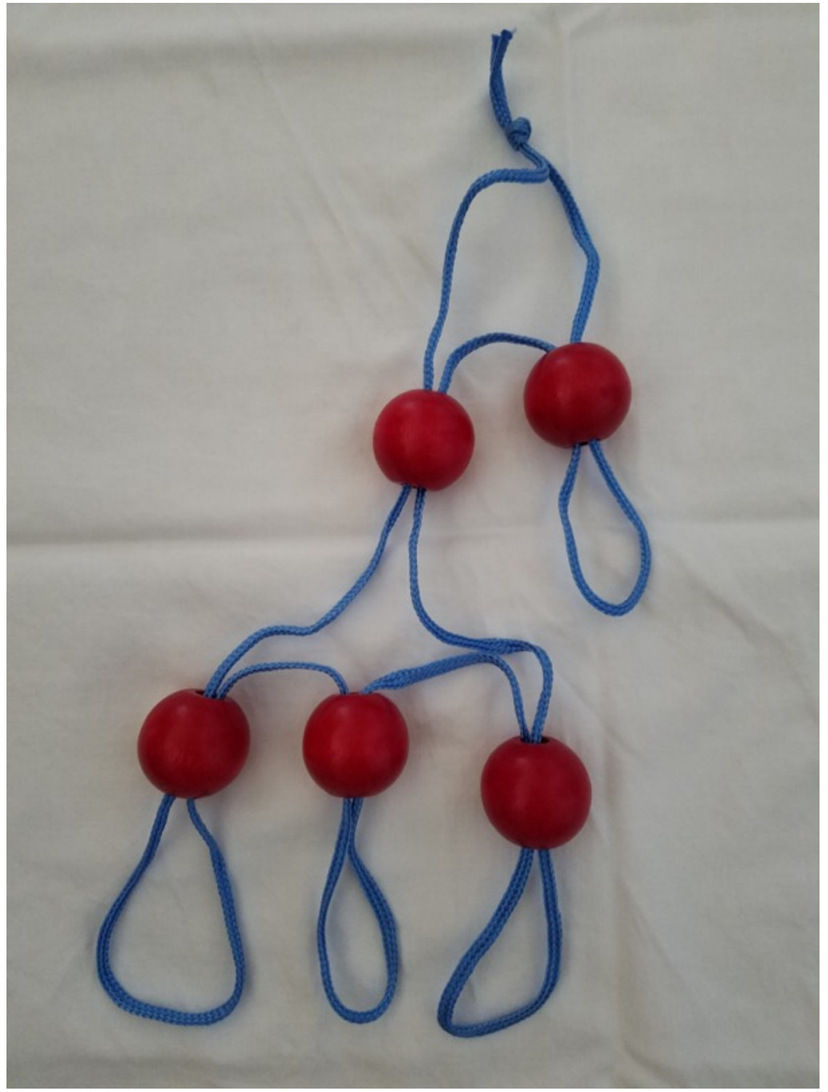
Hierarchies



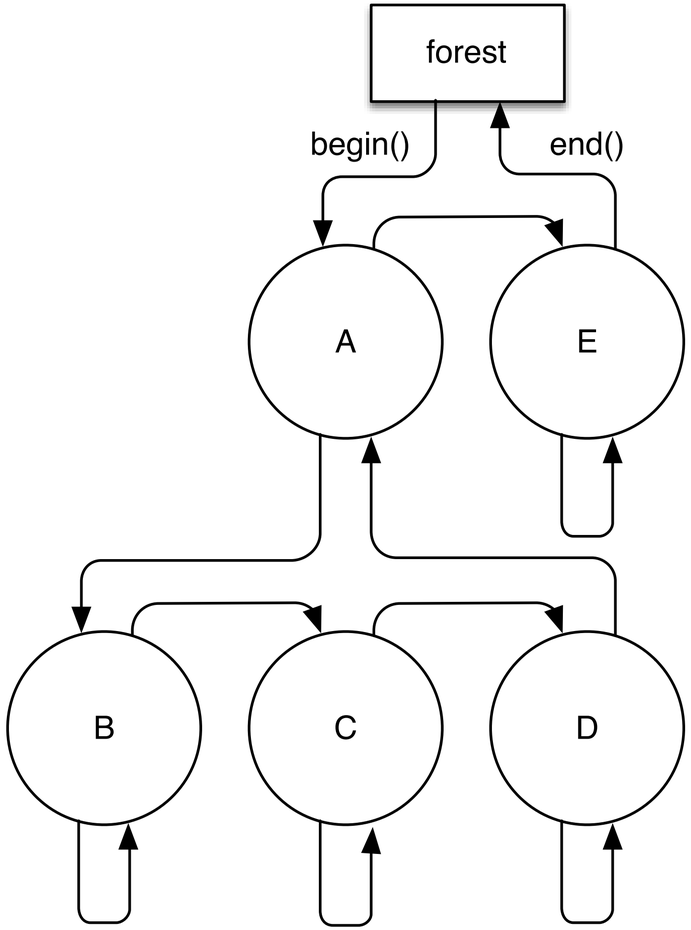
Hierarchies



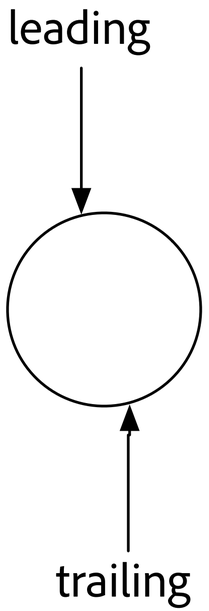
Hierarchies



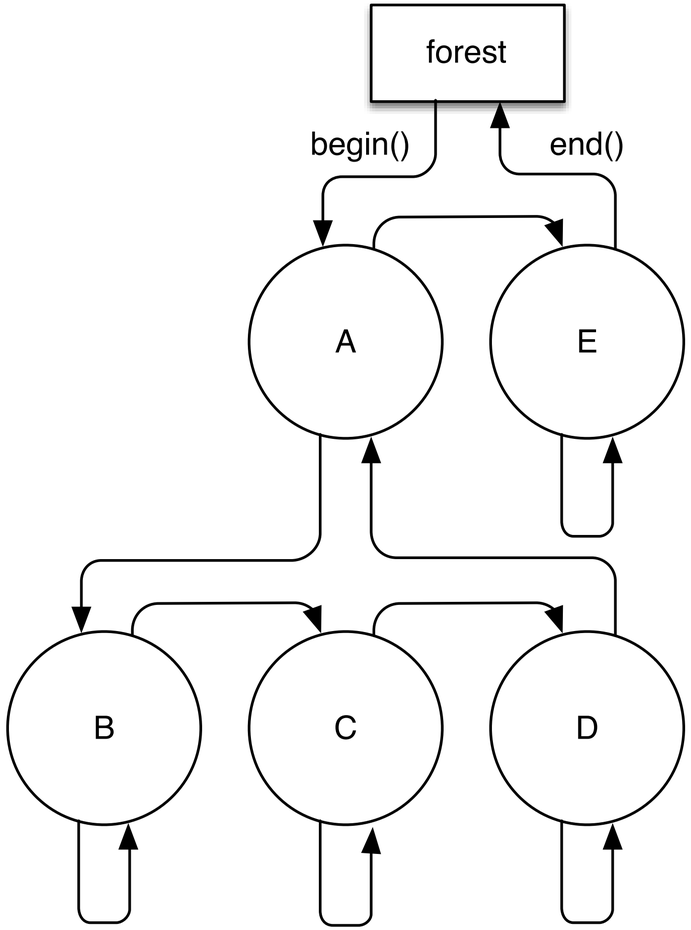
Hierarchies



Hierarchies

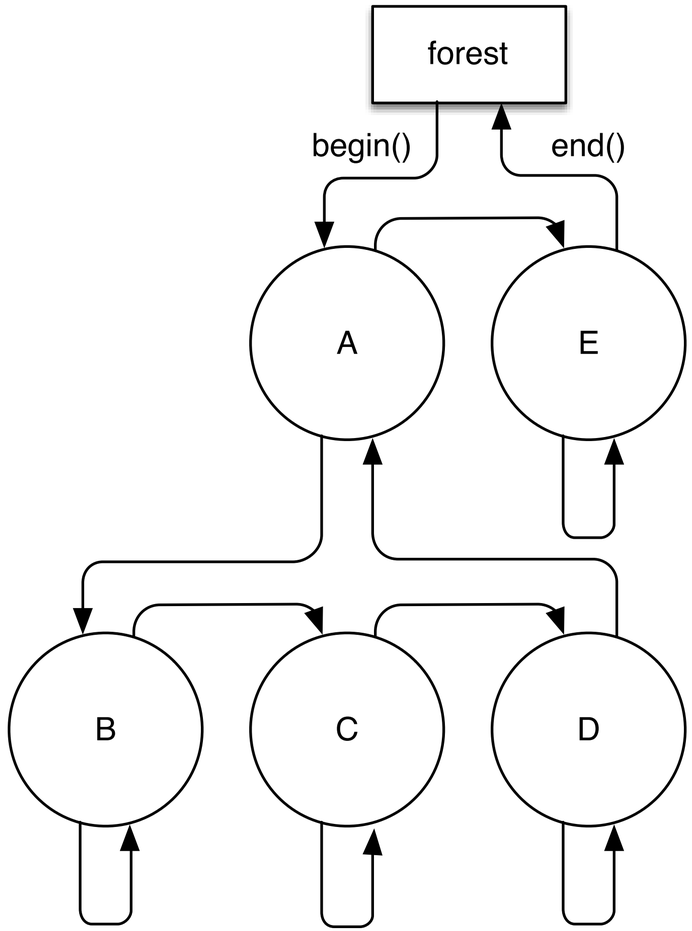


Hierarchies



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");
```



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");

auto r = depth_range(f);
for (auto f = begin(r), l = end(r); f != l; ++f) {
    cout << string(f.depth() * 4, ' ') << (f.edge() ? "<" : "</") << *f << “>\n";
}
```

```
<A>
    <B>
    </B>
    <C>
    </C>
    <D>
    </D>
</A>
<E>
</E>
```

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

- Slides and code from talk:
- https://github.com/sean-parent/sean-parent.github.io/wiki/Papers-and-Presentations

- Forest library:
- https://github.com/stlab/adobe_source_libraries

No incidental data structures

Composite Types

Better Code


Adobe