Separation Logic

Zhaoguo Wang

Adapted From:

SMT-based Verification of Heap-manipulating Programs, Yale

Malloc & Free

malloc(size)

- Allocate size bytes of memory
- Return a pointer to the allocated memory

free(ptr)

- Release the memory pointed by ptr

```
int *p = malloc(sizeof(int));

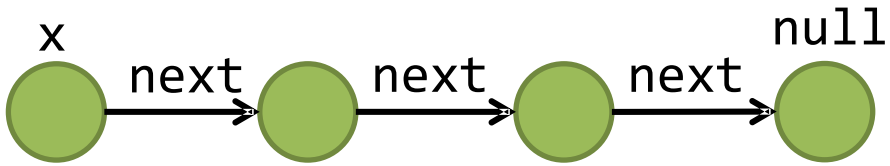
*p = 10;

free(p);
```

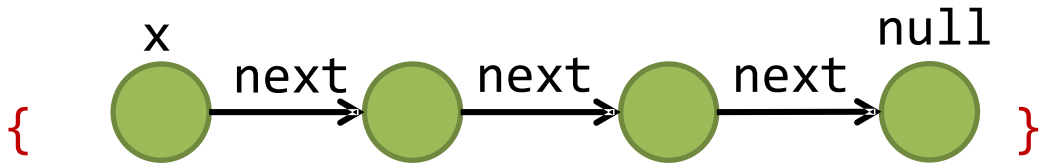
Program state changes dynamically, which is hard to be formalized

A Motivating Example

```
procedure delete(x: Node)
{
   if (x != null) {
      delete(x.next);
      free(x);
   }
}
```



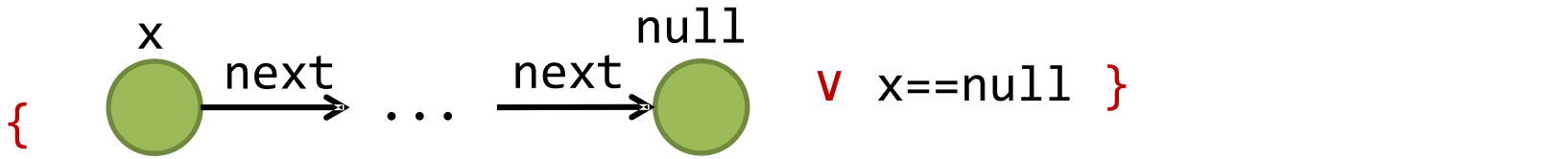
A Motivating Example



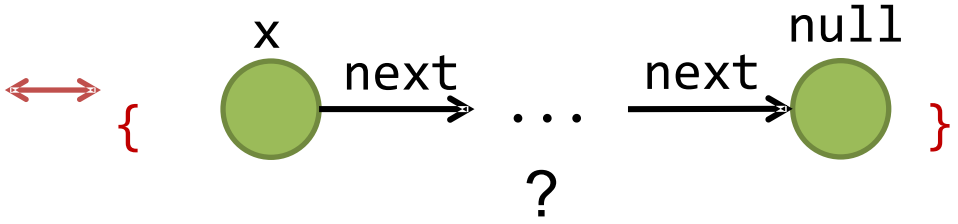

```
procedure delete(x: Node)
{
   if (x != null) {
      delete(x.next);
      free(x);
   }
}
```

{ }

A Motivating Example

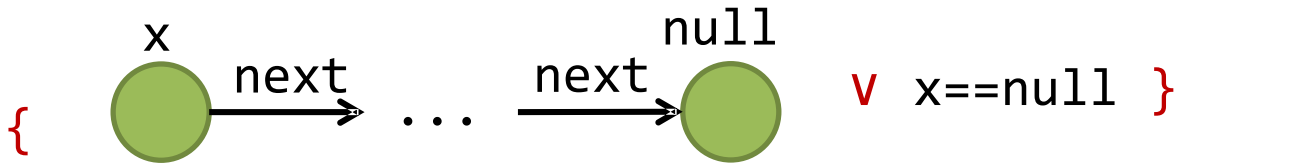



```
procedure delete(x: Node)
{
   if (x != null) {
      delete(x.next);
      free(x);
   }
}
```

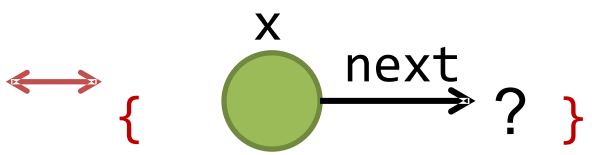


{ }

A Motivating Example

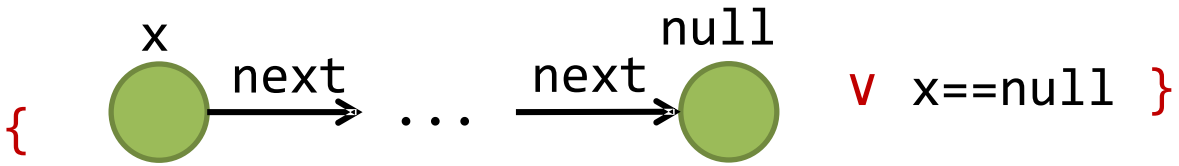



```
procedure delete(x: Node)
{
   if (x != null) {
      delete(x.next);
      free(x);
   }
}
```



{ }

A Motivating Example




```
procedure delete(x: Node)
{
   if (x != null) {
      delete(x.next);
      free(x);
   }
}
```



{ }

Heap

An allocator tracks used and free blocks, including their sizes.

malloc(size) selects a free block and may split a larger block.

free(ptr) mark the blocks to be free again

Example: splitting a free block to satisfy a request

Before



After malloc(32)



Separation Logic（分离逻辑）

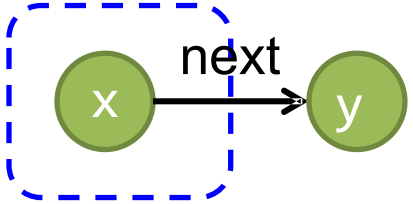
[Reynolds, O'Hearn, Yang, 2001]

- Propositions used to describe heap
  - emp : the allocated heap is empty
  - acc(x) : *Exactly* one single node pointed by x is allocated in heap
  - F * G : the allocated heap can be divided into 2 disjoint partitions

    One partition satisfies F, another partition satisfies G
  - The precondition must explicitly specify what heap has been allocated

    Heap that is not allocated should not be accessed by program
  - Other propositions (as in Hoare logic)

acc(x) * x.next = y

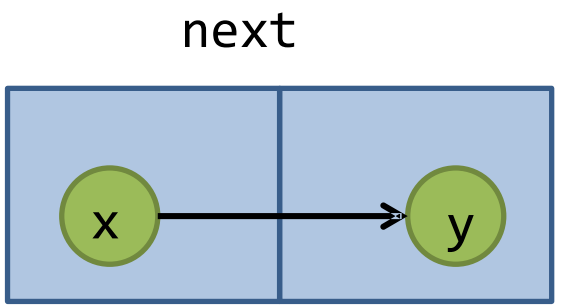


Example

Formalize the following with separation logic formulas

- There are two different nodes pointed by x and y
- The ”next” field of one node points to another node

```
acc(x) * ( acc(y) /\ x.next == y )
```



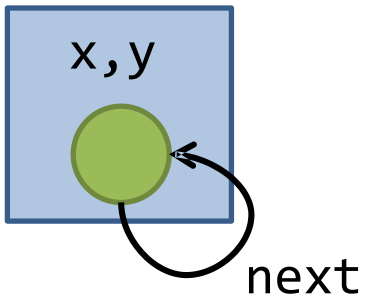
∧ has higher precedence than *

Example

Formalize the following with separation logic formulas

- There is one node pointed by both x and y
- The ”next” field of the node points to itself

$$\mathbf{acc}(x) \wedge \mathbf{acc}(y) \wedge x.\mathtt{next} == y$$



∧ has higher precedence than *

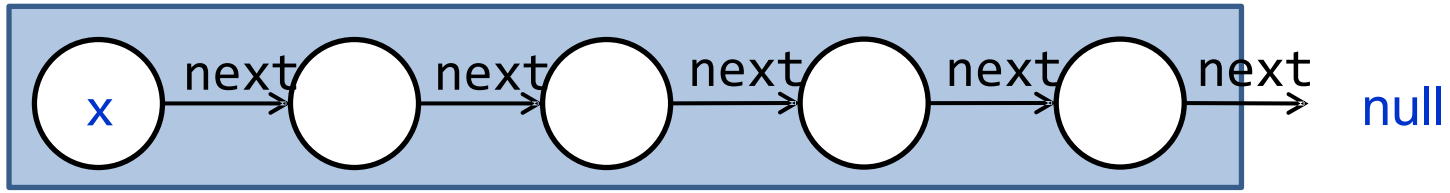
Formalization of a linked list

Formalize a linked list with a recursive predicate lseg(x) in separation logic

There are two cases

- x is null and the linked list is empty
- x is not null, x points to the head node of the list, x.next points to tail list

$$
\begin{aligned}
&\mathtt{lseg(x)} \equiv \\
&\quad \mathtt{(x == null)} \vee \\
&\quad \mathtt{((x \neq null} \wedge \mathbf{acc}\mathtt{(x))} * \mathtt{lseg(x.next))}
\end{aligned}
$$



Validity of Hoare Triples

Proving {P} C {Q} in separation logic ensures memory safety

- Memory safety means C only access allocated memory

Common violation of memory safety

- Out-of-bounds access
- Use-after-free
- Double-free

Some Examples of Hoare Triples

- `{acc(x)} x.next := y; {acc(x) /\ x.next == y}` 
- `{acc(y)} x.next := y; {acc(y) /\ x.next == y}` 

Precondition does not say x=y
x may point to a non-allocated memory
The program should not access non-allocated memory

Some Examples of Hoare Triples

- `{acc(x)} x.next := y; {acc(x) /\ x.next == y}` 
- `{acc(y)} x.next := y; {acc(y) /\ x.next == y}` 
- `{emp} x := new Node; {acc(x)}` 
- `{emp} free(x); {emp}` 

x does not point to an allocated memory
The program should not free a non-allocated memory

Some Examples of Hoare Triples

- {acc(x)} x.next := y; {acc(x) /\ x.next == y} 
- {acc(y)} x.next := y; {acc(y) /\ x.next == y} 
- {emp} x := new Node; {acc(x)} 
- {emp} free(x); {emp} 
- {acc(x)} free(x); {emp} 
- {acc(x) * (acc(y) /\ y.next == z)} 

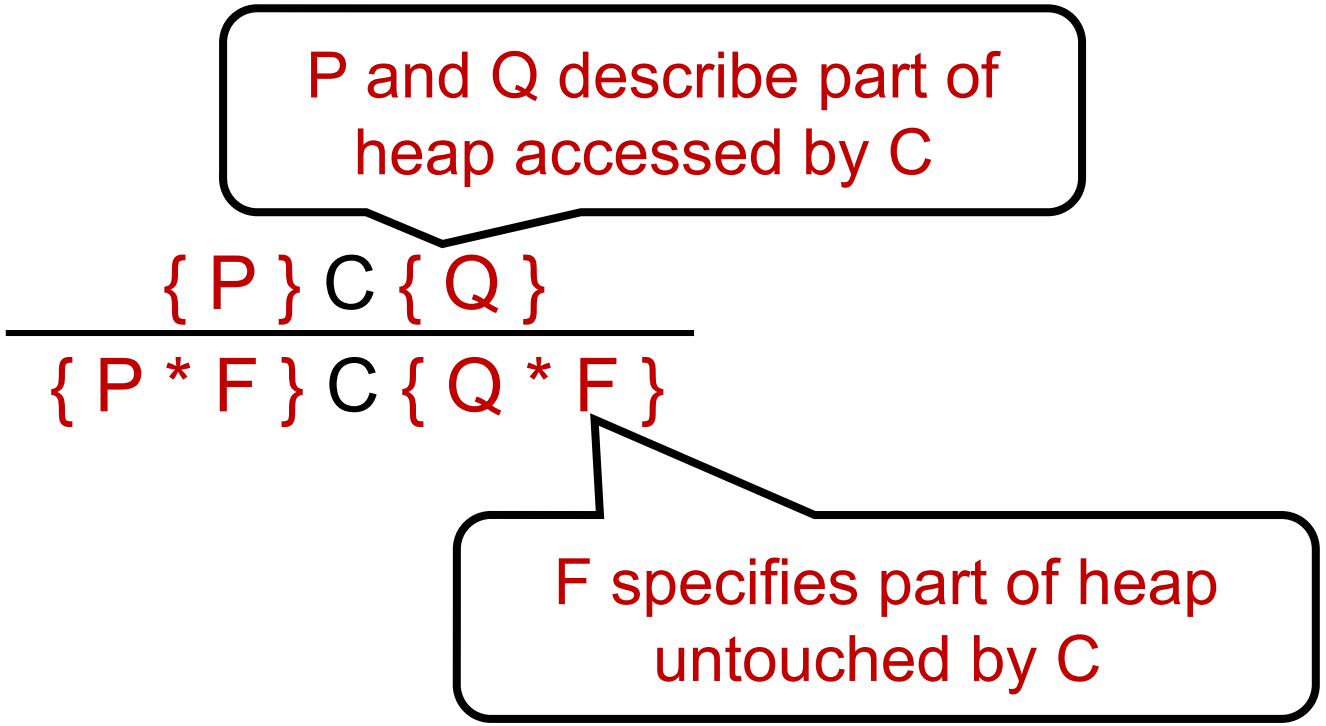
  x.next := y;

  {(acc(x) /\ x.next == y) * (acc(y) /\ y.next == z)}

A New Inference Rule: Frame Rule

Basic idea

- If code only accesses a part of allocated heap, the other part of allocated heap should stay unchanged



Inference Rule: Frame Rule

Basic idea

- If a **function invocation** only accesses a part of allocated heap, the other part of allocated heap should stay unchanged

If allocated heap satisfies P, then the heap consists of two partitions. One partition satisfies P' and another partition satisfies F

Function call only access allocated heap specified by P'

$$\frac{P \Rightarrow P' * F \qquad \{ P' \}\ \text{func}\ \{ Q' \}}{\{ P \}\ \text{func}\ \{ Q' * F \}}$$

Heap specifies by F is unchanged

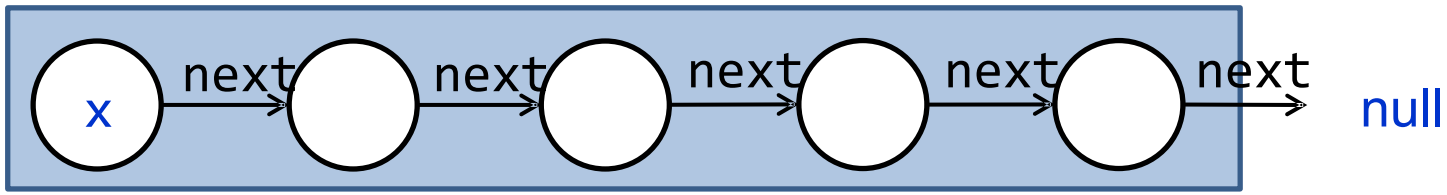
Specification of delete

```
{ lseg(x) }

procedure delete(x: Node)
{
   if (x != null) {
      delete(x.next);
      free(x);
   }
}

{ emp }
```

$$\begin{aligned} \mathrm{lseg}(x) \equiv\ & (x == \mathrm{null}) \vee \\ & ((x \neq \mathrm{null} \wedge \mathbf{acc}(x)) * \mathrm{lseg}(x.\mathrm{next})) \end{aligned}$$



Verifying delete

```
lseg(x) ≡
    (x == null) ∨
    ((x ≠ null ∧ acc(x)) * lseg(x.next))
```

```
{ lseg(x) }

procedure delete(x: Node)
{
  if (x != null) {
    {(x≠null /\ acc(x)) * lseg(x.next)}
    delete(x.next);

    free(x);

  }
}
{emp}
```

When “if (x != null)” holds, we know x≠null.
Then, lseg(x) is equivalent to (x≠null /\ acc(x)) * lseg(x.next)

Verifying delete

```
lseg(x) ≡
    (x == null) ∨
    ((x ≠ null /\ acc(x)) * lseg(x.next))
```

```
{ lseg(x) }

procedure delete(x: Node)
{
   if (x != null) {
      {(x≠null /\ acc(x)) * lseg(x.next)}
      delete(x.next);
      {(x≠null /\ acc(x)) * emp}
      free(x);

   }
}
{emp}
```

Based on frame rule, delete transforms lseg(x.next) to emp. The other part stay unchanged

Frame Rule:

$$\frac{P \Rightarrow P' * F \qquad \{P'\}\ \text{func}\ \{Q'\}}{\{P\}\ \text{func}\ \{Q' * F\}}$$

In this example:

func: delete(x.next)

P: (x≠null/\acc(x))*lseg(x.next)

P': lseg(x.next)

F: (x≠null/\acc(x))

Q': emp

Verifying delete

```
lseg(x) ≡
    (x == null) ∨
    ((x ≠ null /\ acc(x)) * lseg(x.next))
```

```
{ lseg(x) }

procedure delete(x: Node)
{
   if (x != null) {
      {(x≠null /\ acc(x)) * lseg(x.next)}
      delete(x.next);
      {(x≠null /\ acc(x)) * emp}
      free(x);
      {(x≠null /\ emp) * emp}
   }
}
{emp}
```

Specification of free:

{acc(x)} free(x) {emp}

Frame Rule:

$$\frac{P \Rightarrow P' * F \qquad \{P'\}\ \text{func}\ \{Q'\}}{\{P\}\ \text{func}\ \{Q' * F\}}$$

In this example:

func: **free**(x)

P: (x≠null /\ acc(x)) * emp

P': acc(x)

Q': emp

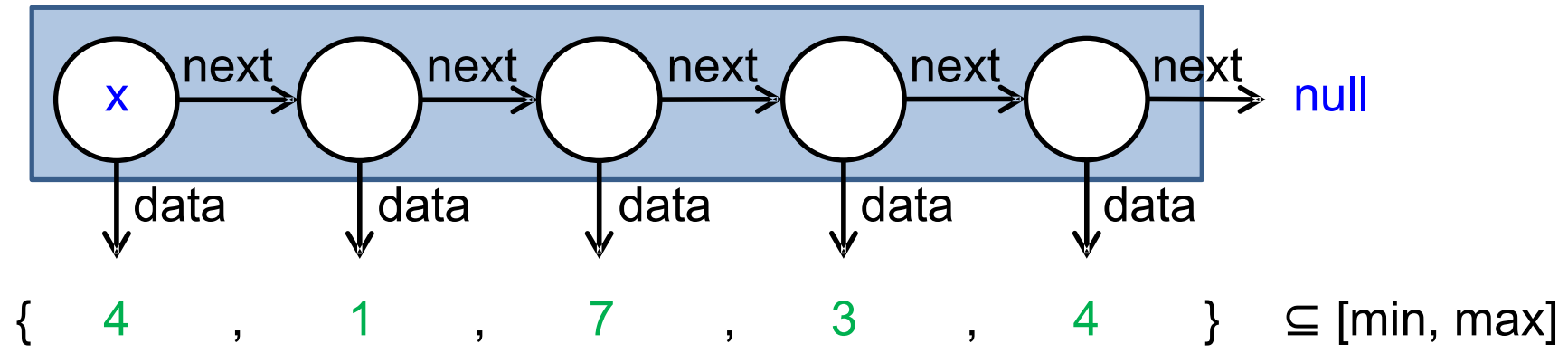
F: x≠null /\ emp

Exercise

Formalization of a bounded list segment

- x is null or x points to a linked list
- The “data” of each node should be within [min, max]

```
bnd_lseg(x, min, max) = ?
```

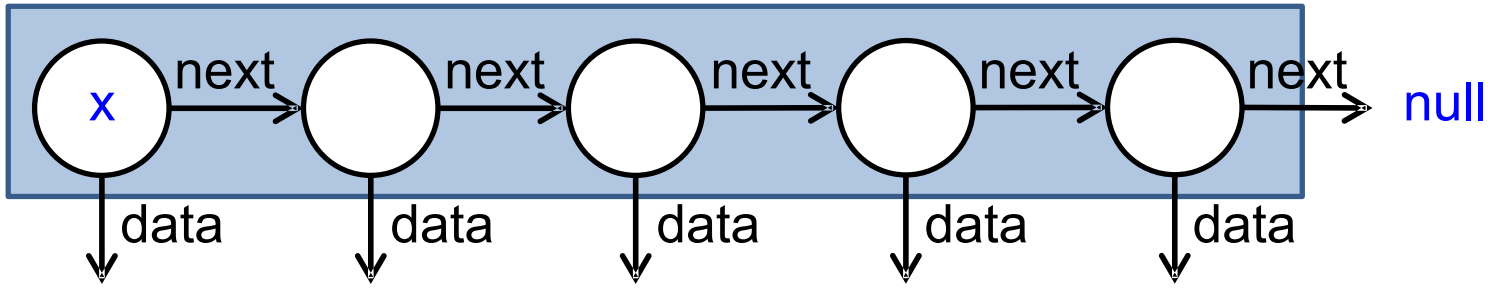


Exercise

Formalization of a bounded list segment

- x is null or x points to a linked list
- The “data” of each node should be within [min, max]

```
bnd_lseg(x, min, max) =
    (x = null) ∨
    ((x ≠ null /\ acc(x) /\ min ≤ x.data ≤ max) * bnd_lseg(x.next, min, max))
```



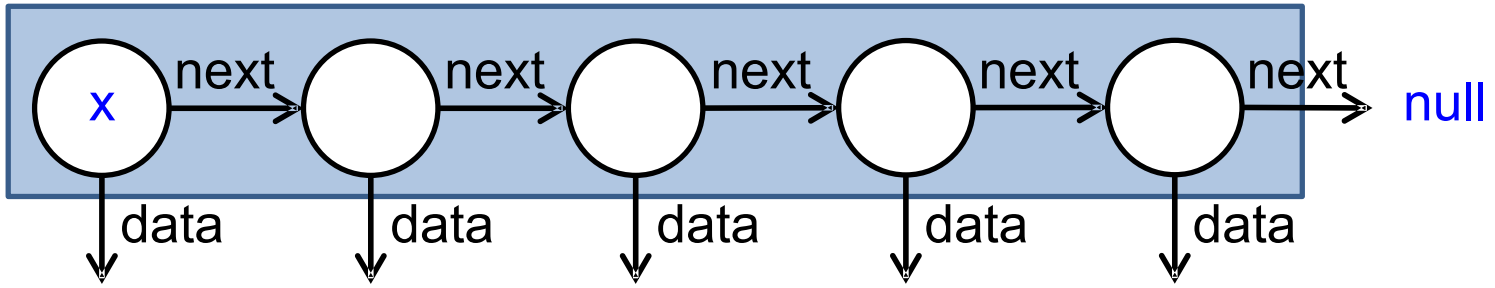
{ 4 , 1 , 7 , 3 , 4 } ⊆ [min, max]

Exercise

Formalization of a sorted list segment

- x is null or x points to a sorted linked list
- The “data” of each node should be within [min, max]

```
srt_lseg(x, min, max) = ?
```



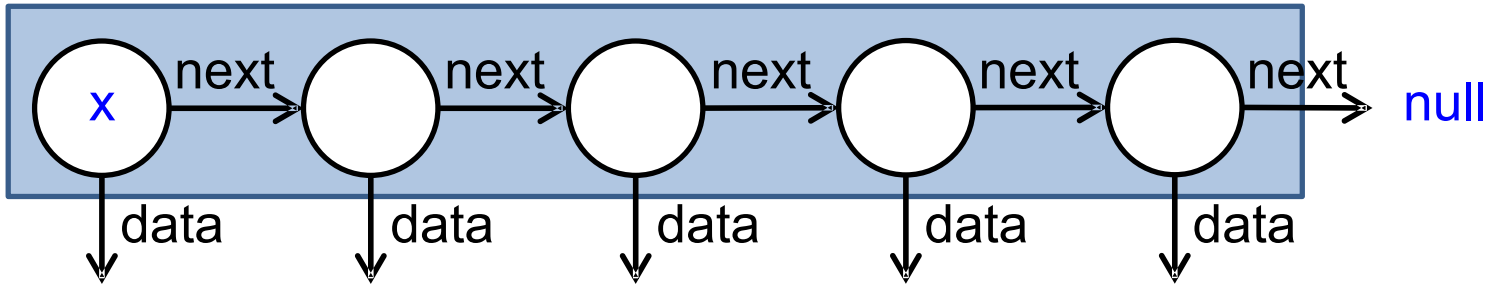
min ≤ 1 ≤ 3 ≤ 4 ≤ 4 ≤ 7 ≤ max

Exercise

Formalization of a sorted list segment

- x is null or x points to a sorted linked list
- The “data” of each node should be within [min, max]

```
srt_lseg(x, min, max) =
  (x = null) ∨
  ((x ≠ null /\ acc(x) /\ min ≤ x.data ≤ max) * srt_lseg(x.next, x.data, max))
```



min ≤ 1 ≤ 3 ≤ 4 ≤ 4 ≤ 7 ≤ max

VeriFast

A formal verifier for C and Rust programs.

- Users provide specification and proof annotations
  - It is not fully automated
  - “Dafny” for C and Rust
- Verifast checks memory safety and specified postconditions.

https://github.com/verifast/verifast

Separation Logic Notation in VeriFast

SL Notation	VeriFast Notation	Meaning
`F * G`	`F &*& G`	The allocated heap consists of two disjoint parts
`acc(x)`	`malloc_block_node(x)`	x points to an allocated node in heap
`x->next == y`	`x->next |-> y`	The “next” field of x is y
`emp`	`emp`	No allocated heap
`{P} c {Q}`	`requires P;` `ensures Q;`	Precondition and postcondition

All code used for verification (specification, predicate, ...) are within following special comments:

- /*@ ...VeriFast code... @*/
- //@ ...VeriFast code...

Define node and linked list in Verifast

A single node in linked list

```
#include <stdlib.h>

struct node {
    struct node *next;
};

/*@
predicate nodes(struct node *x) =
    x == 0 ?
        emp
    :
        x->next |-> ?next &*&
        malloc_block_node(x) &*&
        nodes(next);
@*/
```

- ?next is a placeholder representing the value of x->next
- `x->next |-> ?next` means that `exists ? next, the “next” field of x is ?next`
- malloc_block_node(x) means acc(x)
- &*& means “*” in separation logic

The predicate is the verifast code for the formula in previous slides:

$$lseg(x) \equiv (x == null)\ \vee\ ((x \neq null\ \wedge \mathbf{acc}(x)) * lseg(x.next))$$

Define Specification of delete in VeriFast

```
void delete(struct node *x)
    //@ requires nodes(x);
    //@ ensures emp;
{

    if (x != 0) {
        struct node *next = x->next;
        delete(next);
        free(x);
    }
}
```

```
predicate nodes(struct node *x) =
    x == 0 ?
        emp
    :
        x->next |-> ?next &*&
        malloc_block_node(x) &*&
        nodes(next);
```

Pre-condition: x is null or x points to a linked list

Post-condition: No allocated memory

Define Specification of node_create in VeriFast

The following function inserts a head node into a linked list

```
struct node *node_create(struct node *next)
    //@ requires nodes(next);
    //@ ensures nodes(x);
{
    struct node *x = malloc(sizeof(struct node));

    x->next = next;

    return x;
}
```

Pre-condition: next is null or next points to a linked list

Post-condition: x points to a linked list

Running the Complete Example

```
$ verifast delete.c
0 errors found (15 statements verified)
Linking...
Program linked successfully.

$ gcc -std=c11 -Wall -Wextra -Werror delete.c -o delete_demo
$ ./delete_demo
$ echo $?
0
```

VeriFast checks all code

The proof covers any input satisfying pre-condition

An Invalid Example

```
void delete(struct node *x)
    //@ requires nodes(x);
    //@ ensures emp;
{

    if (x != 0) {
        free(x);
        struct node *next = x->next;
        delete(next);
    }
}
```

Use-after-free error

VeriFast reports:

```
(26,10-12): error: No matching heap chunks: [_]node_next(x, _)
26 |        next = x->next;
   |        ^^
```

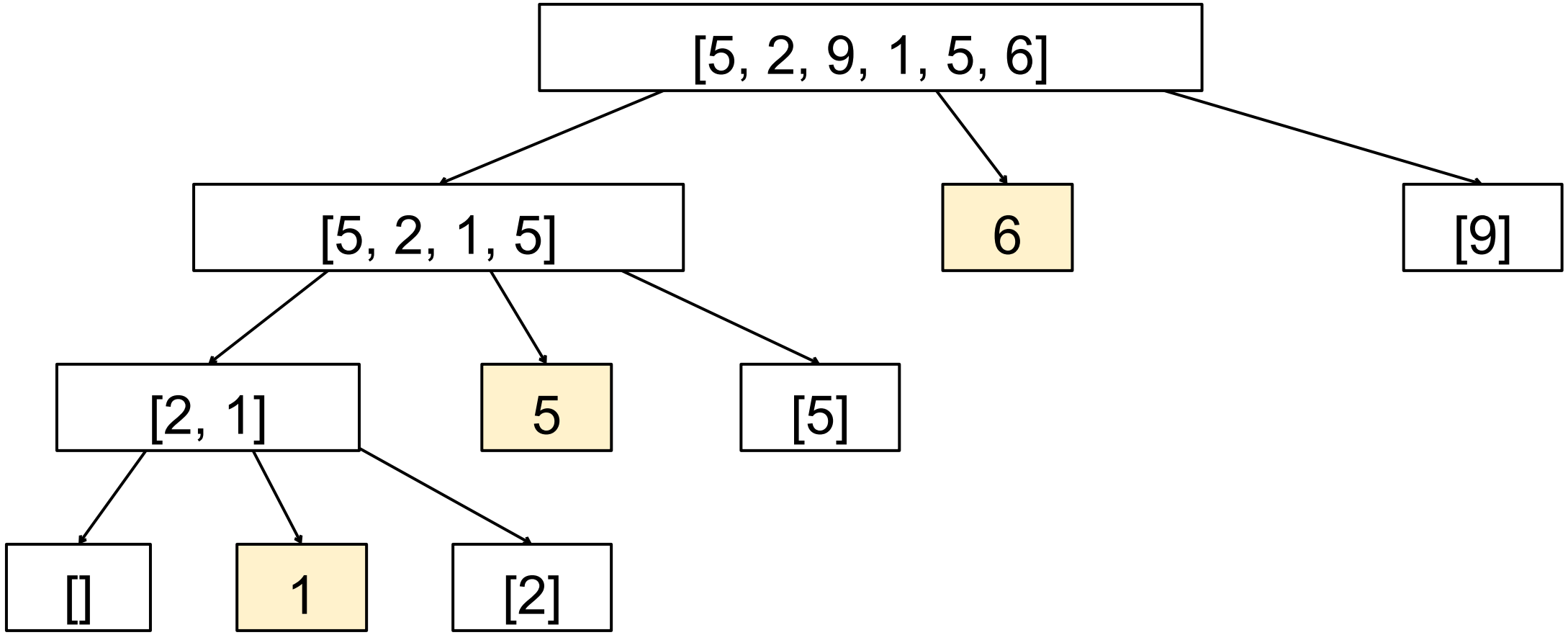
Exercise: Quicksort

```
void quicksort(int *a, int n)
{
    if (n <= 1) return;
    int p = partition(a, n);
    quicksort(a, p);
    quicksort(a + p + 1, n - p - 1);
}
```

- The initial n is the array length
- Select the last element in the array to be a pivot
- Partition the array into two disjoint sides such that
  - Every element a[0], ... a[p-1] <= pivot
  - Every element a[p+1], ..., a[n-1] >= pivot
- The pivot a[p] stays fixed while the two sides are recursively sorted

Quicksort Example

Each yellow pivot stays fixed while the white subarrays are sorted.



Sorted result: [1, 2, 5, 5, 6, 9]

Partition: Scanning the Array

```
int partition(int *a, int n)
{
    int pivot = a[n - 1];
    int i = 0;
    for (int j = 0; j < n - 1; j++) {
        int aj = a[j];
        if (aj < pivot) {
            swap(a, i, j);
            i++;
        }
    }

    if (i < n - 1) swap(a, i, n - 1);
    return i;
}
```

i is an index such that every element a[0], .., a[i-1] <= pivot

If a[j] < pivot, swap a[i] and a[j] such that every element a[0], .., a[i-1] <= pivot

Since each element <= pivot has been in a[0, i-1], pivot should be put to a[i] such that the array is partitioned to two sides. Left side <= pivot and right side >= pivot

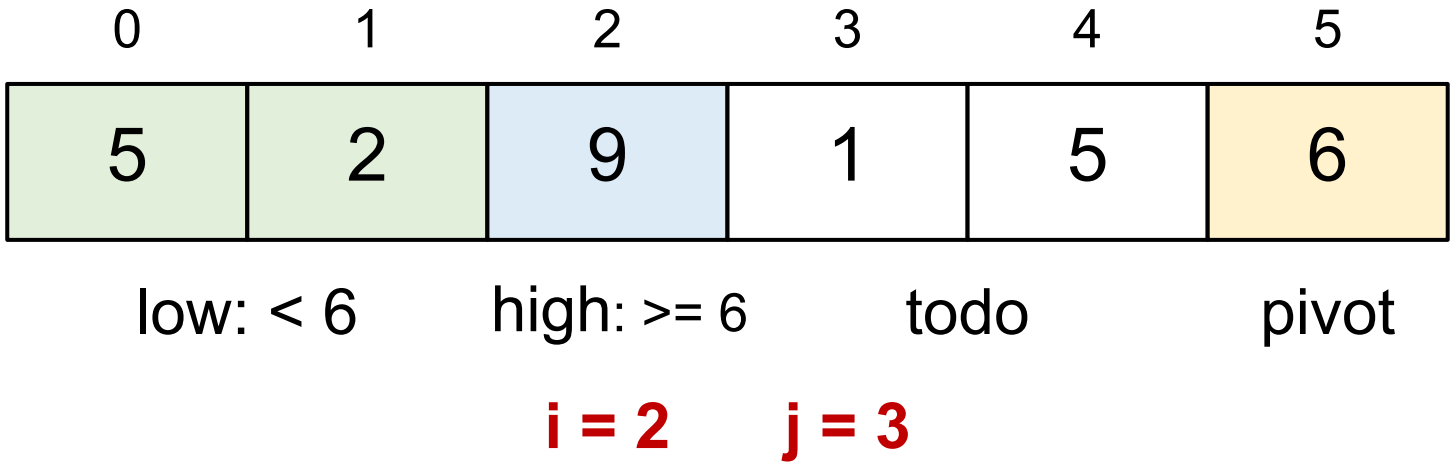
The loop iteratively puts each element <= pivot into a[0, i-1]

Finally, pivot is put to a[i] and it ensures

- every element in a[0, i-1] <= pinot
- every element in a[i, n-1] >= pinot

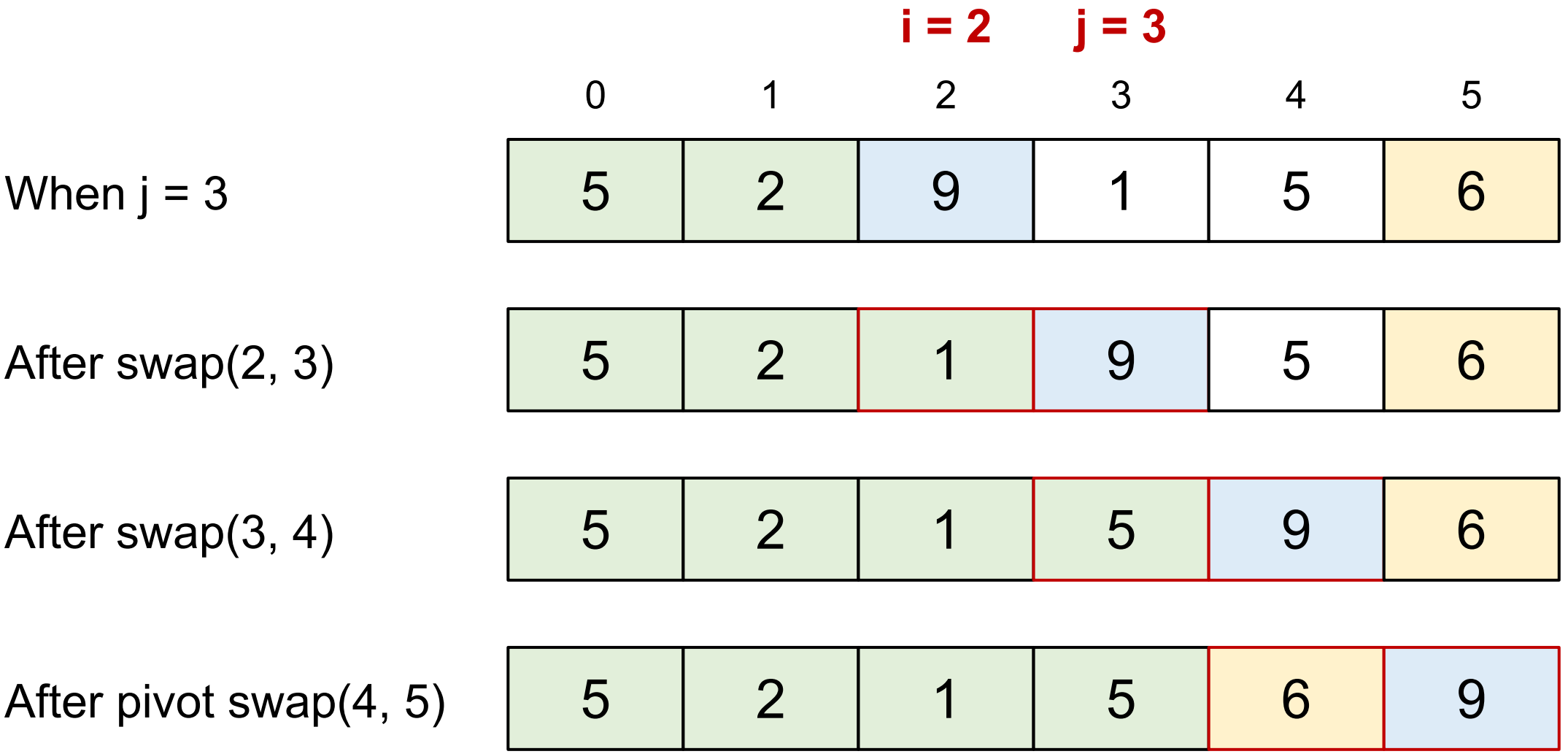
Example of Partition

Assume j = 3, pivot = 6, i = 2



- The last element is the pivot 6
- Every element in a[0, i-1] is less than or equal pivot
- The loop will iteratively check a[j, n-2] to put all element <= pivot into a[0, i-1]

Example of Partition



Specification: quicksort

```
void quicksort(int *a, int n)
    //@ requires a[0..n] |-> ?vs;
    //@ ensures a[0..n] |-> ?ws &*& sorted(?ws) && perm(?vs, ?ws);
```

Pre-condition

- `a[0..n] |-> ?vs:` there exists an integer list ?vs that equals to the initial array
- In VeriFast code, a[0..n] contains a[0], ..., a[n-1] without a[n]

Post-condition

- `a[0..n] |-> ?ws:` there exists an integer list ?ws that equals to the array after quicksort
- `sorted(?ws):` the list ?ws is sorted
- `perm(?vs, ?ws):` ?ws is a permutation of ?vs
  - For every integer, it has the same count in ?vs and ?ws. It means that both lists have the same group of integers

Definition of sorted used in specification

```
fixpoint bool le(int x, int y) { return x <= y; }


fixpoint bool sorted(list<int> vs) {
    switch (vs) {
        case nil: return true;
        case cons(h, t): return forall(t, (le)(h)) && sorted(t);
    }
}
```

- `nil`: empty array
- `cons(h, t)`: The first array element is h, the sub-array following h is t
- `forall(t, (le)(h))`: For each element in t, the value h <= the element
- `sorted(t)`: t is sorted

Definition of permutation

```
fixpoint bool eq(int x, int y) { return x == y; }

fixpoint bool same_count(list<int> xs, list<int> ys, int v) {
    return count(xs, (eq)(v)) == count(ys, (eq)(v));
}

// xs and ys contain every value the same number of times.
fixpoint bool perm(list<int> xs, list<int> ys) {
    return forall_int((same_count)(xs, ys));
}
```

- `count(xs, (eq)(v))`

  The count of elements in xs that is equivalent to v

- `forall_int((same_count)(xs, ys))`

  For each integer, it has the same count in both arrays xs and ys

Specification: swap

```
void swap(int *a, int i, int j)
    //@ requires a[i] |-> ?vi &*& a[j] |-> ?vj;
    //@ ensures a[i] |-> vj &*& a[j] |-> vi;
{
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```

Pre-condition

- `a[i] |-> ?vi:` there exists a value ?vi such that a[i] is ?vi
- `a[j] |-> ?vj:` there exists a value ?vj such that a[j] is ?vj

Post-condition

- The values of a[i] and a[j] are exchanged

Specification: partition

```
int partition(int *a, int n)
    //@ requires a[0..n] |-> ?vs && 0 < n;
    /*@
    ensures 0 <= result && result < n && a[result] |-> ?p &*&
        a[0..result] |-> ?low && forall(low, (ge)(p)) &*&
        a[result + 1..n] |-> ?high && forall(high, (le)(p)) &*&
        perm(vs, append(low, cons(p, high)));
    @*/
```

Pre-condition

- `a[0..n] |-> ?vs:` there exists an intger list ?vs that equals to the initial array
- `0 < n`: The array is not empty

Specification: partition

```
int partition(int *a, int n)
    //@ requires a[0..n] |-> ?vs && 0 < n;
    /*@
    ensures 0 <= result && result < n && a[result] |-> ?p &*&
        a[0..result] |-> ?low && forall(low, (ge)(p)) &*&
        a[result + 1..n] |-> ?high && forall(high, (le)(p)) &*&
        perm(vs, append(low, cons(p, high)));
    @*/
```

Post-condition

- `0 <= result && result < n && a[result] |-> ?p`

  Return value (`result`) is an array index and there exists a pivot value ?p such that a[result] = ?p

- `a[0..result] |-> ?low && forall(low, (ge)(p))`

  There exists an integer list ?low such that

  - ?low equals a[0...result]
  - pivot value >= each element in a[0...result]

Specification: partition

```
int partition(int *a, int n)
    //@ requires a[0..n] |-> ?vs && 0 < n;
    /*@
    ensures 0 <= result && result < n && a[result] |-> ?p &*&
        a[0..result] |-> ?low && forall(low, (ge)(p)) &*&
        a[result + 1..n] |-> ?high && forall(high, (le)(p)) &*&
        perm(vs, append(low, cons(p, high)));
    @*/
```

Post-condition

- `a[result + 1..n] |-> ?high` && forall(high, (le)(p))

  There exists a integer list ?high such that

  - ?high equals a[result+1..n]
  - pivot value ?p <= each element in a[result+1..n]

Verify QuickSort

```
void quicksort(int *a, int n)
    //@ requires a[0..n] |-> ?vs;
    //@ ensures a[0..n] |-> ?ws &*& sorted(ws) && perm(vs, ws);
```

```
$ verifast quicksort.c
0 errors found (112 statements verified)
Linking...
Program linked successfully.
```

VeriFast checks all code

The proof covers any input satisfying `a[0..n] |-> ?vs`