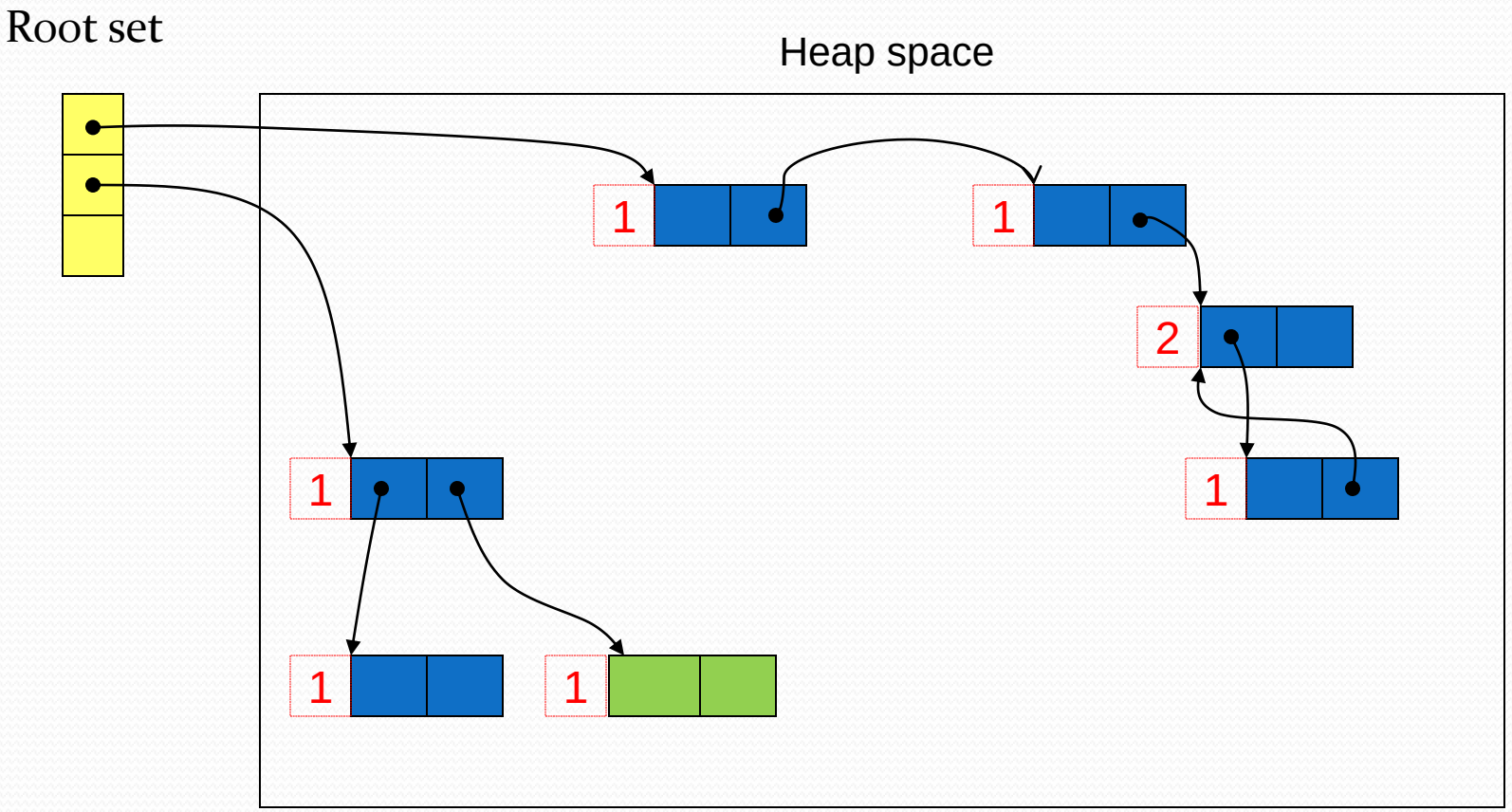
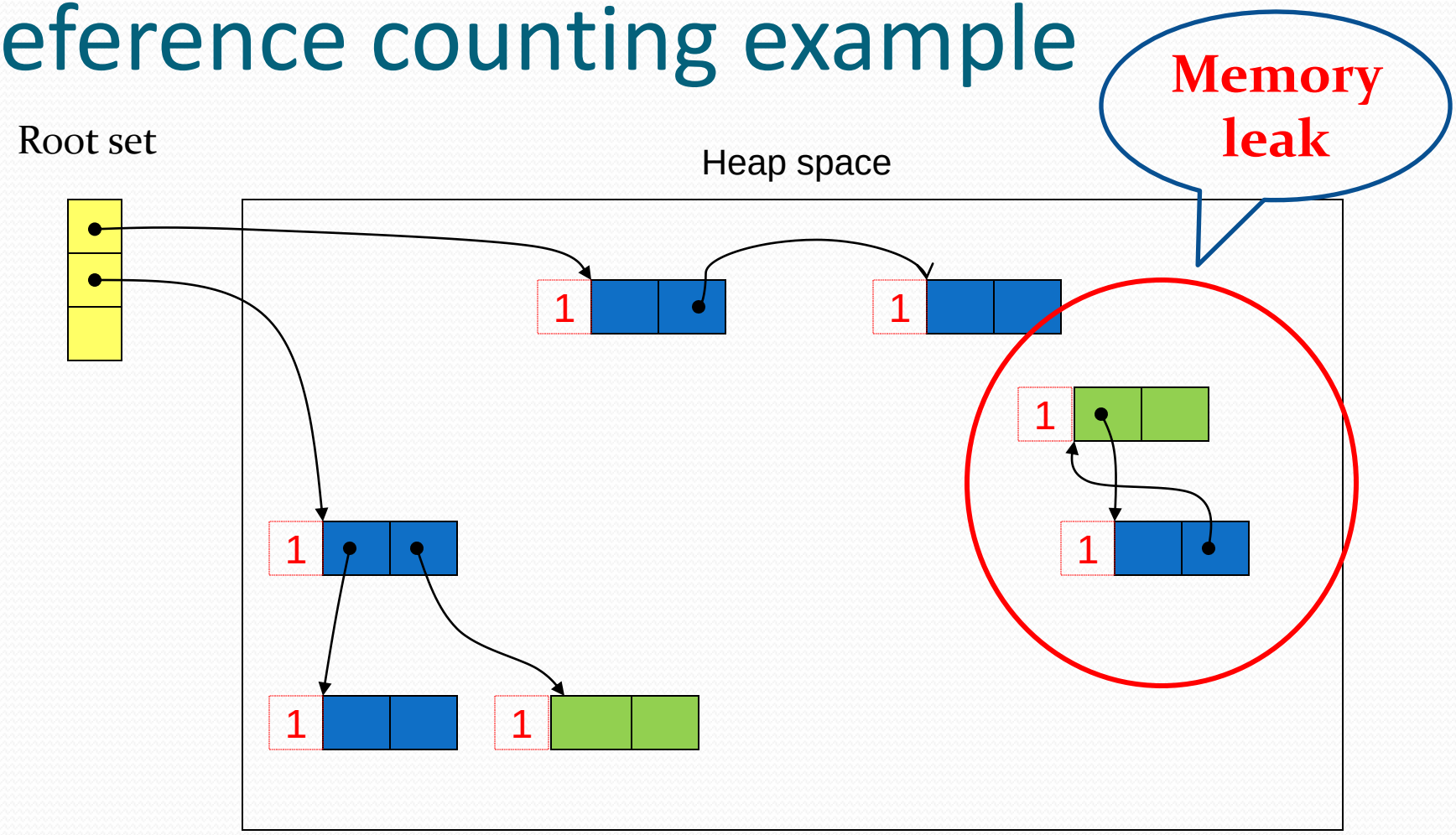
Reclaiming cyclic structures in RCGC

RCGC can be more attractive

Reference counting example



Reference counting example



Deficiencies of RCGC

- Cost of removing last pointer unbounded
- Total overhead of adjusting RCs significantly greater than that of tracing collectors
- Substantial space overhead
- Inability to reclaim cyclic data structures

How do we overcome shortcomings?

- Problem
  - **Inability to reclaim cyclic data structures**
    - RC of objects in cycle never get to zero
    - Cyclic data structures are common
      - At application level
        - Back pointers (e.g., doubly linked list)
        - Back edge in the link to a hash table chain
      - At system level
        - Functional languages use cycles to express recursion
    - Memory leak
- Solution
  - **Cyclic reference counting**

Functional programming languages

- Cycles created in well-defined manner
  - Treat specially
    - Created only by recursive definitions
    - References to such structures must follow these restrictions:
      - Circular structure created all at once
      - Use of proper subset that does not include root is copied as independent structure, not shared
      - Cycle-closing pointers to head of cycles are tagged
    - Ensure cycle is treated as single entity
    - Access to cycle is only through pointer to its root

Bobrow's technique (#2)

- Distinguish between pointers **internal** to the cycle from **external** references.
- External pointers counted as pointers to structure as a whole
- Internal pointers not counted
- Idea:
  - Collect groups of objects
  - Programmer assign objects to groups
  - Each group is reference counted
  - Group membership determined by object's address

Bobrow's technique

```
update(R, S){
    T = *R
    gr = group_no(R)
    if gr != group_no(S)                    // external reference
            increment_groupRC(S)
    if gr != group_no(T)                    // external reference
            decrement_groupRC(T)
            if groupRC(T) == 0
                    reclaim_group(T)
    *R = S
}
```

Weak pointer algorithms (#3)

- Distinguishing cycle closing pointers (*weak pointers*) from other references (*strong pointers*)
- Basis: Two invariants:
  - Each live object must be reachable from a root by a chain of strong pointers (*strongly reachable*)
  - Strong pointers must never be allowed to form cycles
  - Objects have 2 RC
    - Strong RC (SRC)
      - Pointers to new objects
    - Weak RC (WRC)
      - Closing link on pointer copy

Weak pointer algorithms

```
// Brownbridge's new
new() {
    if freeList == empty
            abort "Memory exhausted"
    newCell = allocate()
    SRC(newCell) = 1
    return strong(newCell)
}
```

```
//Salkild's update
update(R, S){
    WRC(S) = WRC(S) + 1
    delete(*R)
    *R = S
    weaken(*R)
}
```

Disadvantages of weak pointer alg.

- Cyclic structures can be incorrectly discarded
- Algorithm fails to terminate in some cases
  - A suicide pass searches for and breaks strong cycles
- Pathological case can lead to exponential time complexity in the worst case
- Space overhead is high: 2 RC fields

Hybrid algorithms (#4)

- Most objects freed by RC
  - Ideal candidates are uniquely referenced
- Cyclic structures freed by mark-sweep collector
  - Shared objects are cycle candidates
- Lin's algorithm (Lazy tracing of graphs)
  - Do not trace sub-graph every time shared pointer is deleted
  - Save values of deleted pointers in *control set*
    - Traps pointer writes
    - Uses extra field to colors objects
    - At suitable time search *control set* for garbage

Lin's Algorithm

Uses for colors for objects

Black:

- Active objects are painted black; including new objects

White:

- Garbage and free cells are painted white

Gray:

- Cells visited in marking phase are painted grey, have to be visited again

Purple:

- Cells that may be part of isolated cycles, have to be traversed by collector

Lin's algorithm

- When pointer to shared object deleted, object painted **purple**
  - Address put in *control set*
  - Avoids duplicate in *control set*
- New objects are allocated black
- Both arguments to *update()* must be removed from *control set* to prevent them from being mark-swept
  - They must be active
  - Painted **black**
  - *Control set* used to identify potential free space
    - Mark-sweep is used if picking object from it is still purple

Details of Lin's algorithm

```
// New objects allocated black
delete(T) {
    RC(T) = RC(T) – 1
    if RC(T) == 0
            color(T) = black
            for U in children(T)
                    delete(*U)
            free(T)
    else if color(T) != purple
            if control_set is full
                    gc_control_set()
            color(T) = purple
            push(T, control_set)
}
```

```
update(R, S){
    RC(S) = RC(S) + 1
    color(R) = black
    color(S) = black
    delete(*R)
    *R = S
}
```

Details of Lin's algorithm

- Discussion of mark-sweep algorithm in text
- Example helps explain algorithm
- Will differ discussion until we explore mark-sweep