Mark-sweep GC

How do we mark reachable objects?

Disadvantages of mark-sweep GC

- Stop-the-world algorithm
  - Computation suspended while GC runs
  - Pause time may be high
    - Not practical for real-time, interactive applications, video games
- High cost:
  - proportional to size of heap (not just live objects)
  - Why?
    - Active objects visited by mark phase
    - All of memory visited by sweep phase

Mark-sweep algorithm

```
// The mark-sweep collector
mark_sweep() {
    for R in Roots
        mark(*R)
    sweep()
    if free_pool is empty
        abort "Memory exhausted"
}
// Simple recursive marking
mark(N) {
    if mark_bit(N) == unmarked
        mark_bit(N) = marked
        for M in Children(N)
            mark(*M)
}
```

```
// The eager sweep of the heap
sweep() {
    N = Heap_bottom
    while N < Heap_top
        if mark_bit(N) == unmarked
            free(N)
        else mark_bit(N) = unmarked
        N = N + size(N)
}
```

How can we improve marking?

- Using a marking stack
  - Problem:
    - Recursion may cause system stack to overflow
    - Procedure overhead: both time and space
  - Solutions:
    - Replace recursive calls with iterative loops
    - Use auxiliary data structures (*e.g.*, a stack data structure)
      - Stack holds pointers to objects known to be live
      - Unmarked children marked, pushed on stack if have pointers
      - Objects without pointers only marked
      - Marking phase terminates when stack is empty

Marking stack

- Maximum depth of stack
  - Depends on longest path through graph that has to be marked
  - In most systems stacks are generally shallow
- Safe GC must be able to handle exceptional cases
  - May need to minimize stack depth

Iterative Marking

```
mark_heap() {
    mark_stack = empty
    for R in Roots
        mark_bit(*R) = marked
        push(*R, mark_stack)
        mark()
}
```

```
mark() {
    while mark_stack != empty
        N = pop(mark_stack)
        for M in Children(N)
            if mark_bit(*M) == unmarked
                mark_bit(*M) = marked
                if not atom(*M)
                    push(*M, mark_stack)
}
```

Marking with resumption stack

Minimizing stack depth

- Why is this important?
  - Stack can overflow
  - Whys is GC needed?
    - What if the GC runs out of storage?
- How can it be done?
  - push constituent pointers of large objects in small groups onto the stack (Boehm-Demers-Weiser)
  - Using pointer reversal

Addressing stack overflow

- Marking stack can make detection easier and recovery action taken
  - Check in each push operation ($$$$$$)
  - Single check by counting # of pointers in each popped node
  - Use guard page (if OS support)
    - Read-only page. Cannot push unto guard page
- How to handle stack overflow?
  - Knuth's approach
  - Kurokawa's approach

Knuth proposed in 1973

- Treat marking stack circularly

```
for R in Roots
    push(*R, new_roots);
overflow = false;

while true
    overflow = cyclic_stack_mark(new_roots);
    if overflow == true
        new_roots = scan_heap();
    else break;
```

- scan_heap returns marked nodes pointing to unmarked nodes

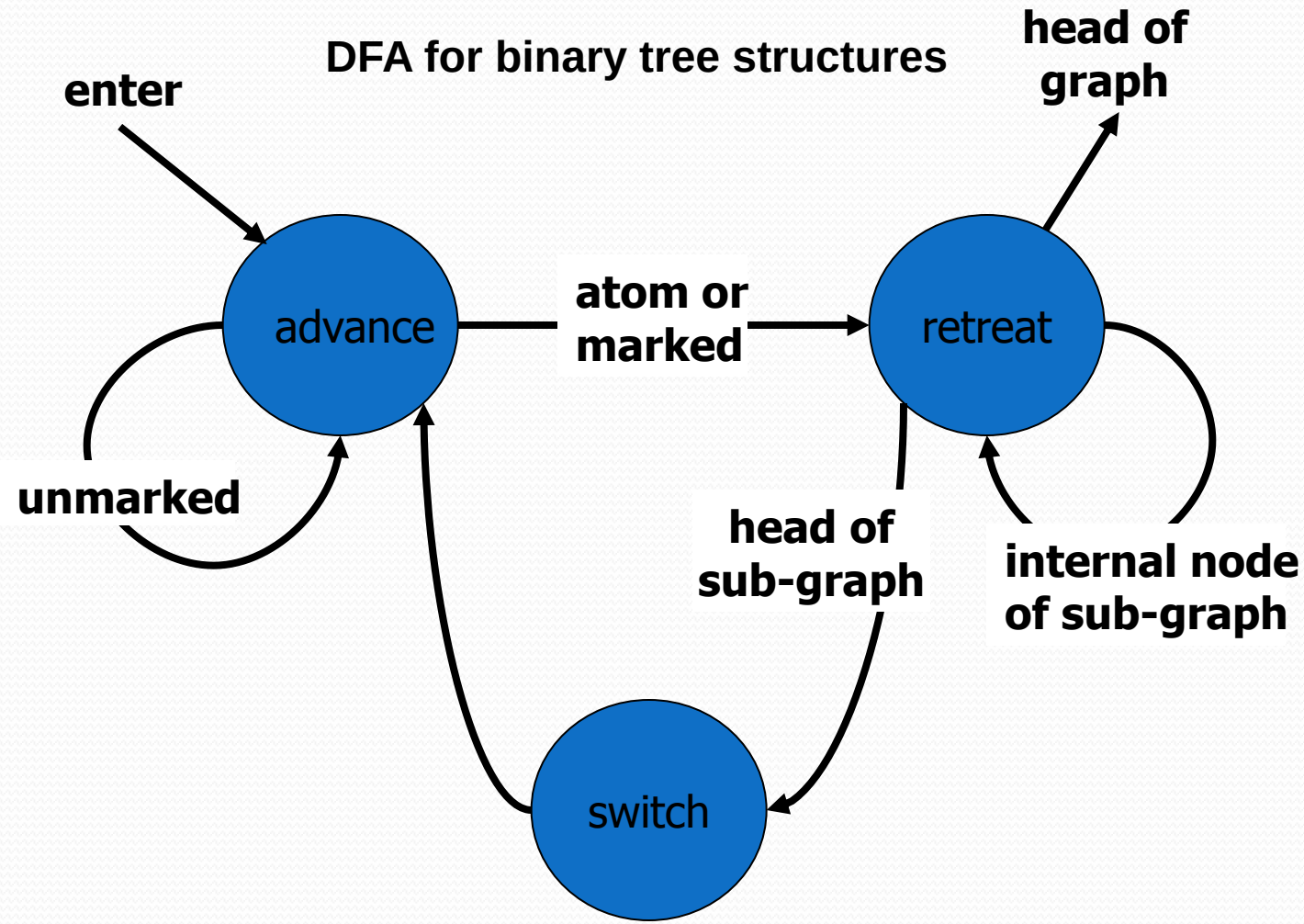
Kurokawa proposal in 1981

- On overflow run ***Stacked Node Checking*** algorithm
- remove items from stack that have fewer than 2 unmarked children
  - no child is unmarked: clear slot
  - one child is unmarked: replace slot entry by a descendent with ***2 or more unmarked*** children marking the passed ones
- Not robust
  - Possible that no additional space will be found on the stack

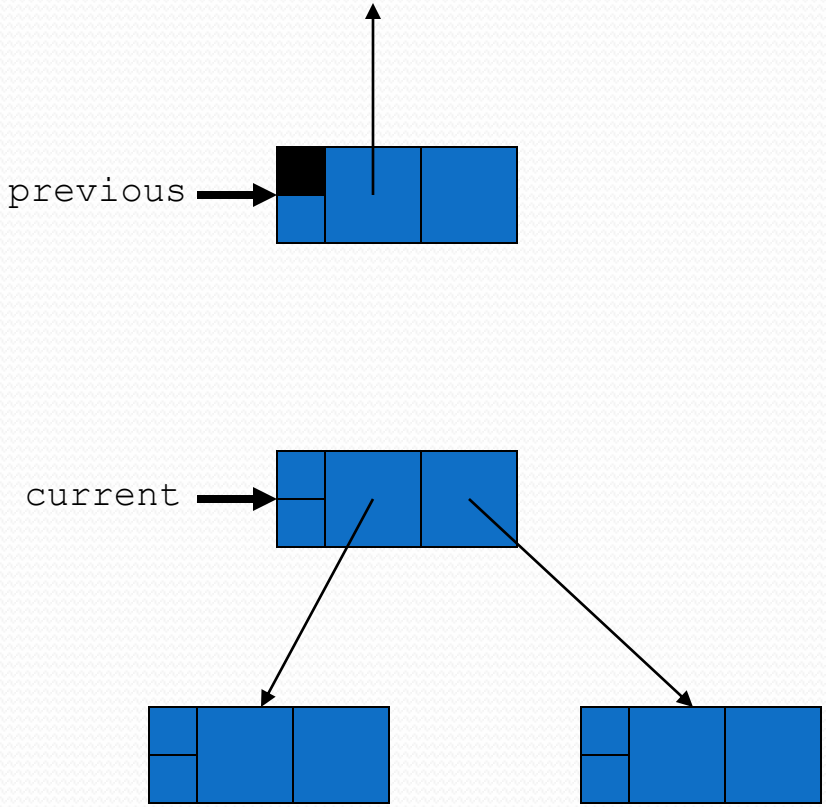
Pointer reversal

- Eliminate need for marking stack
  - Push stack in heap nodes
  - Maintains 3 variables: previous, current, and next
- Any efficient marking process must record the branch-points it passed
- Temporarily reversing of pointers traversed by mark
  - child-pointers become ancestor-pointers
- restore pointer fields when tracing back
- developed independently by Schorr and Waite (1967) and by Deutsch (1973)

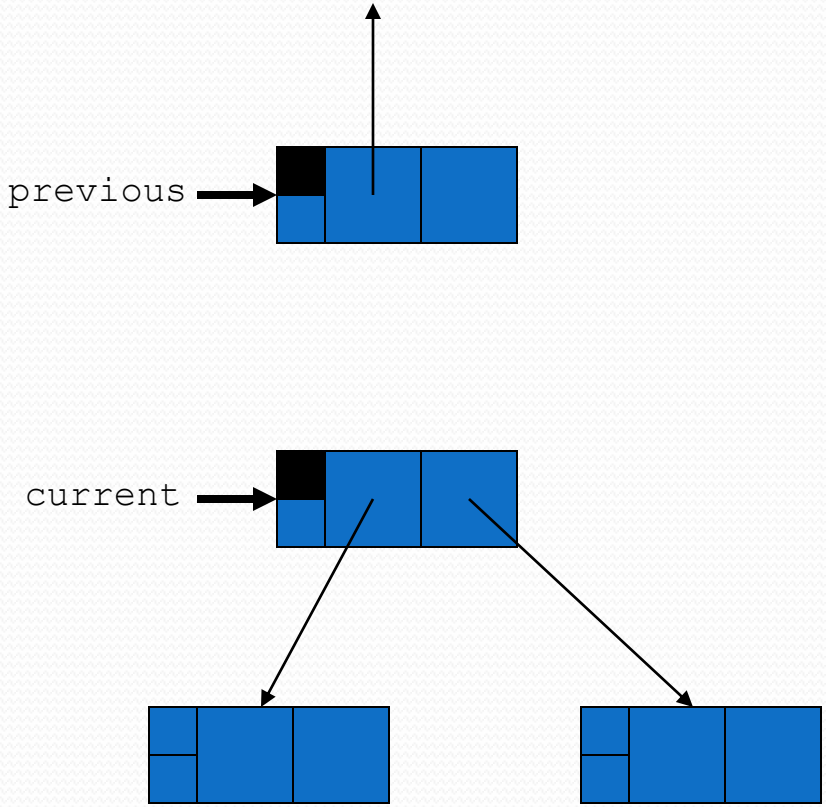
Pointer reversal



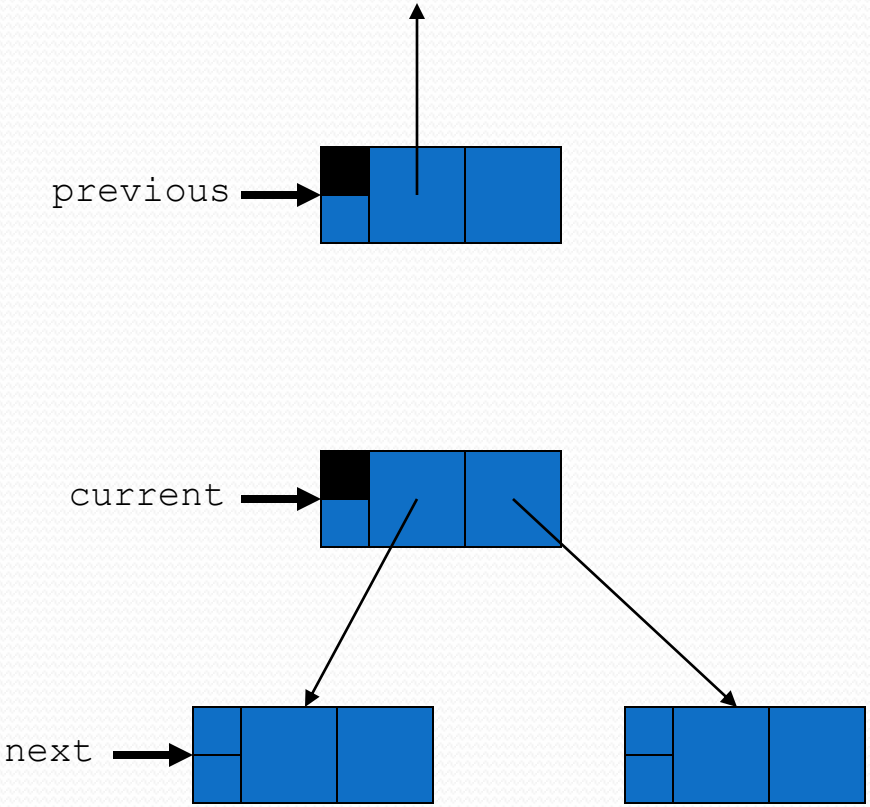
Pointer reversal (advance phase)



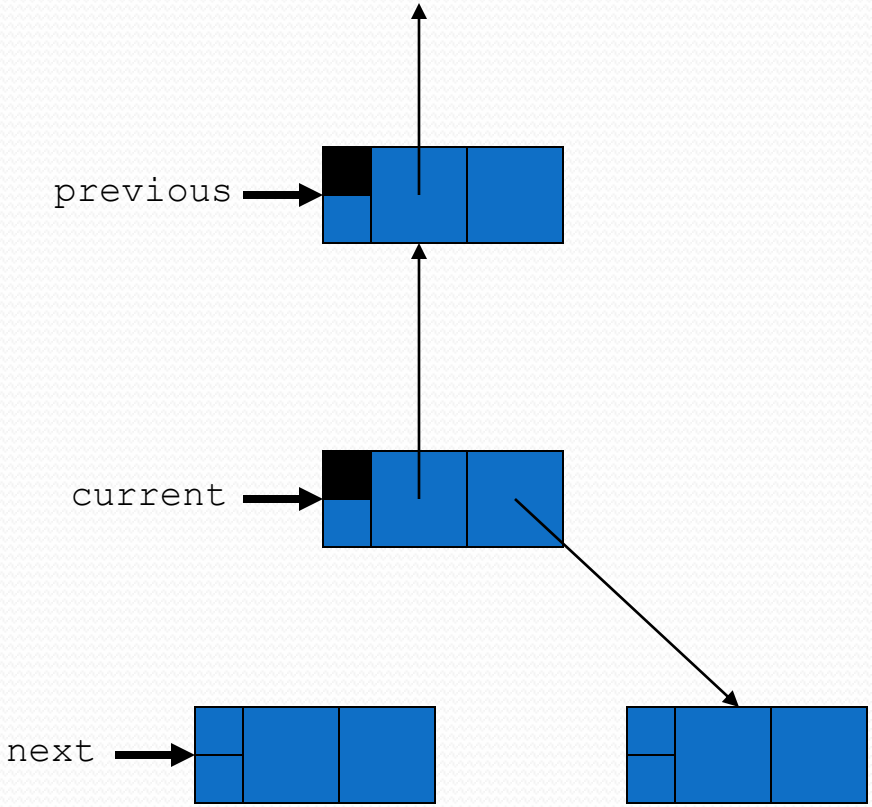
Pointer reversal (advance phase)



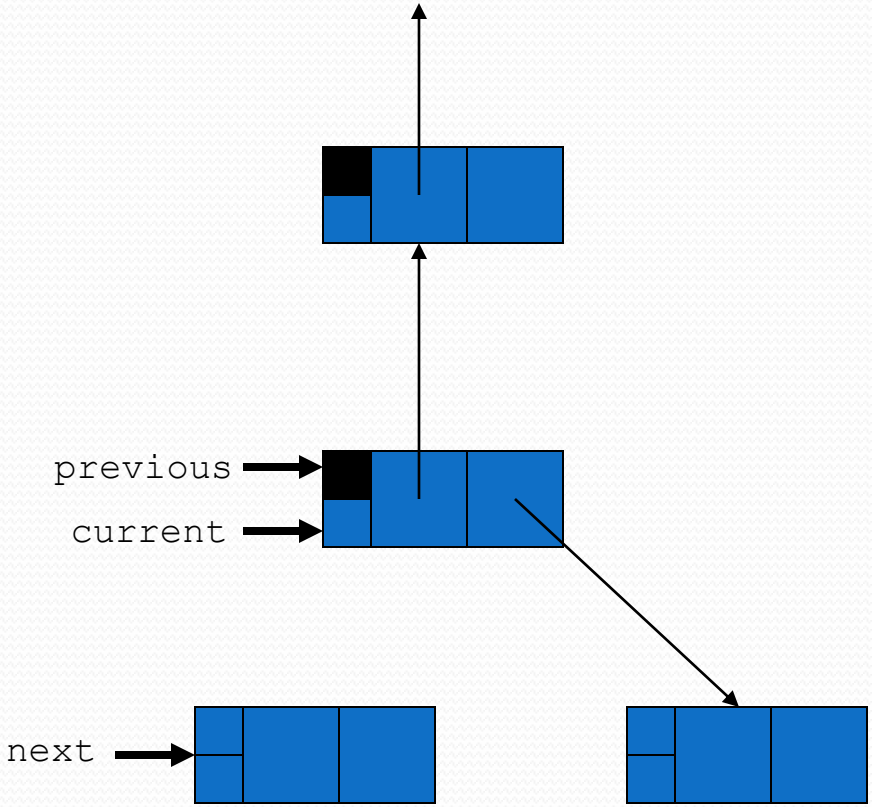
Pointer reversal (advance phase)



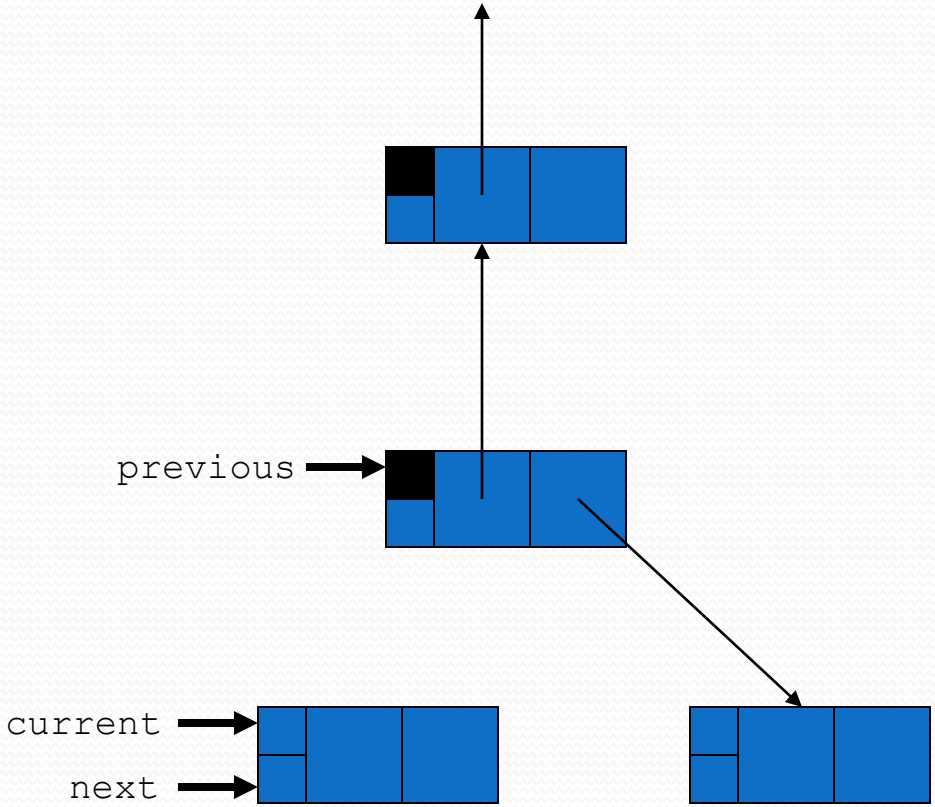
Pointer reversal (advance phase)



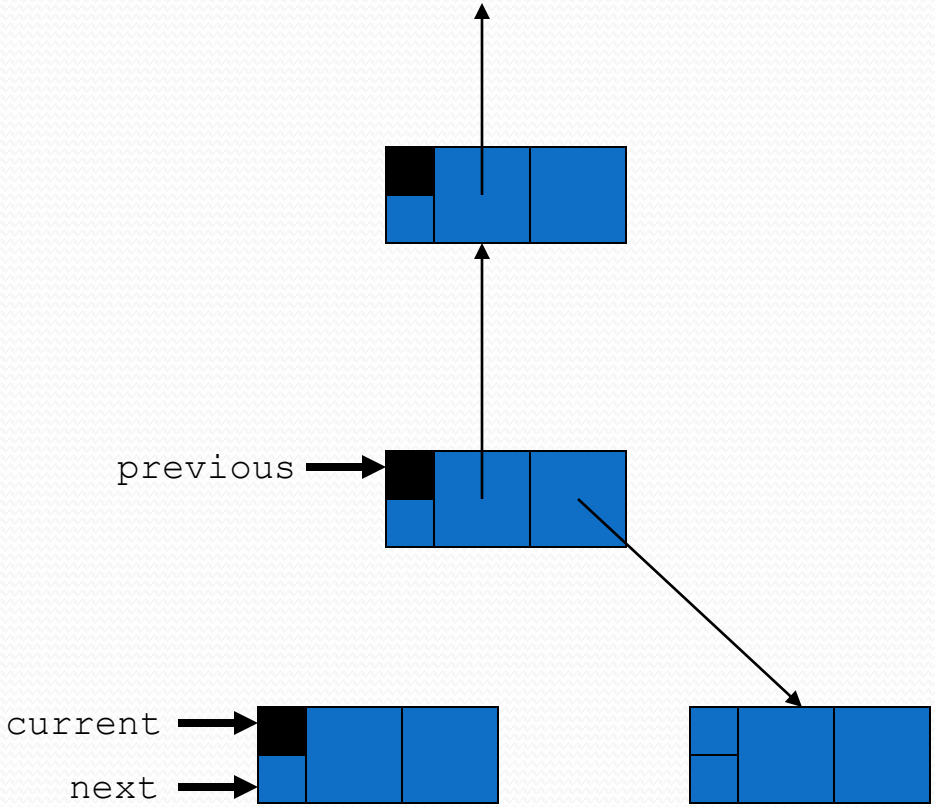
Pointer reversal (advance phase)



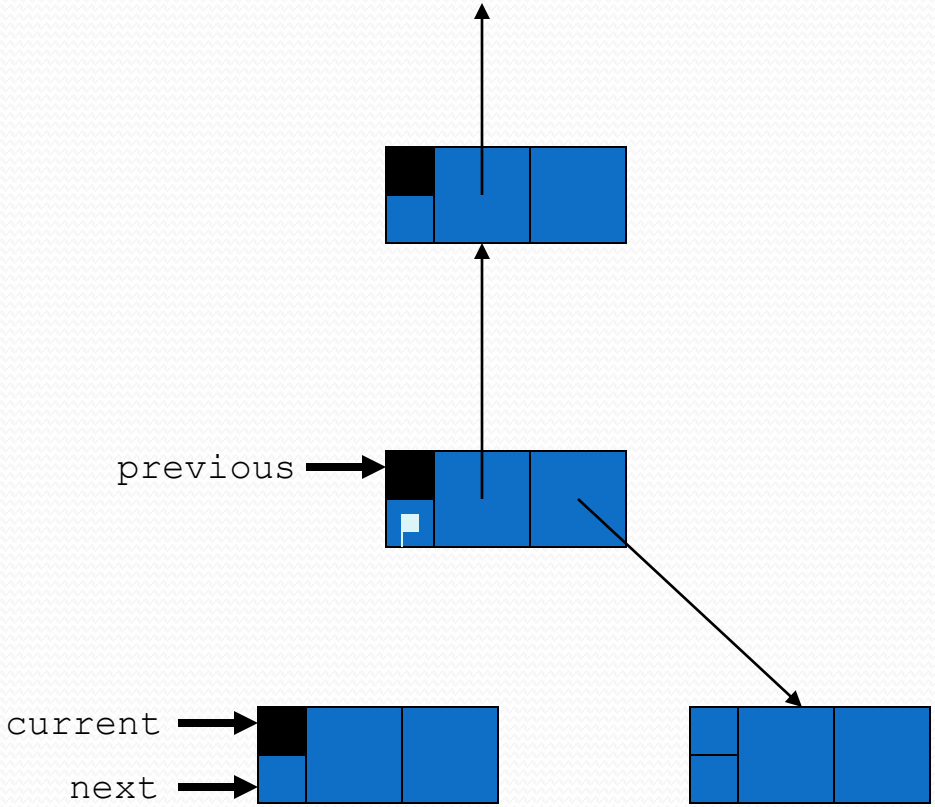
Pointer reversal (advance phase)



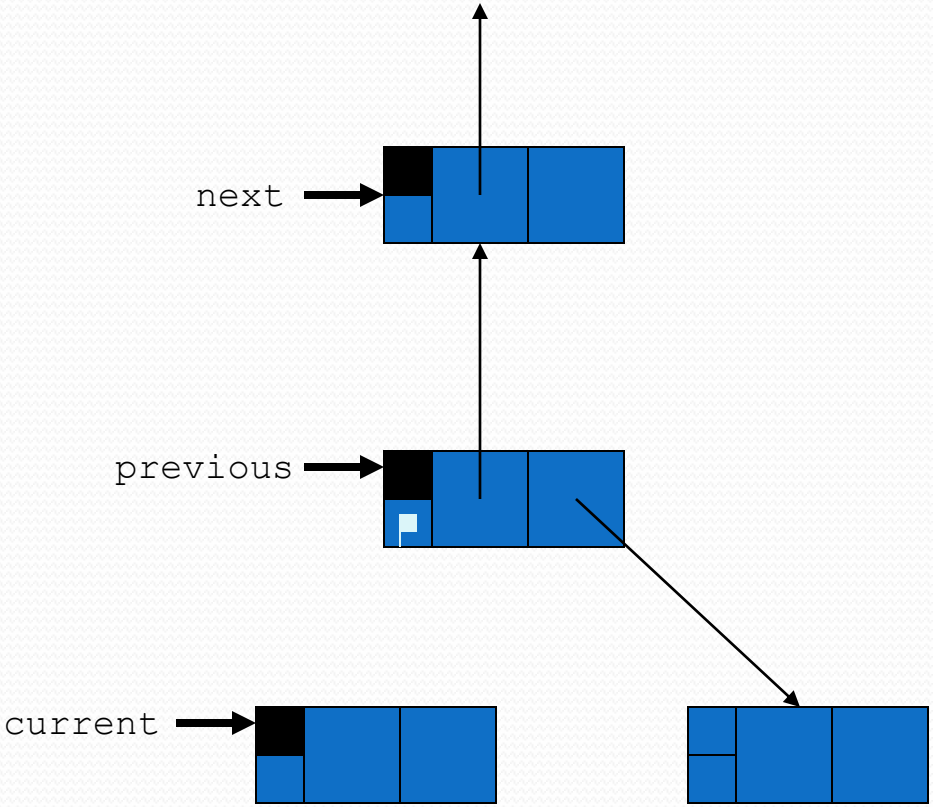
Pointer reversal (switch phase)



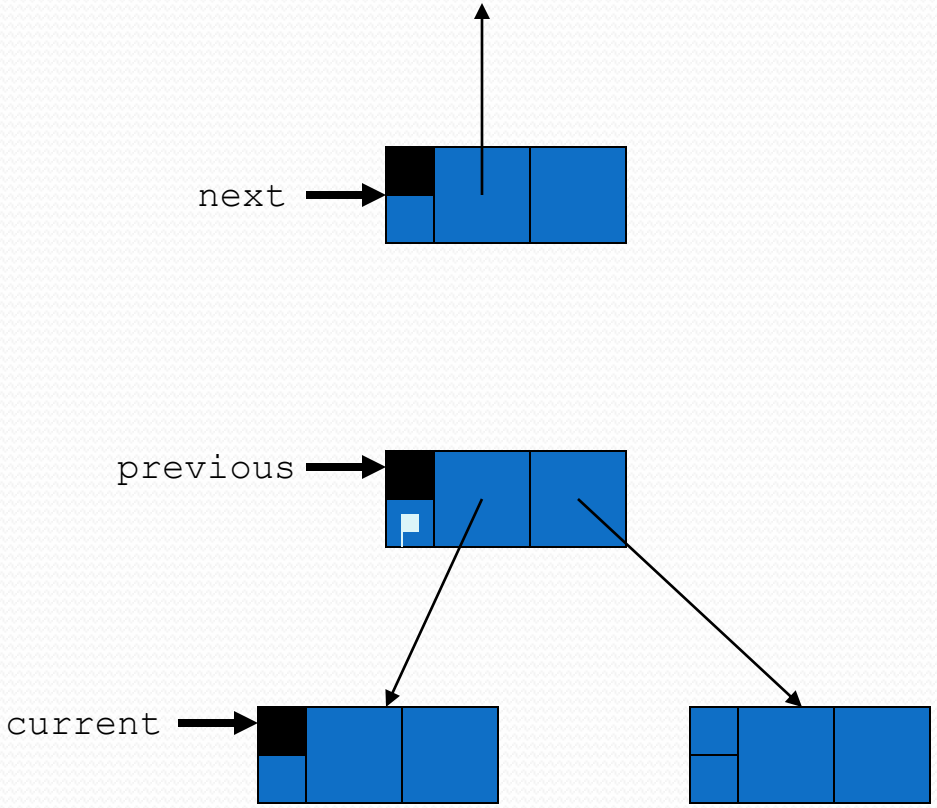
Pointer reversal (switch phase)



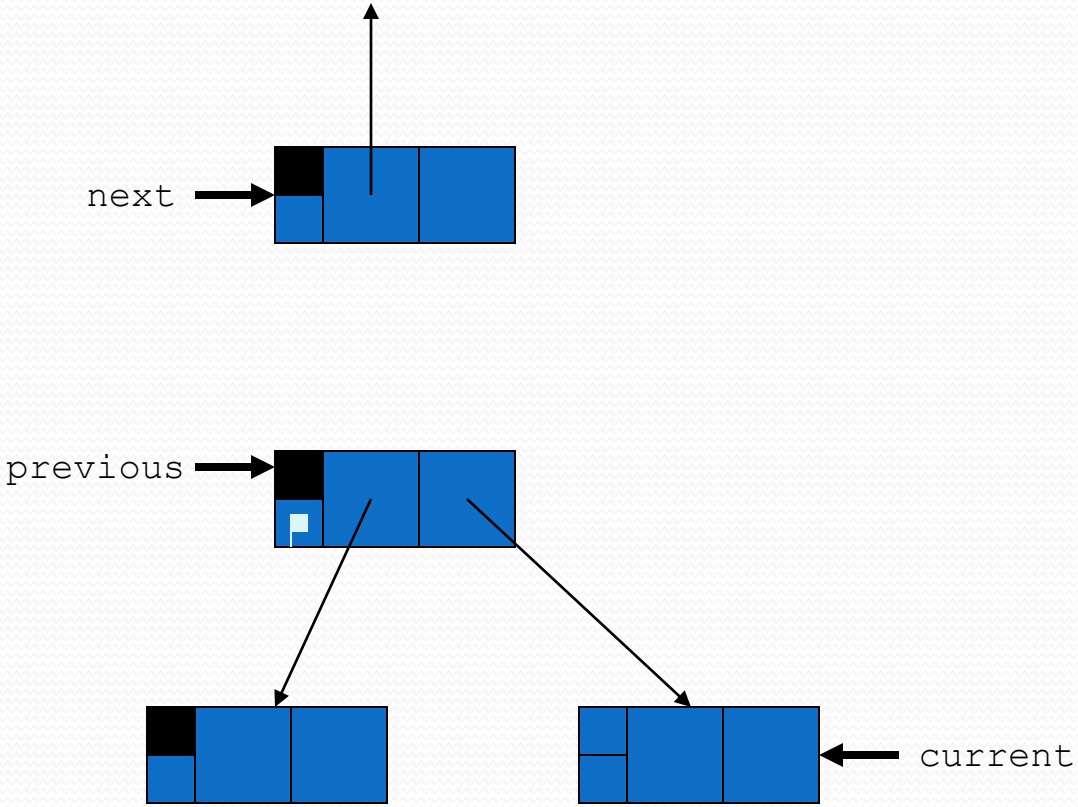
Pointer reversal (switch phase)



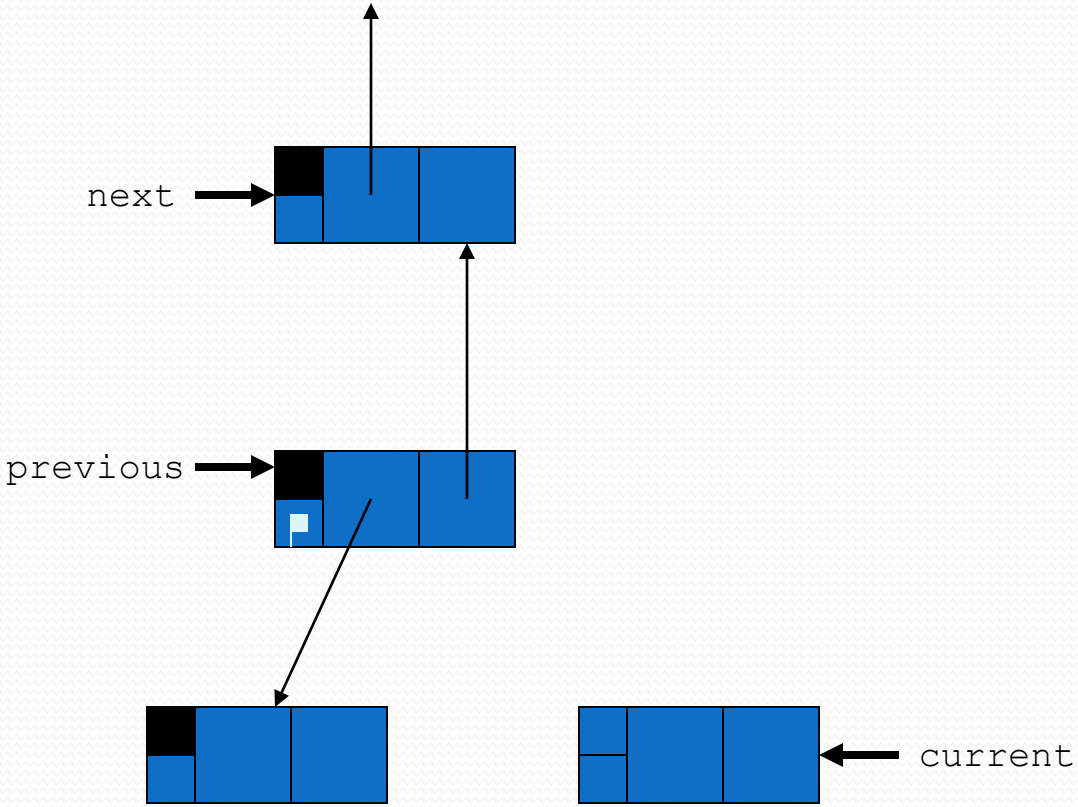
Pointer reversal (switch phase)



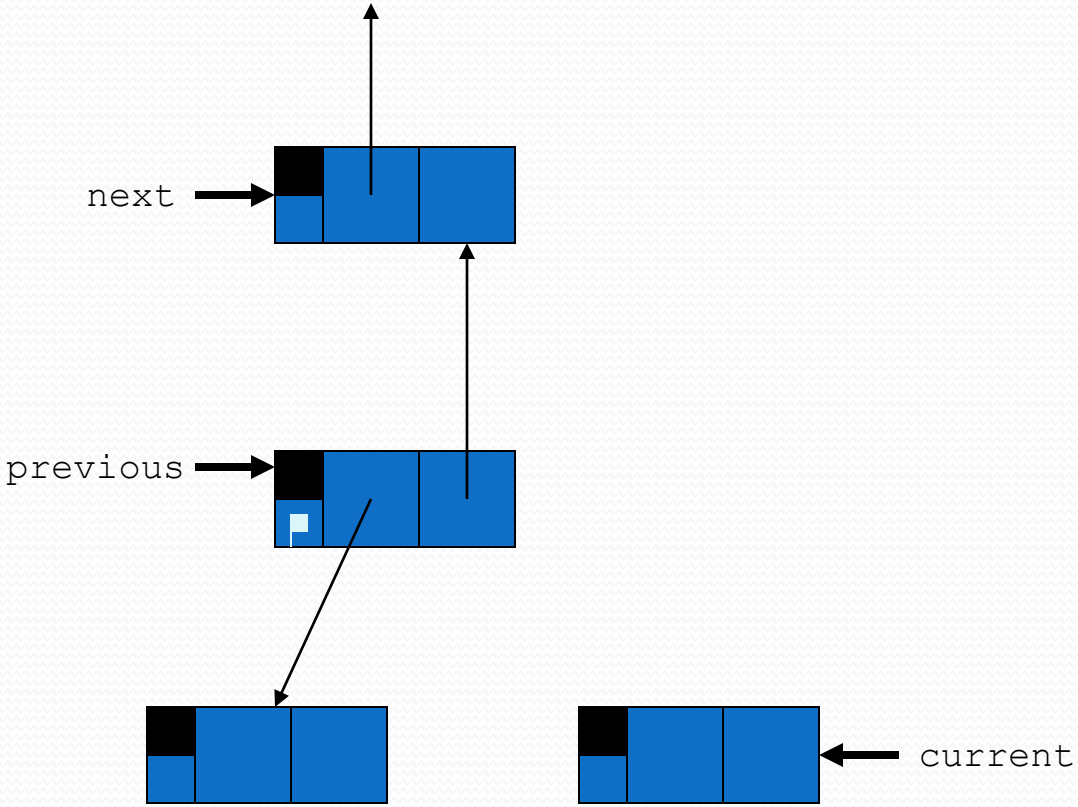
Pointer reversal (switch phase)



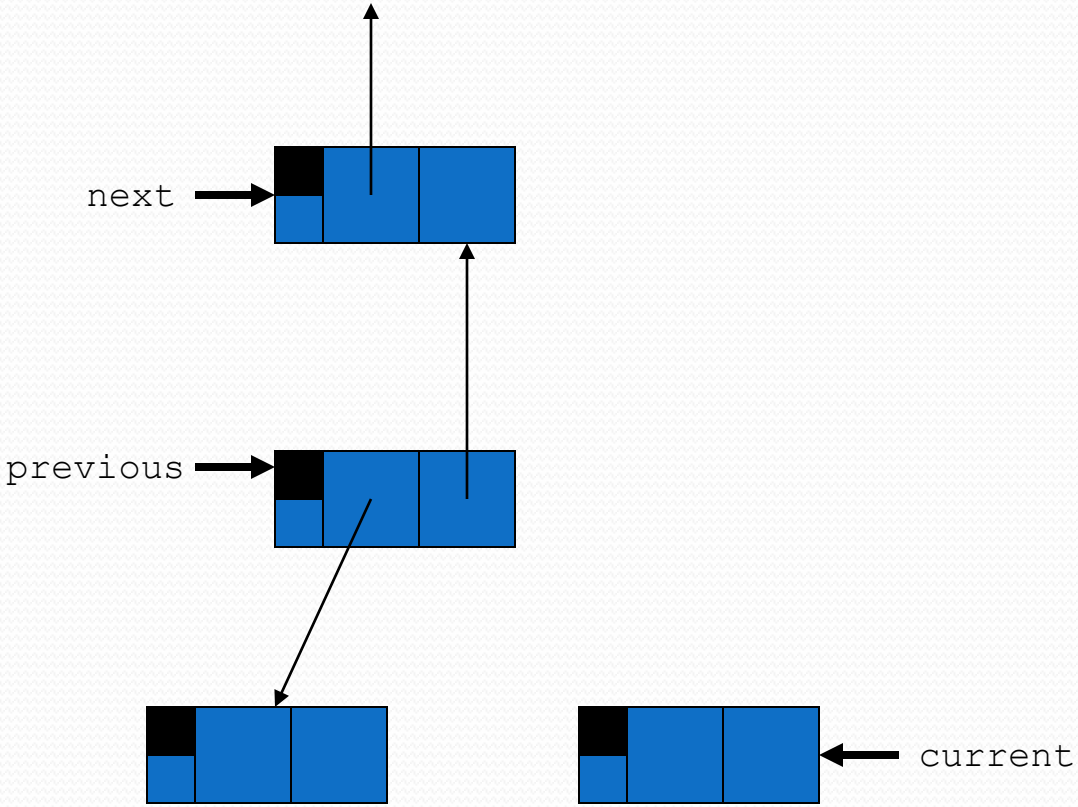
Pointer reversal (switch phase)



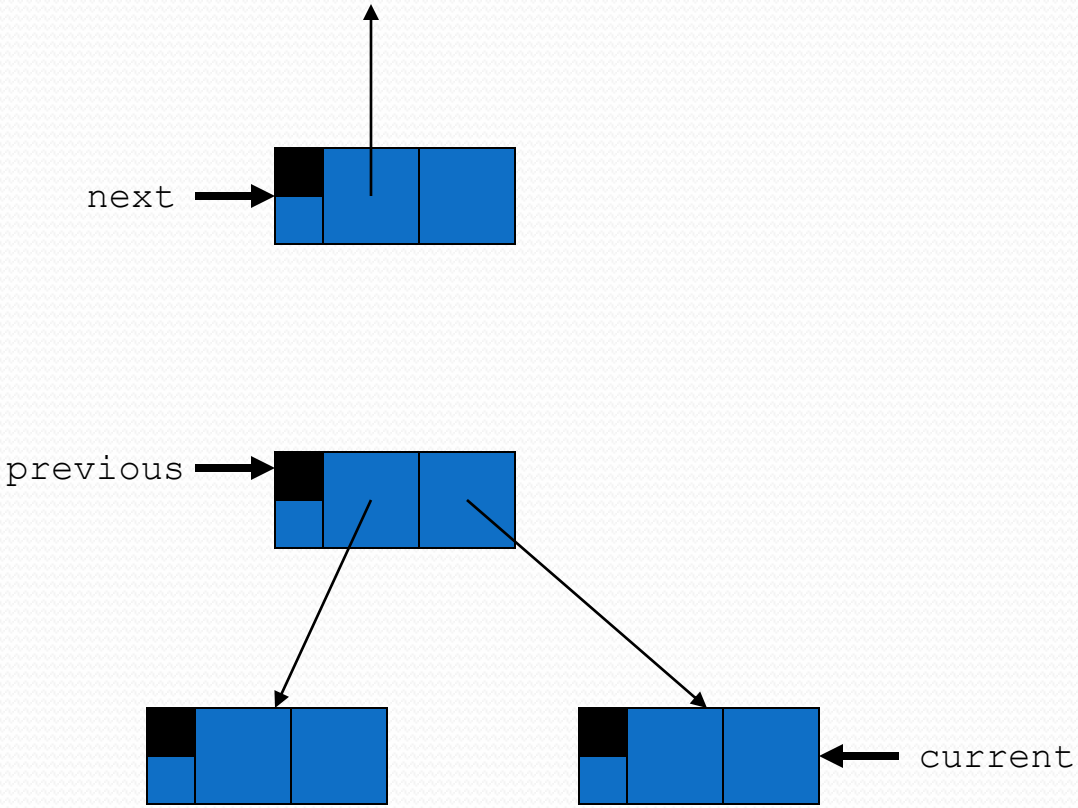
Pointer reversal (retreat phase)



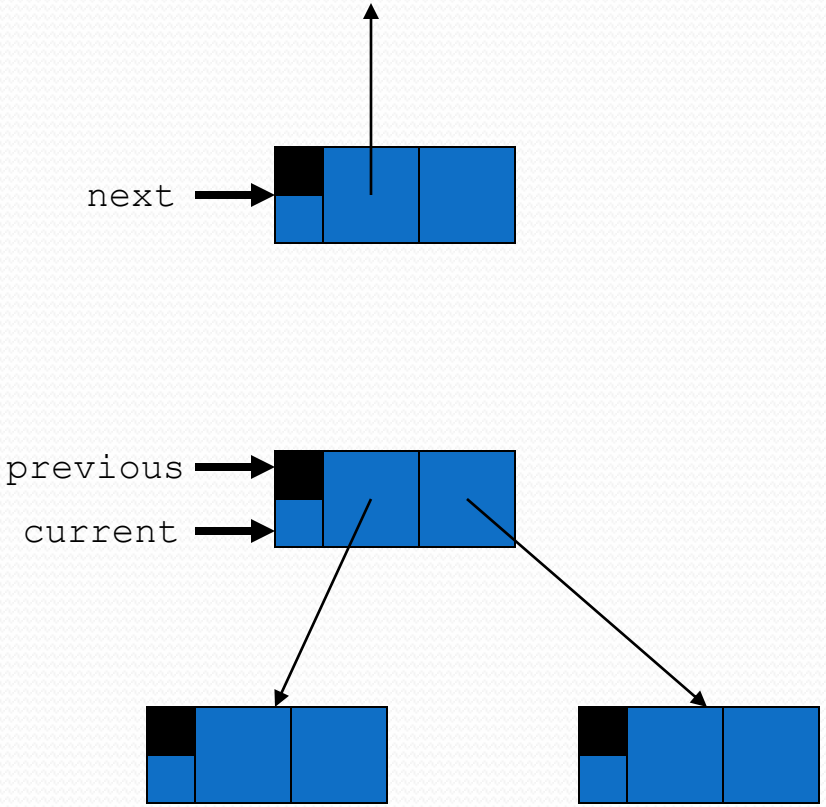
Pointer reversal (retreat phase)



Pointer reversal (retreat phase)



Pointer reversal (retreat phase)



Pointer reversal (retreat phase)