Garbage Collection – overview of the three classical approaches

(based on chapter 2 of Jones and Lins)

Reference Counting

- a simple technique used in many systems
- eg, Unix uses it to keep track of when a file can be deleted (references to files come from directories)
- each object contains a counter which tracks the number of references to the object;
  if the count becomes zero, the storage of the object is immediately reclaimed (put into a free list?)
- distributes the cost of gc over the entire run of a program

Pseudocode for Reference Counting

```
// called by program to get a
// new object instance
function New():
  if freeList == null then
    report an error;
  newcell = allocate();
  newcell.rc = 1;
  return newcell;

// called by program to overwrite
// a pointer variable R with
// another pointer value S
procedure Update(var R, S):
  if S != null then
    S.rc += 1;
  delete(*R);
  *R = S;
```

```
// called by New
function allocate():
  newcell = freeList;
  freeList = freeList.next;
  return newcell;

// called by Update
procedure delete(T):
  T.rc -= 1;
  if T.rc == 0 then
    foreach pointer U held
        inside object T do
      delete(*U);
    free(T);

// called by delete
procedure free(N):
  N.next = freeList;
  freeList = N;
```

`rc` is the reference count field in the object

Benefits of Reference Counting

- GC overhead is distributed throughout the computation ==> smooth response times in interactive situations.
  (Contrast with a stop and collect approach.)

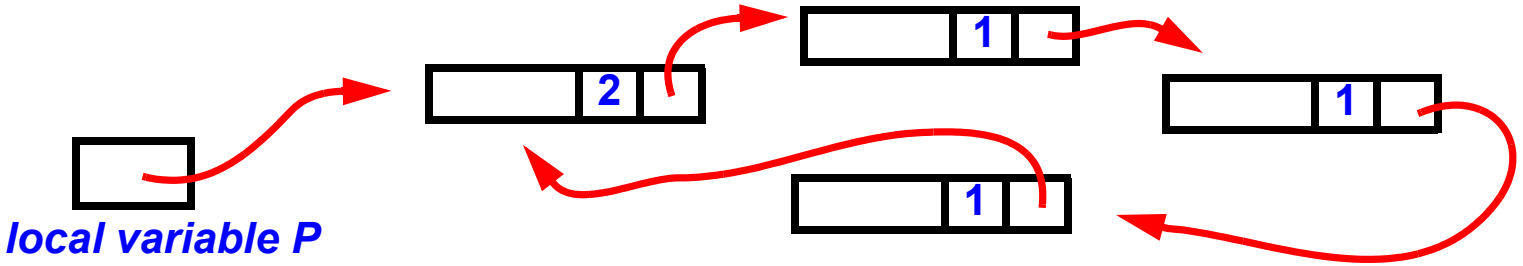
- Good memory locality 1 – the program accesses memory locations which were probably going to be touched anyway.
  (Contrast with a marking phase which walks all over memory.)

- Good memory locality 2 – most objects are short-lived; reference counting will reclaim them and reuse them quickly.
  (Contrast with a scheme where the dead objects remain unused for a long period until the next gc and get paged out of memory.)

Issues with Reference Counting, cont'd

- Extra storage requirements
  - Every object must contain an extra field for the reference counter. (And how big should it be?)

- **Does not work with cyclic data structures!!!**



Mark-Sweep (aka Mark-Scan) Algorithm

- First use seems to be Lisp
- Storage for new objects is obtained from a free pool
- No extra actions are performed when the program copies or overwrites pointers
- When the free pool is exhausted, the `New()` operation invokes the mark-sweep gc to return inaccessible objects to the free pool and then resumes

Pseudocode for Mark-Sweep

```
function New():
  if freeList == null then
    markSweep();
  newcell = allocate();
  return newcell;

// called by New
function allocate():
  newcell = freeList;
  freeList = freeList.next;
  return newcell;

procedure free(P):
  P.next = freeList;
  freeList = P;

procedure markSweep():
  foreach R in RootSet do
    mark(R);
  sweep();
  if freeList == null then
    abort "memory exhausted"
```

```
// called by markSweep
procedure mark(N):
  if N.markBit == 0 then
    N.markBit = 1;
    foreach pointer M held
        inside the object N do
      mark(*M);

// called by markSweep
procedure sweep():
  K = address of heap bottom;
  while K < heap top do
    if K.markBit == 0 then
      free(K);
    else
      K.markBit = 0;
    K += size of object
        referenced by K;
```

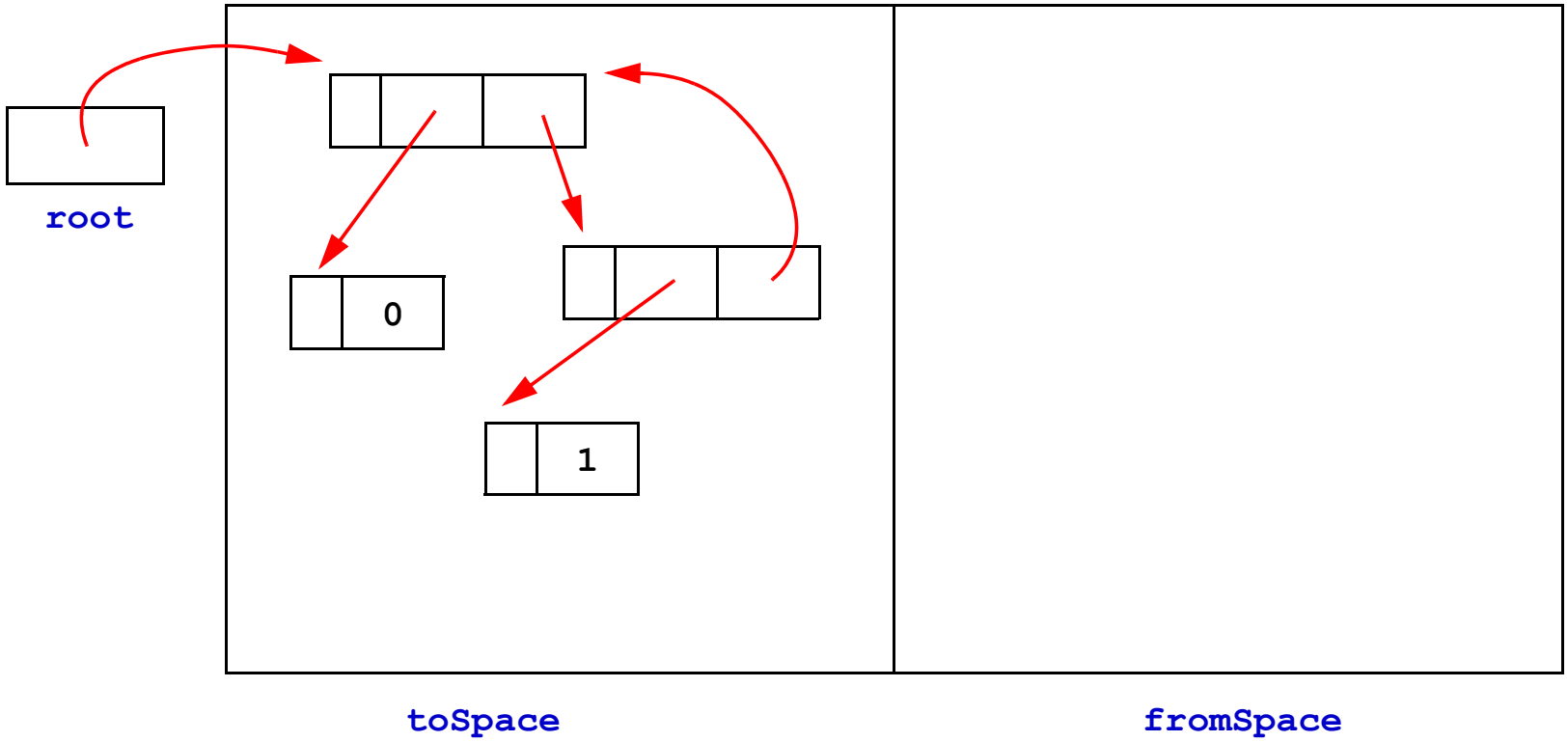
Pros and Cons of Mark-Sweep GC

- Cycles are handled automatically
- No special actions required when manipulating pointers
- It's a stop-start approach – in the 1980's, Lisp users got interrupted for about 4.5 seconds every 79 seconds.
- Less *total* work performed than reference counting.
- Tends to fragment memory, scattering elements of linked lists all across the heap
- Performance degrades as the heap fills up with active cells (causing more frequent gc)

Copying Garbage Collectors

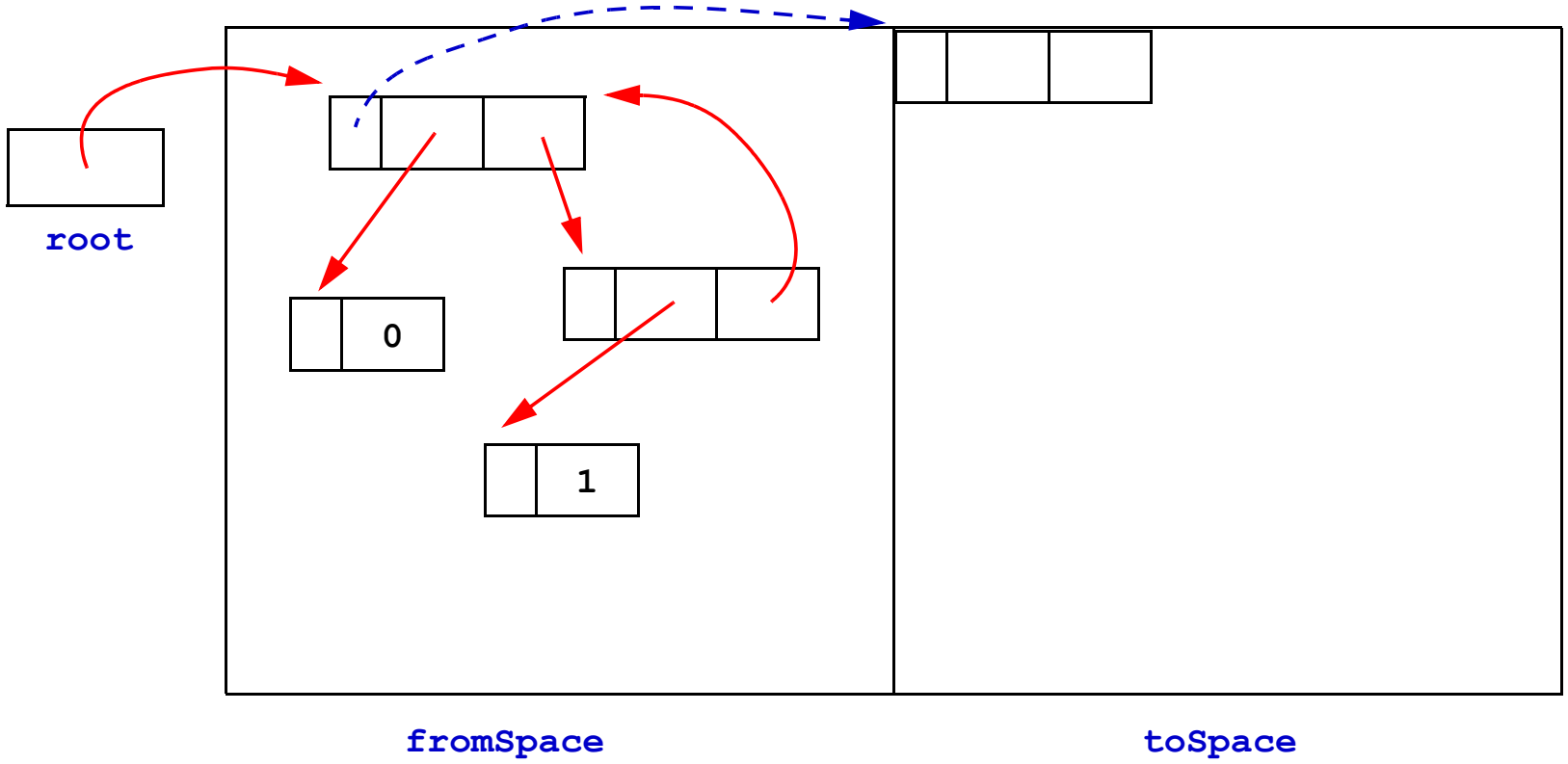
- The heap is divided into two equal sized regions – the *fromSpace* and the *toSpace.*
- The roles of the two spaces are reversed at each gc.
- At a gc, the active cells are copied from the old space (the *fromSpace*) into the new space (the *toSpace*), and the program's variables are updated to use the new copies.
- Garbage cells in the fromSpace are simply abandoned.
- Storage in the toSpace is automatically compacted during the copying process (no gaps are left).

Example of Copying Collector in Action



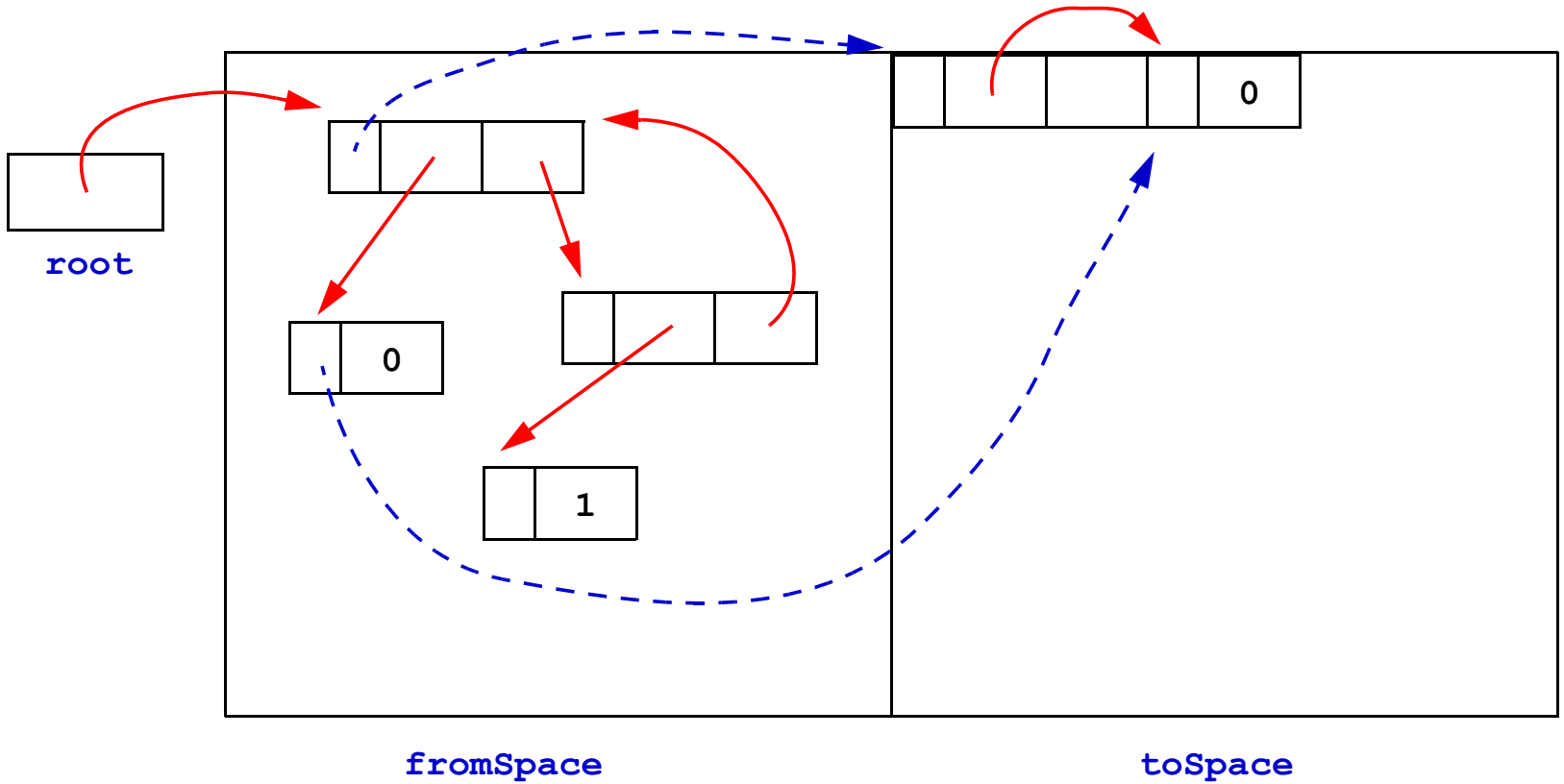
1. a gc is initiated; the fromSpace & toSpace are swapped ...

Example of Copying Collector in Action



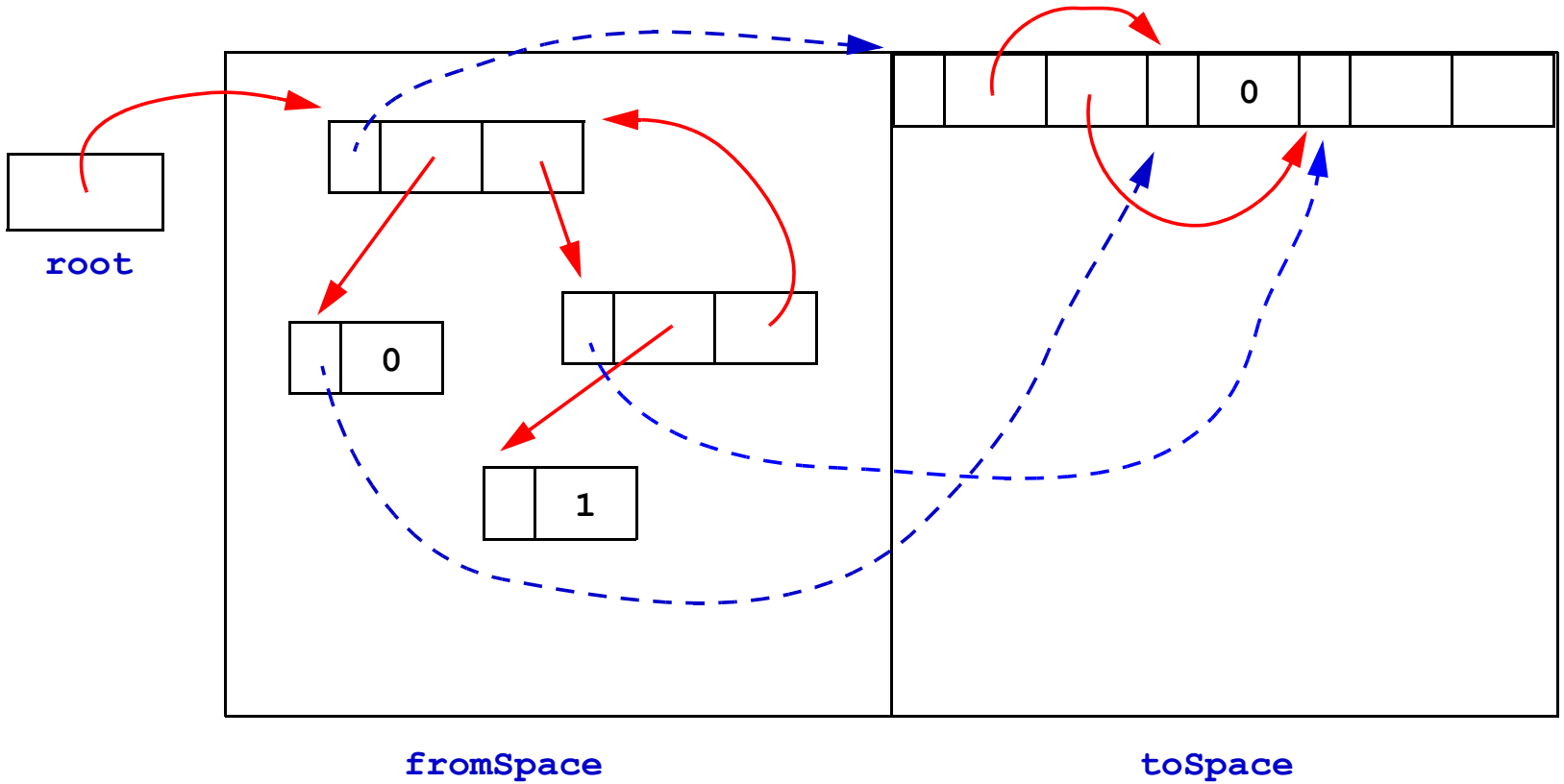
... the root node is copied, and a forwarding pointer added

Example of Copying Collector in Action



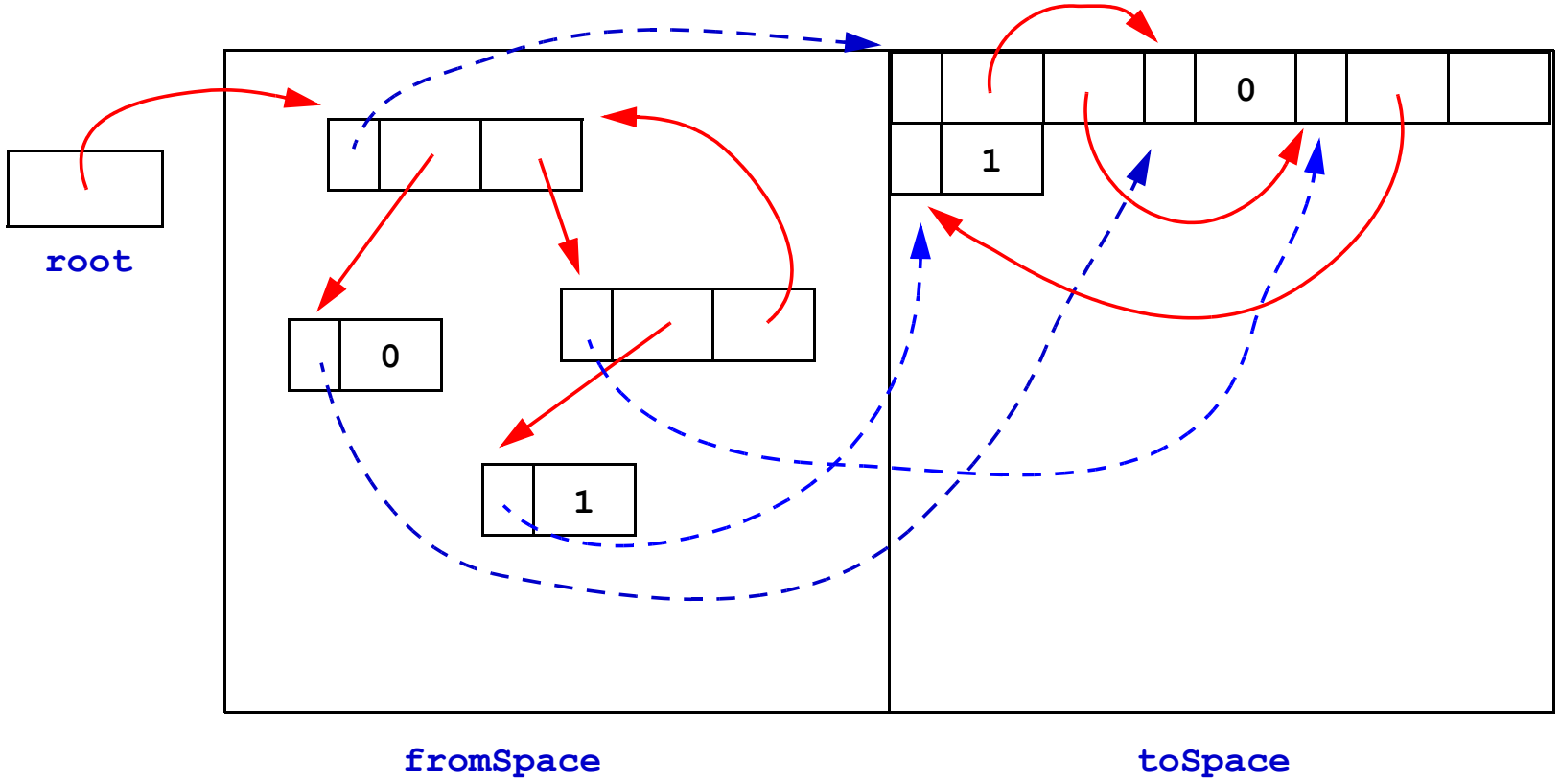
... the left child of first node is copied

Example of Copying Collector in Action



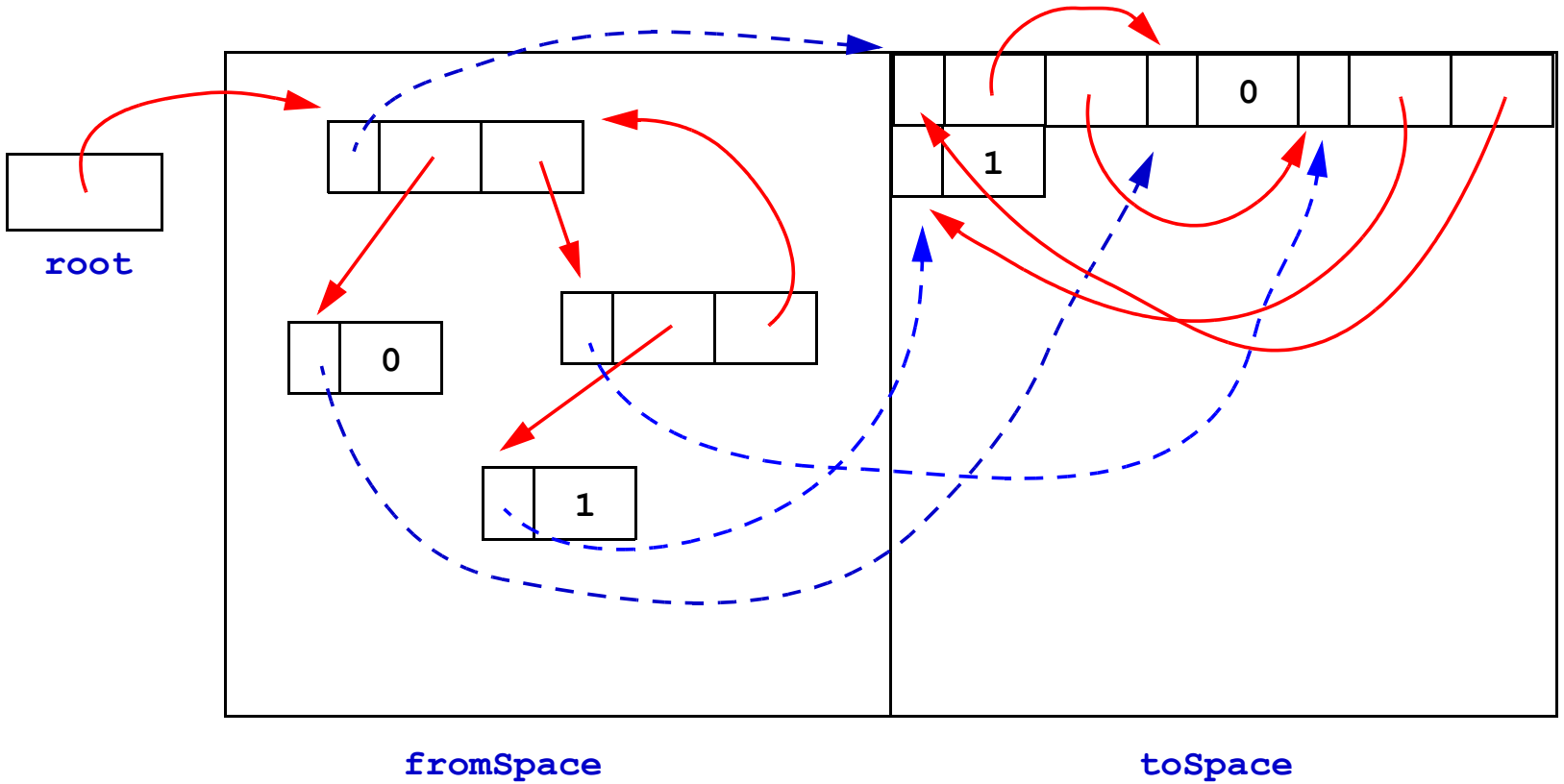
... and the right child of the first node is copied

Example of Copying Collector in Action



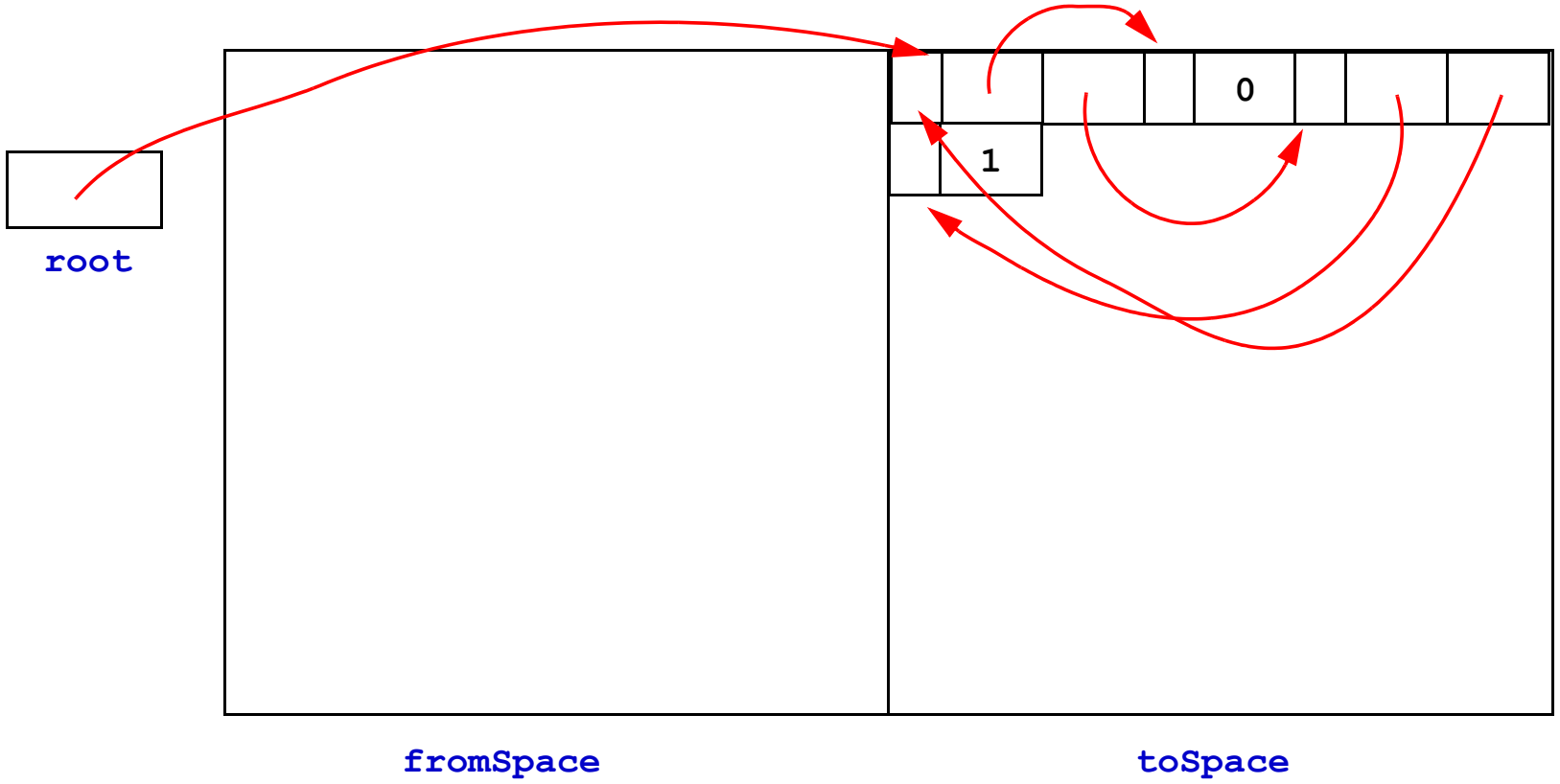
... and when the right child of the right child is copied ...

Example of Copying Collector in Action



... and we are almost finished

Example of Copying Collector in Action



done ... and we carry on allocating new nodes in the toSpace

Pseudocode for a Copying Collector

```
procedure init():
  toSpace = start of heap;
  spaceSize = heap size / 2;
  topOfSpace =toSpace+spaceSize;
  fromSpace = topOfSpace+1;
  free = toSpace;

// n = size of object to allocate
function New(n):
  if free + n > topOfSpace then
    flip();
  if free + n > topOfSpace then
    abort "memory exhausted";
  newcell = free;
  free += n;
  return newcell;

procedure flip():
  fromSpace, toSpace =
        toSpace, fromSpace;
  free = toSpace;
  for R in RootSet do
    R = copy(R);
```

```
// parameter P points to a word,
// not to an object
function copy(P):
  if P is not a pointer
       or P == null then
    return P;
  if P[0] is not a pointer
       into toSpace then
    n = size of object
         referenced by P;
    PP = free;
    free += n;
    temp = P[0];
    P[0] = PP;
    PP[0] = copy(temp);
    for i = 0 to n-1 do
      PP[i] = copy(P[i]);
  return P[0];

// Note:
// The first word of an object,
// P[0], serves a dual role to
// hold a forwarding pointer.
```

Pros and Cons of Copying Collectors

- Very cheap allocation cost (just incrementing a pointer)
- Fragmentation of memory is eliminated at each gc

- At any time, at least 50% of the heap is unused
  (may not be a problem with virtual memory systems where we can have big address spaces)