Vrije Universiteit Brussel

Software Languages Lab

Programming Language Engineering
Master of Computer Science

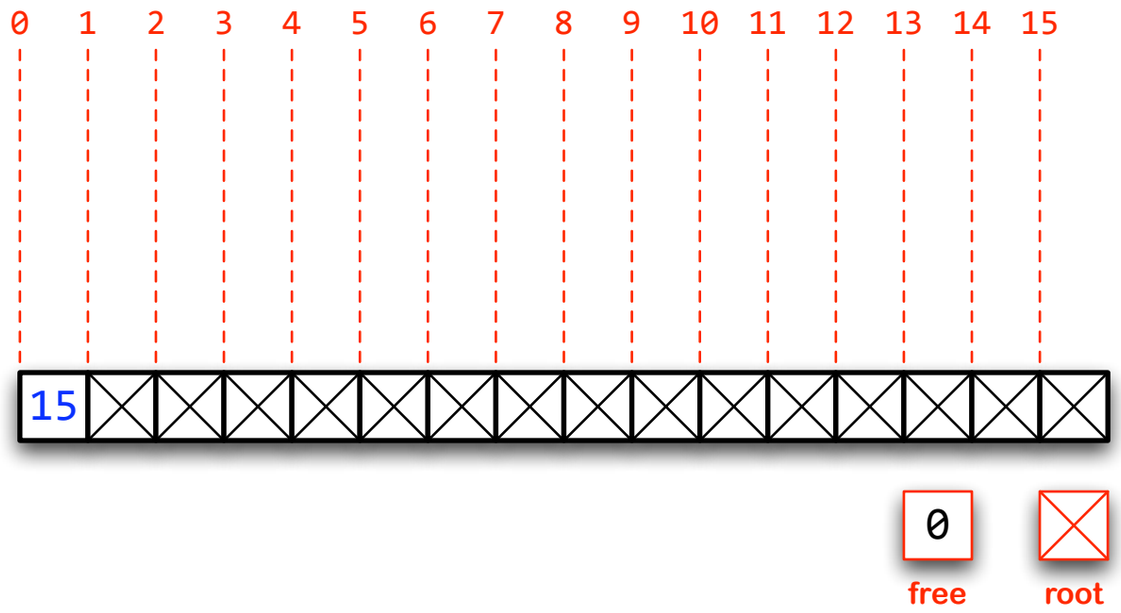
Faculty of Science and Bio-Engineering Sciences
Vrije Universiteit Brussel

Section 3: Garbage Collection

Theo D'Hondt
Software Languages Lab

"... Interpreters are critical because they reveal nuances of meaning, and are the direct path to more efficient compilation and to other kinds of program analyses..."

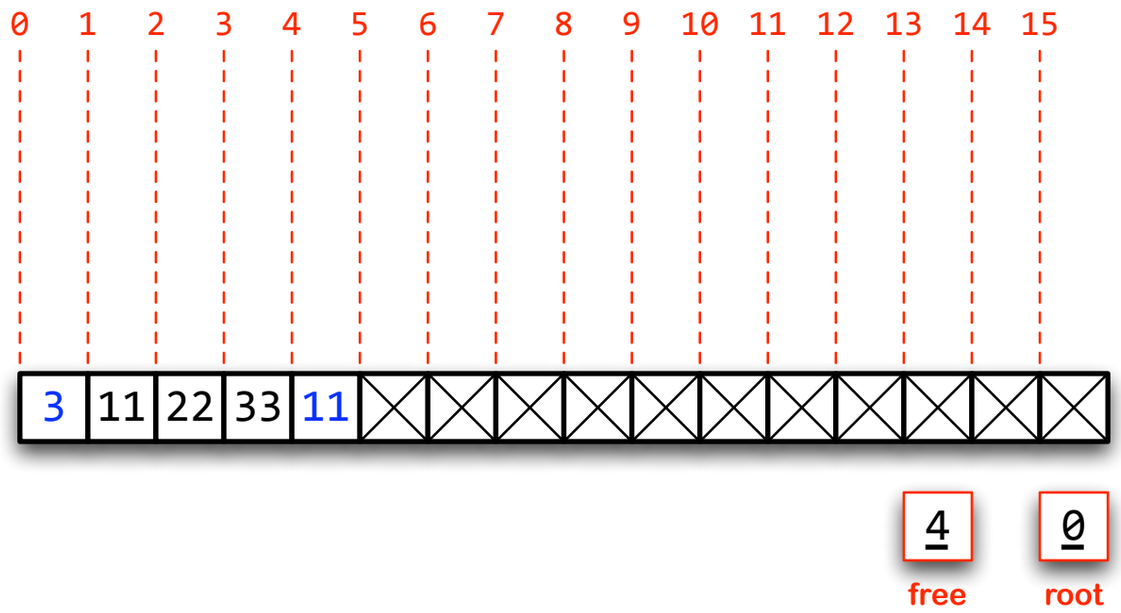
Memory Layout



16	size	free
5	atom	used
0	pointer	void

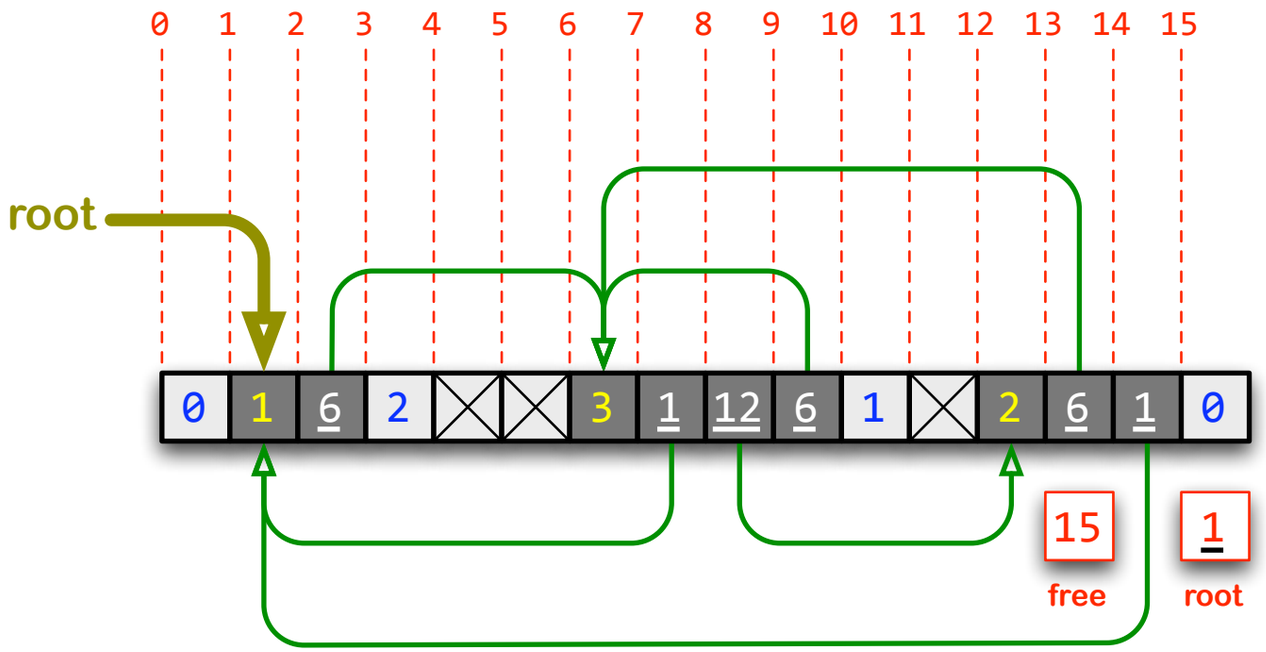
```
(let*
 ((SIZE 16)
  (BIAS 100)
  (VOID 999)
  (FREE 0))
 (define ROOT VOID)
 (define memory (make-vector SIZE VOID))
 (define (number-to-chunk number)
   (+ BIAS number))
 (define (chunk-to-number chunk)
   (- chunk BIAS))
 (define (chunk? any)
   (and (>= any BIAS) (< any VOID)))
 (define (number? any)
   (and (>= any 0) (< any BIAS)))
 (define (void? any)
   (= any VOID))
 (vector-set! memory 0 (- SIZE 1))
 (display memory))

#(15 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999)
<undefined>
```

Allocating Memory



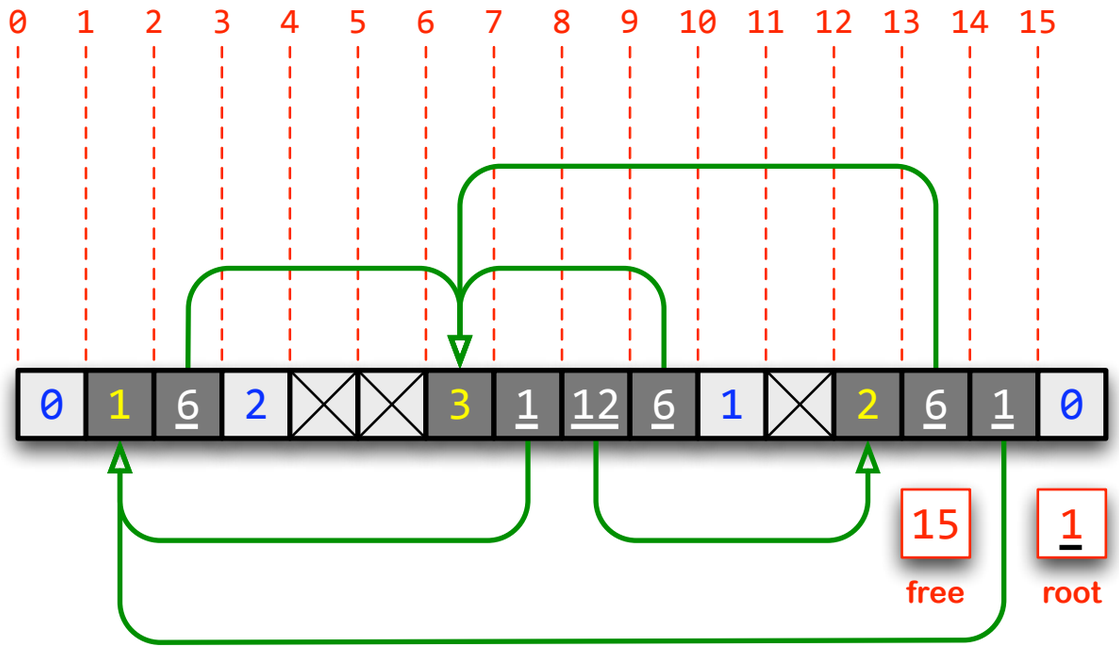

```
...
  (define (make-chunk size)
    (define total (vector-ref memory FREE))
    (define chunk (number-to-chunk FREE))
    (vector-set! memory FREE size)
    (set! FREE (+ FREE size 1))
    (vector-set! memory FREE (- total size 1))
    chunk)
  (define (chunk-set! chunk offset value)
    (define index (chunk-to-number chunk))
    (vector-set! memory (+ index offset 1) value))
  (define (chunk-ref chunk offset)
    (define index (chunk-to-number chunk))
    (vector-ref memory (+ index offset 1)))
  (vector-set! memory 0 (- SIZE 1))
  (let ((r (make-chunk 3)))
    (chunk-set! r 0 11)
    (chunk-set! r 1 22)
    (chunk-set! r 2 33))
  (display memory))

#(3 11 22 33 11 999 999 999 999 999 999 999 999 999 999 999)
<undefined>
```

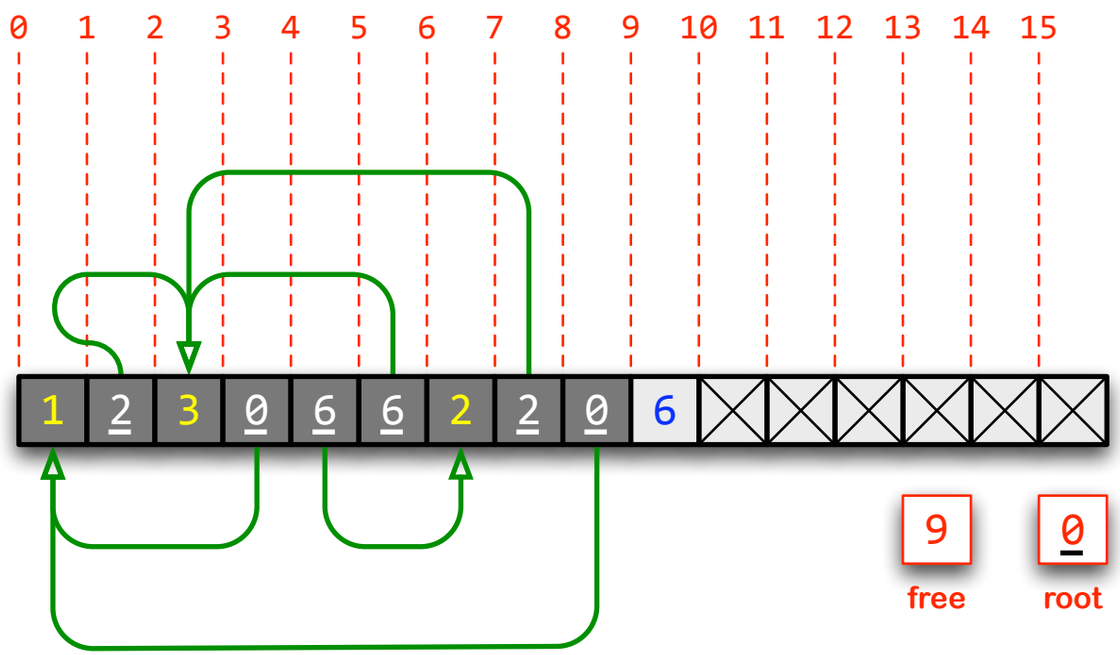
Establishing a Root




```
...
  (define (set-root chunk)
    (set! ROOT chunk))
  (vector-set! memory 0 (- SIZE 1))

  (make-chunk 0)
  (let ((r1 (make-chunk 1)))
    (make-chunk 2)
    (let ((r2 (make-chunk 3)))
      (make-chunk 1)
      (let ((r3 (make-chunk 2)))
        (chunk-set! r1 0 r2)
        (chunk-set! r2 0 r1)
        (chunk-set! r2 1 r3)
        (chunk-set! r2 2 r2)
        (chunk-set! r3 0 r2)
        (chunk-set! r3 1 r1)
        (set-root r1))))
  (display memory))

#(0 1 106 2 999 999 3 101 112 106 1 999 2 106 101 0)
<undefined>
```

Recovering Memory





Recovering Memory

- Identify memory in use
- Anticipate storage move
- Effectively crunch memory

Recovering Memory

- Identify memory in use
- Anticipate storage move
- Effectively crunch memory

Sweep and mark from the root

Recovering Memory

- Identify memory in use
- Anticipate storage move
- Effectively crunch memory

Sequentially sweep memory and reverse pointers

Recovering Memory

- Identify memory in use
- Anticipate storage move
- Effectively crunch memory

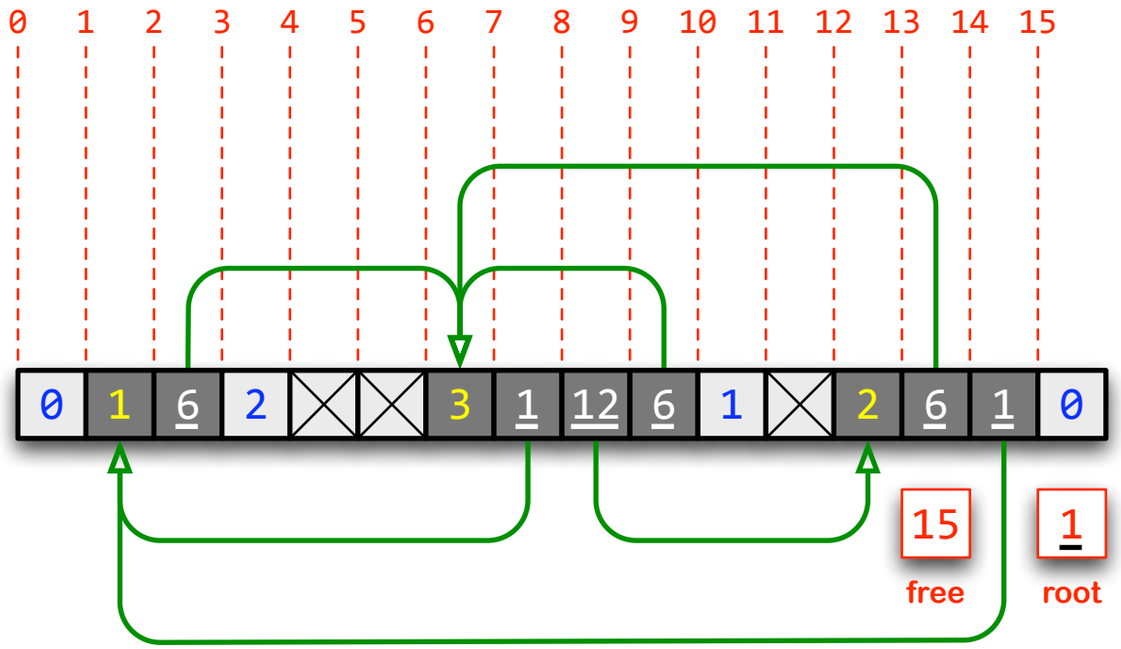
Sequentially sweep memory and move chunks

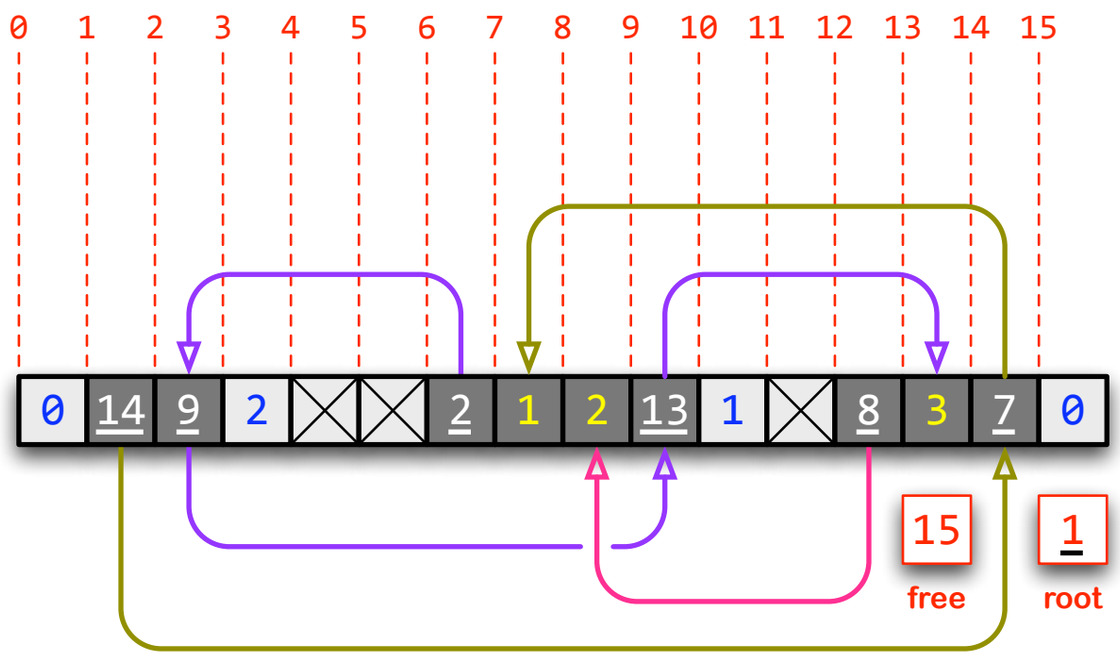
Recovering Memory

- Identify memory in use
- Anticipate storage move
- Effectively crunch memory

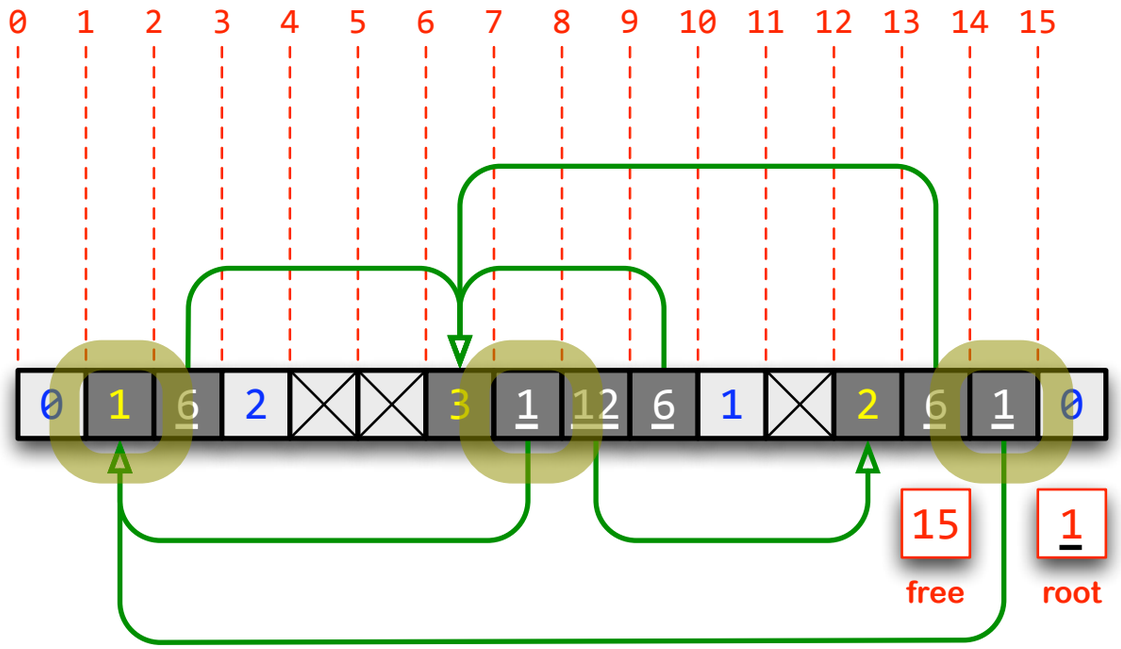
Compacting
mark-and-
sweep

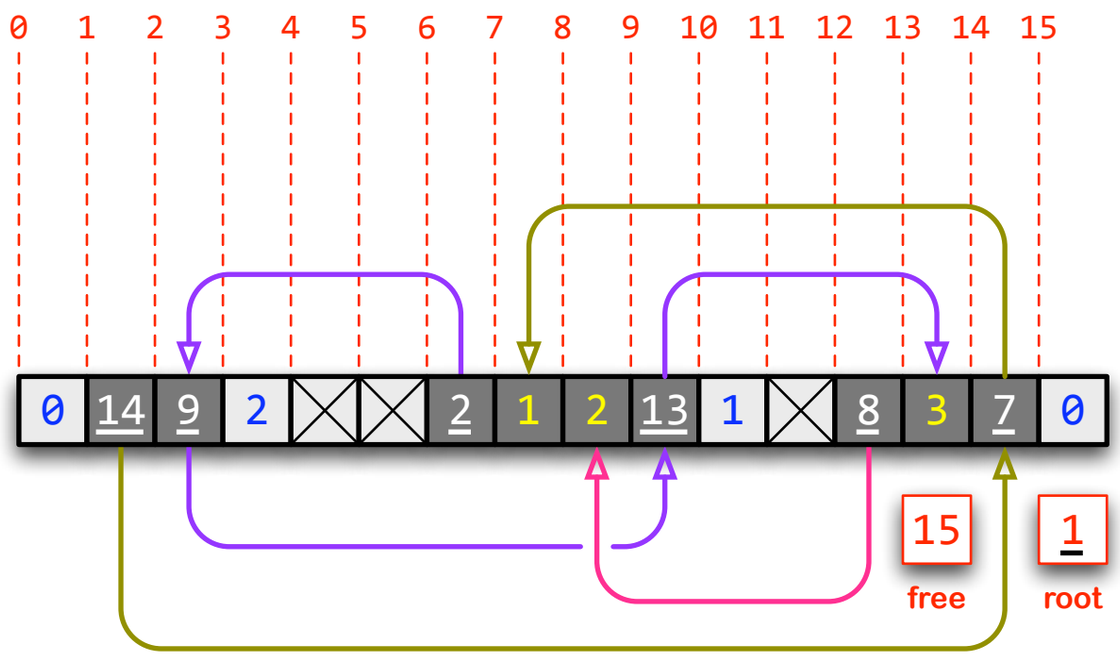
Traverse and Invert Memory



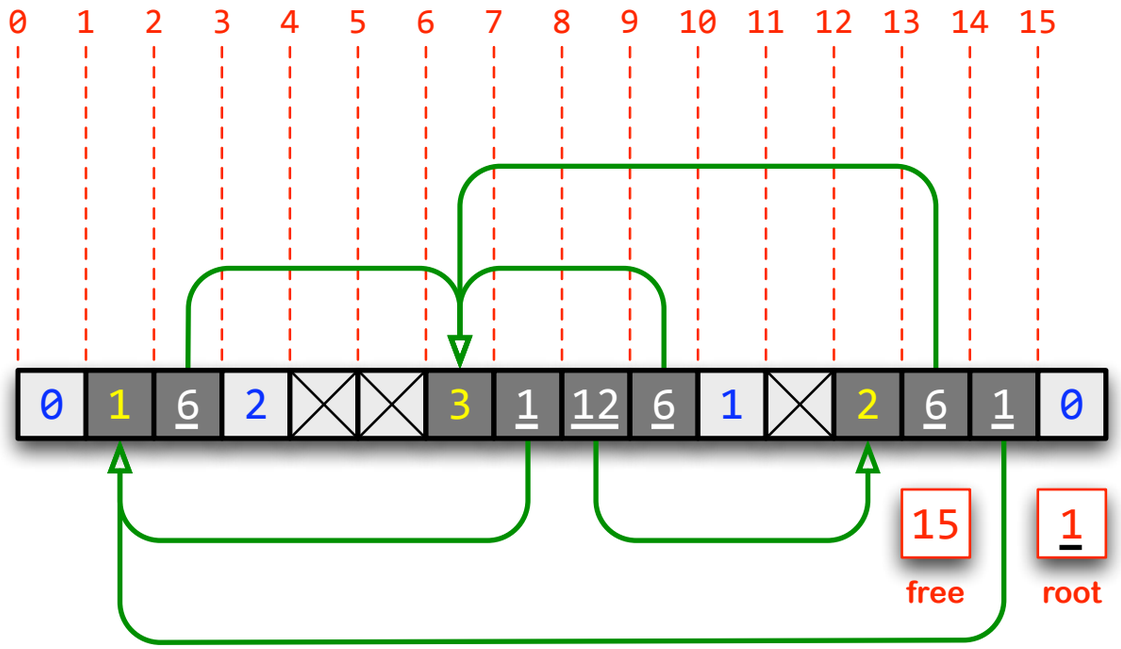


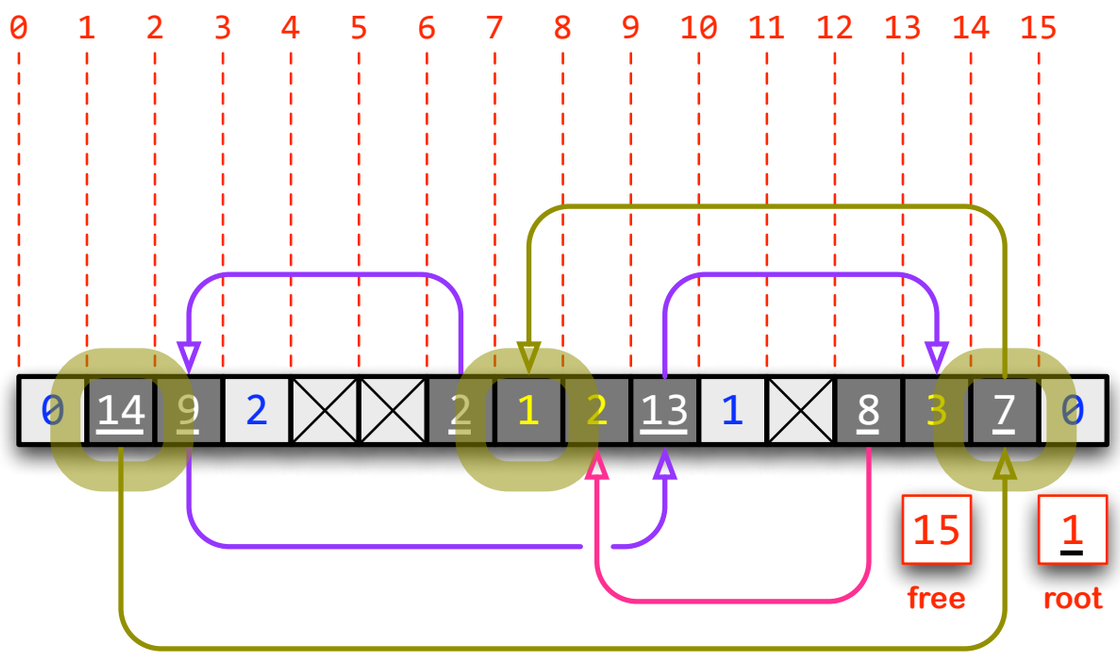
Traverse and Invert Memory



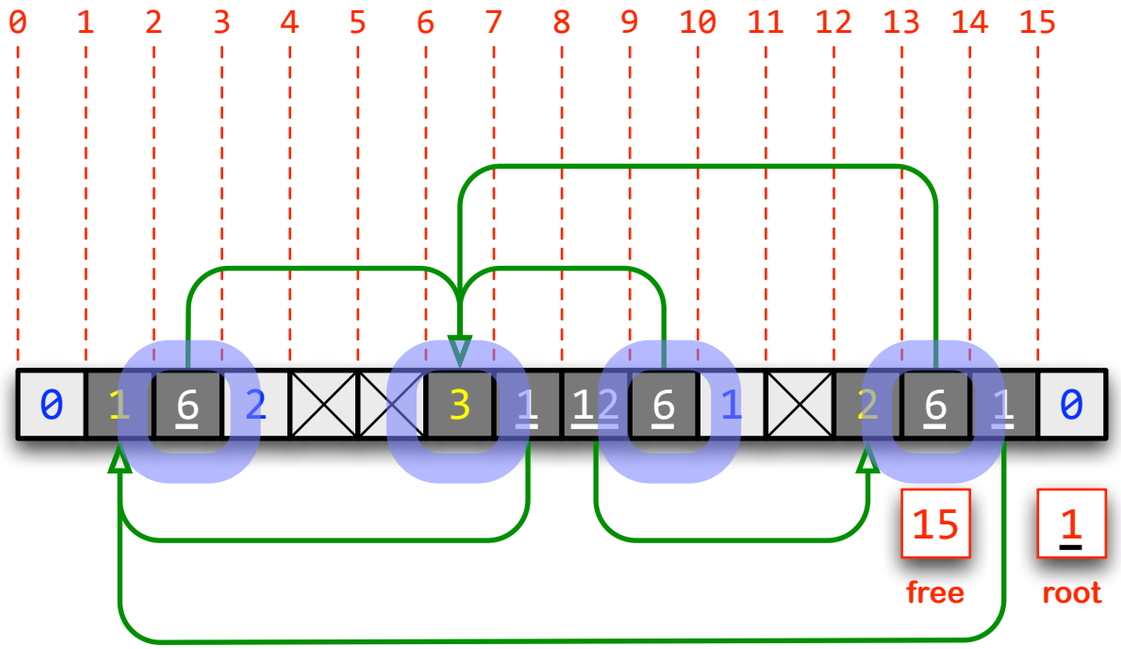


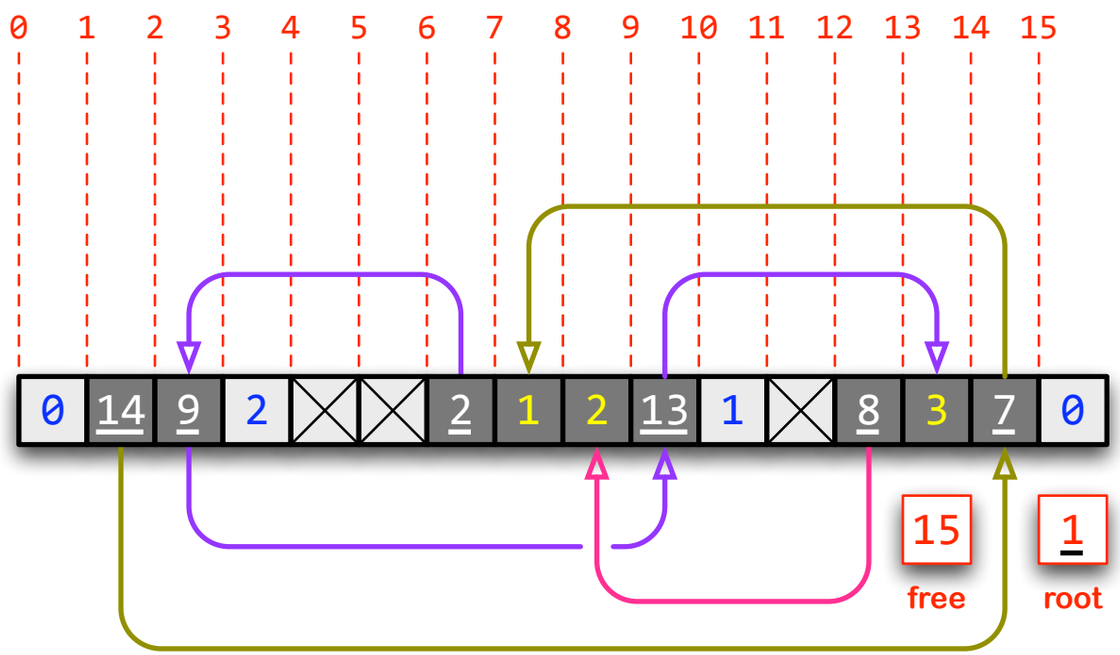
Traverse and Invert Memory



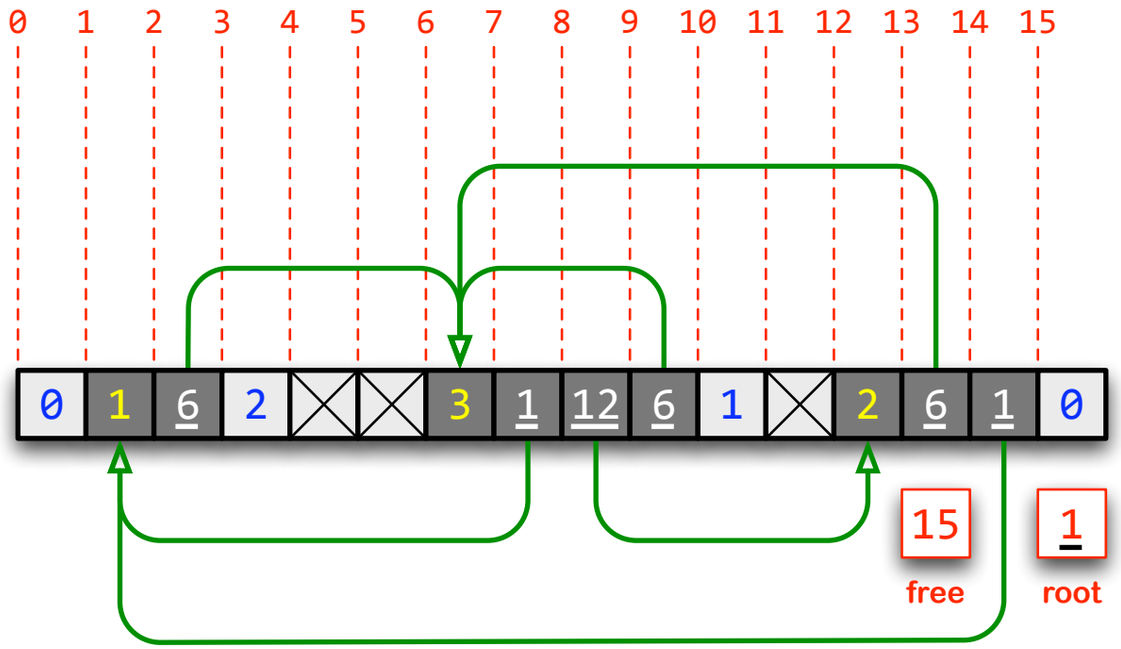


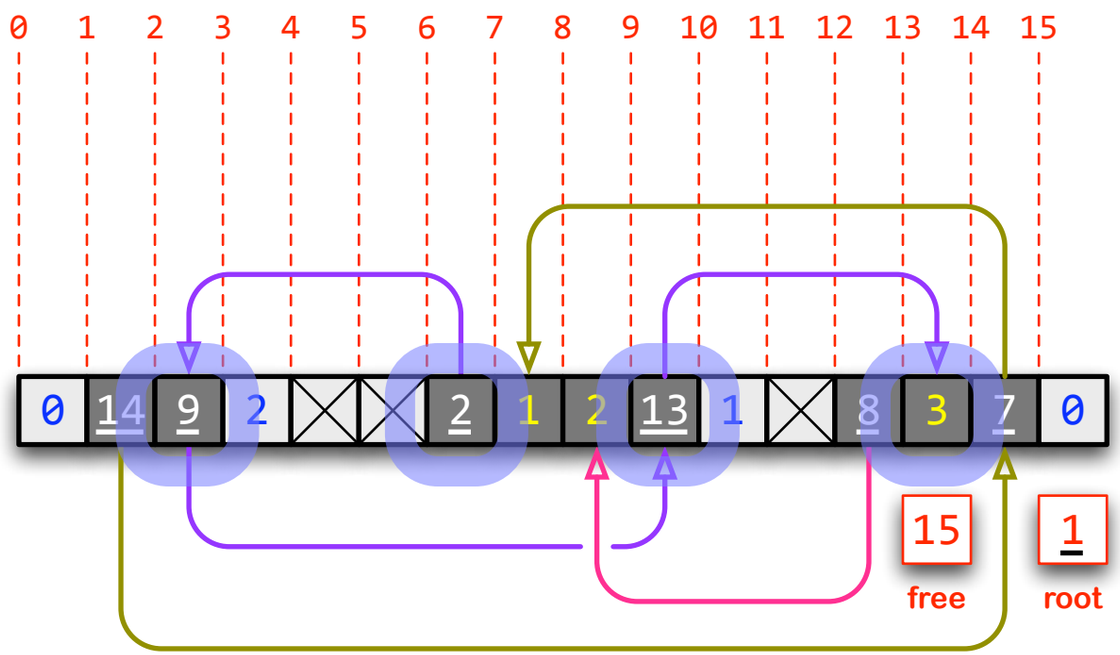
Traverse and Invert Memory



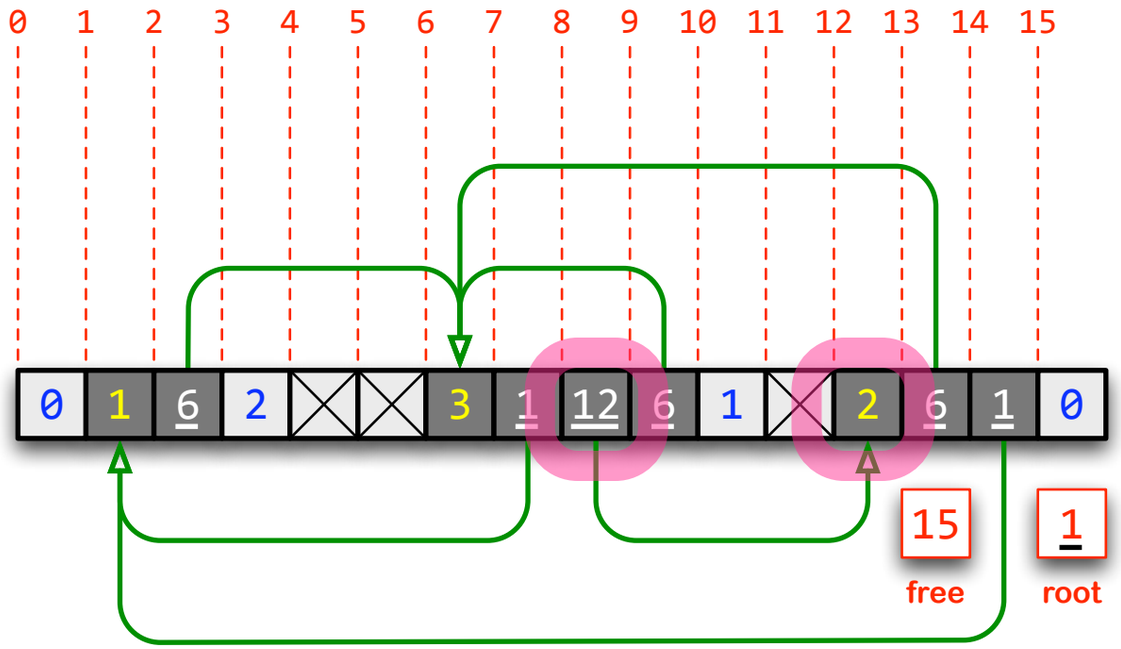


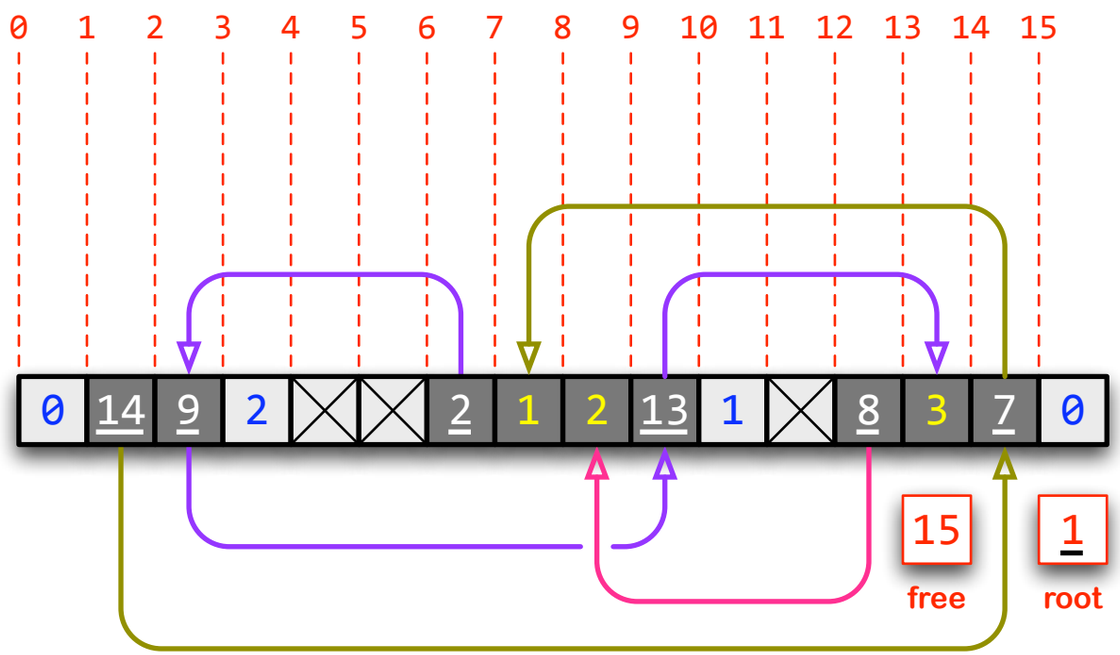
Traverse and Invert Memory



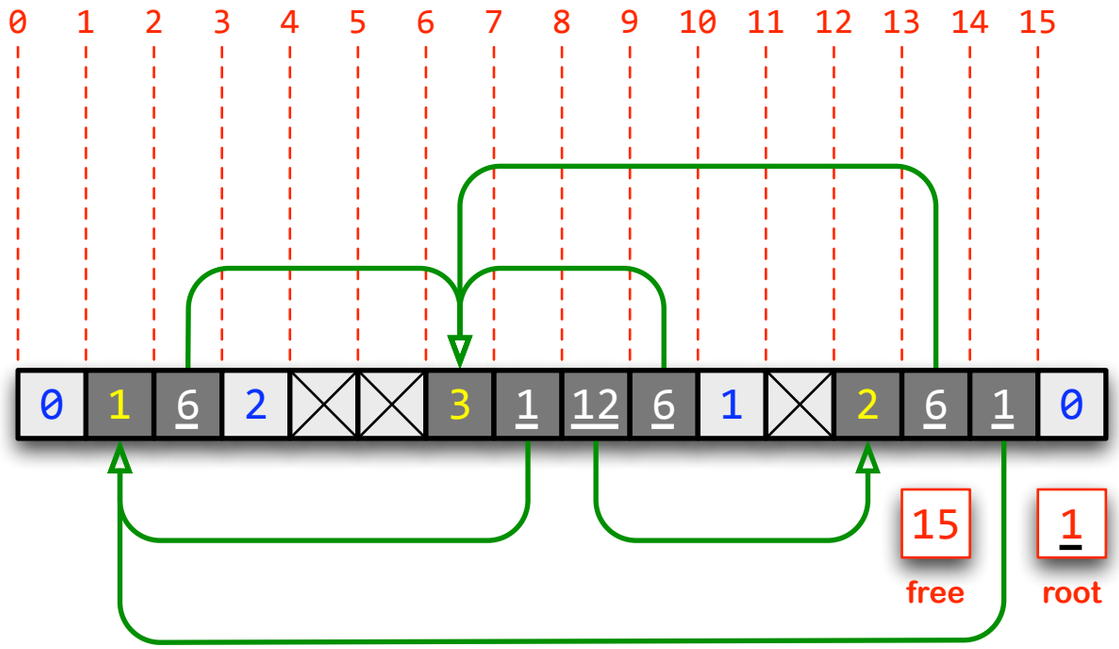


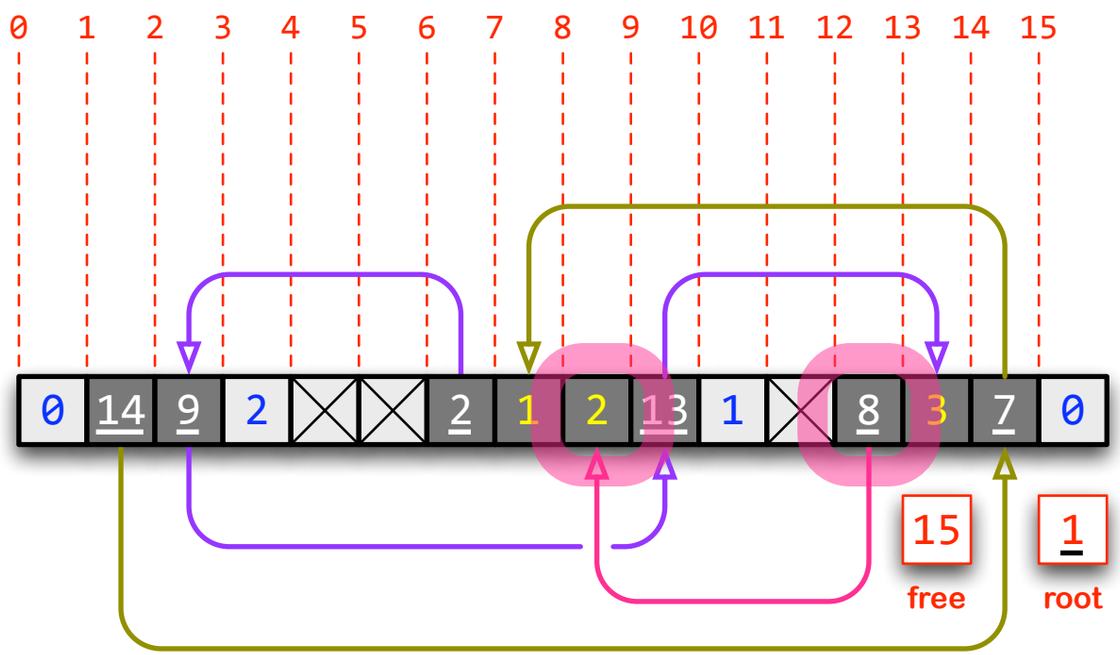
Traverse and Invert Memory



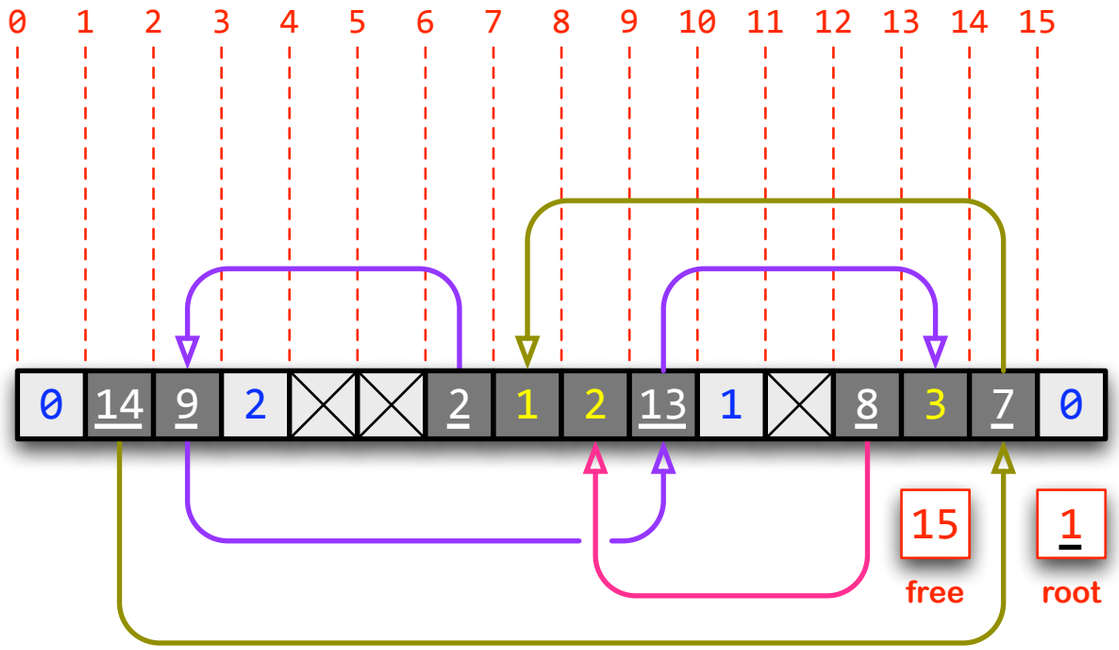


Traverse and Invert Memory



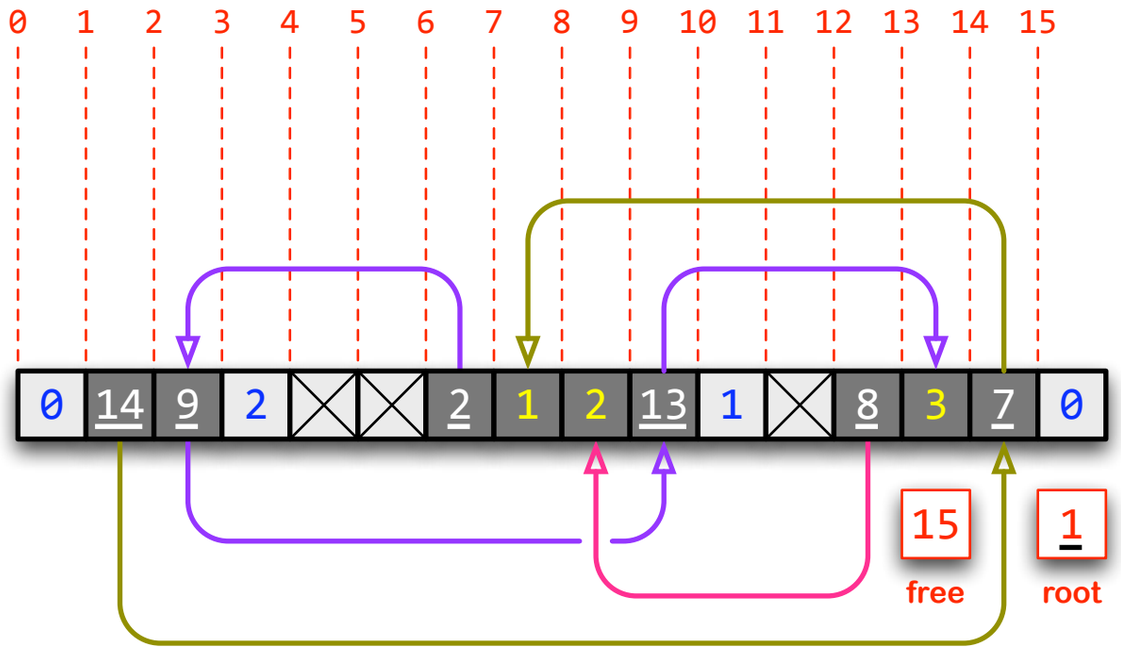


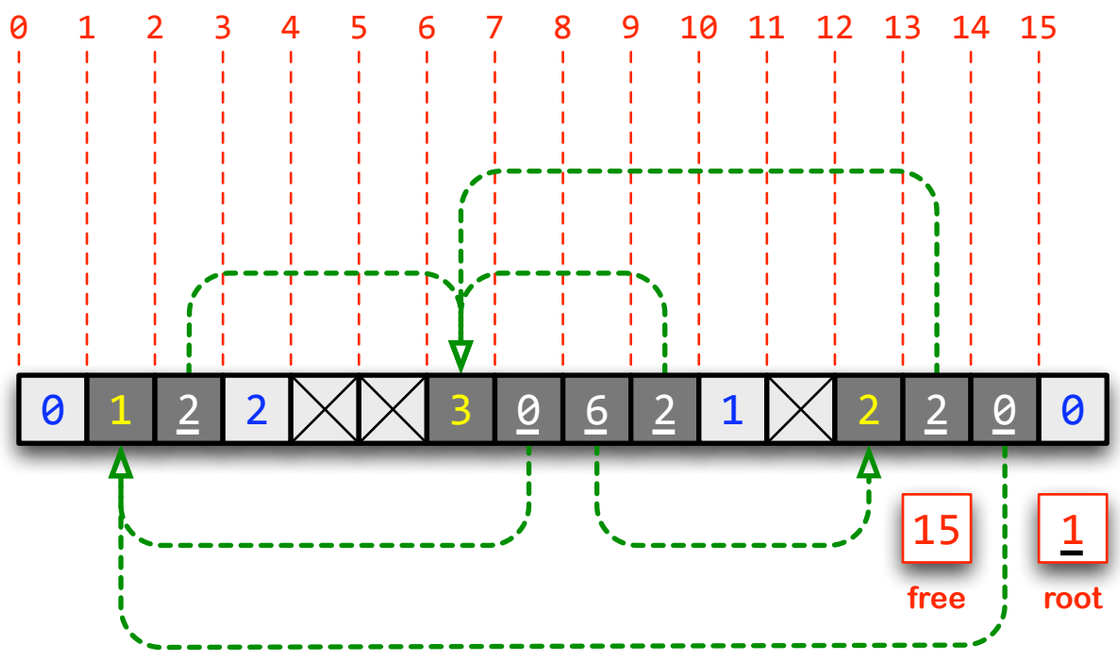
Traverse and Invert Memory



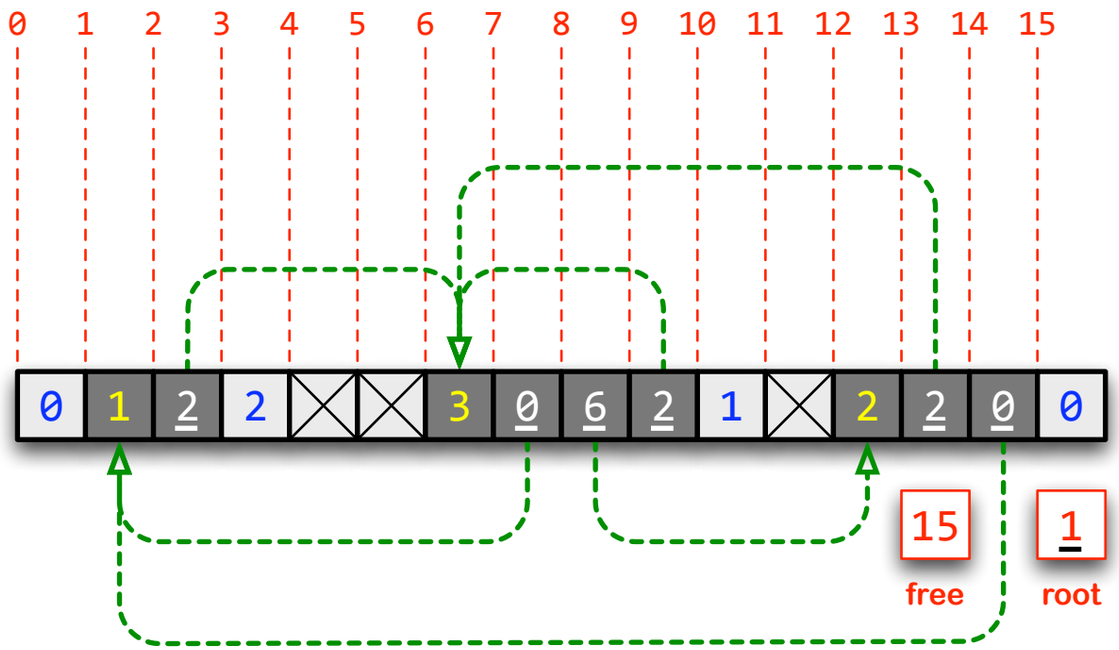

```
...
   (MARK 100)
...
  (define (chunk-size-ref chunk)
    (define index (chunk-to-number chunk))
    (vector-ref memory index))
  (define (chunk-size-set! chunk size)
    (define index (chunk-to-number chunk))
    (vector-set! memory index size))
  (define (set-root chunk)
    (set! ROOT chunk))
  (define (unmarked? size)
    (not (negative? size)))
...

  (define (sweep)
    (define (traverse chunk)
      (define size (chunk-size-ref chunk))
      (cond
        ((number? size)
          (chunk-size-set! chunk (- size MARK))
          (do ((index 0 (+ index 1))) ((= index size))
            (let ((cell (chunk-ref chunk index)))
              (if (chunk? cell)
                  (let* ((offset  (chunk-to-number chunk))
                         (pointer (number-to-chunk
                                    (+ offset index 1))))
                    (traverse cell)
                    (chunk-set! chunk index (chunk-size-ref cell))
                    (chunk-size-set! cell pointer))))))))
      (if (chunk? ROOT)
        (traverse ROOT)))
...
  (display memory)
  (newline)
  (sweep)
  (display memory))

#(0 1 106 2 999 999 3 101 112 106 1 999 2 106 101 0)
#(0 114 109 2 999 999 102 -99 -98 113 1 999 108 -97 107 0)
<undefined>
```

Update Memory



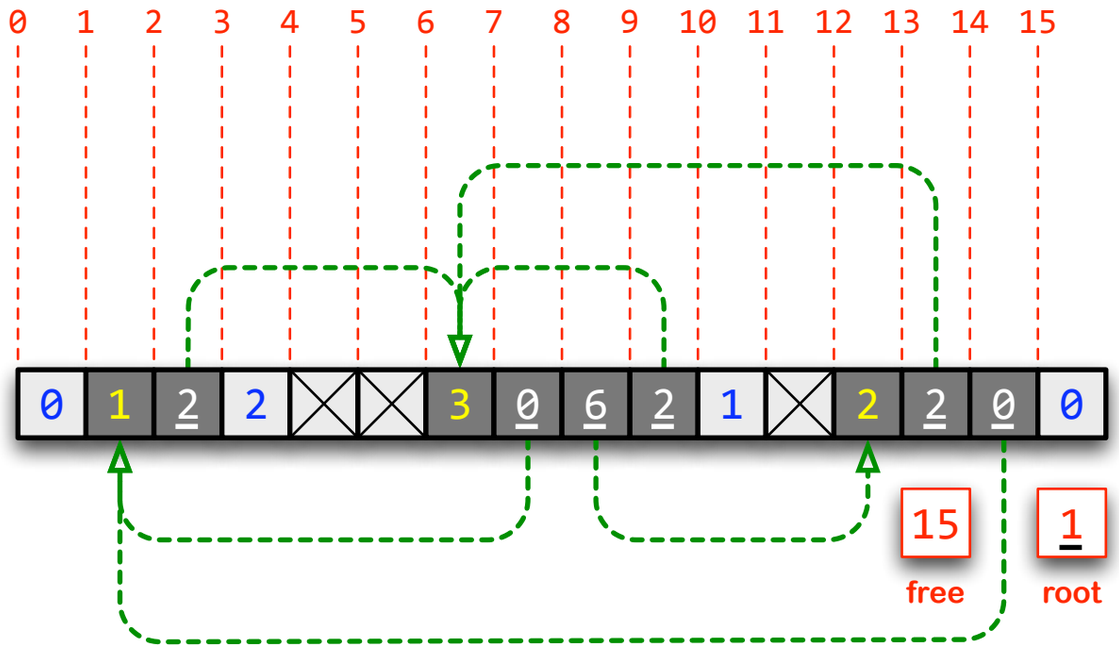


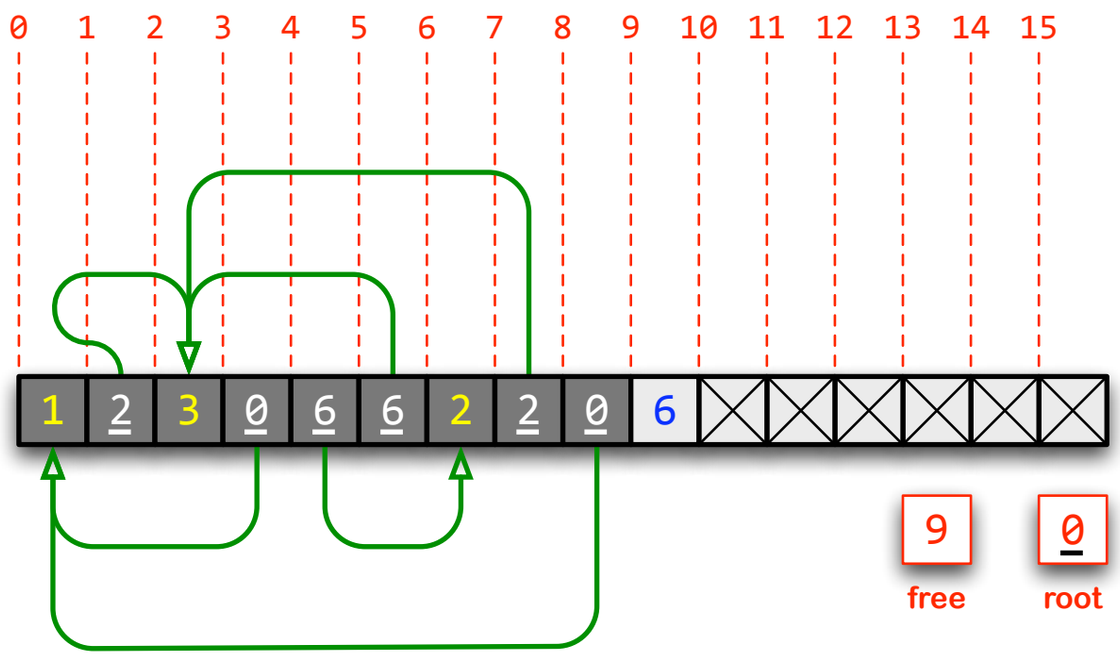
Update Memory



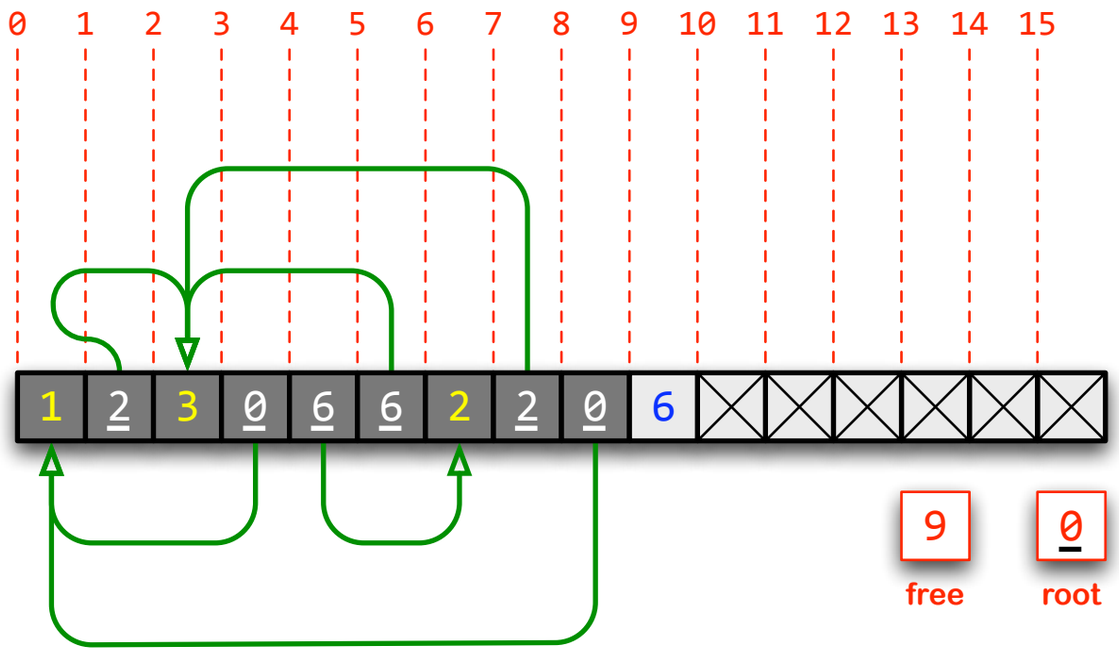

```
...

  (define (update)
    (do ((source 0)
         (destination 0))
        ((= source FREE))
      (let ((siz (vector-ref memory source))
            (new (number-to-chunk destination)))
        (do () ((not (chunk? siz)))
          (let ((index (chunk-to-number siz)))
            (set! siz (vector-ref memory index))
            (vector-set! memory index new)))
        (vector-set! memory source siz)
        (if (negative? siz)
          (let ((siz (+ siz MARK)))
            (set! destination (+ destination siz 1))
            (set! source (+ source siz 1)))
          (set! source (+ source siz 1))))))

...

  (display memory)
  (newline)
  (sweep)
  (display memory)
  (newline)
  (update)
  (display memory))

#(0 1 106 2 999 999 3 101 112 106 1 999 2 106 101 0)
#(0 114 109 2 999 999 102 -99 -98 113 1 999 108 -97 107 0)
#(0 -99 102 2 999 999 -97 100 106 102 1 999 -98 102 100 0)
<undefined>
```

Crunch Memory



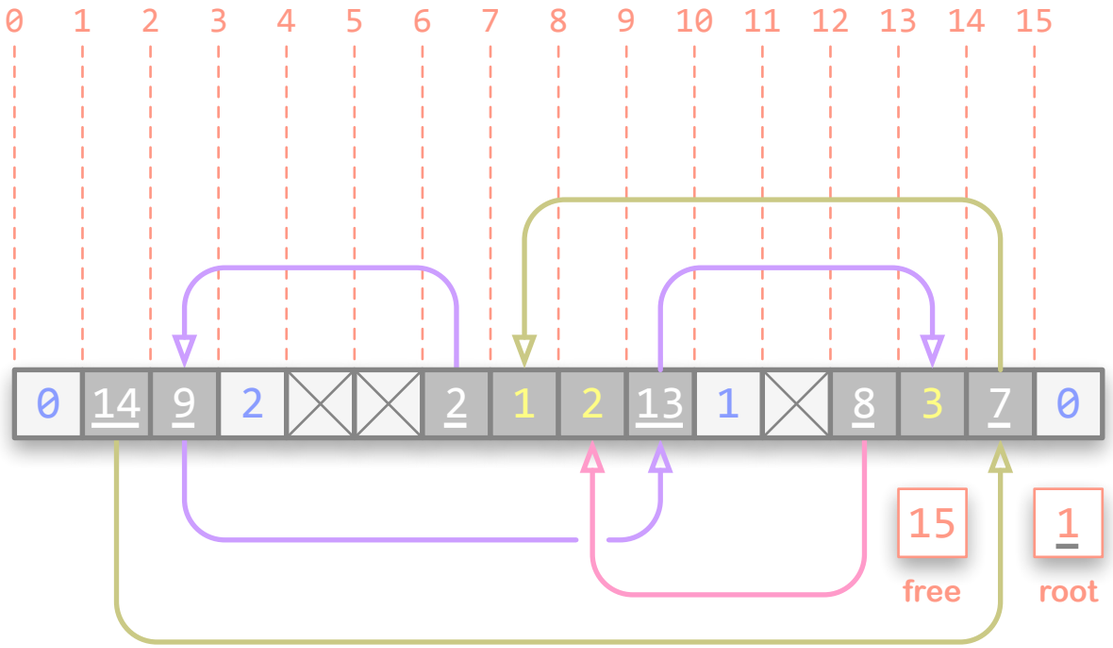


Crunch Memory




```
...
  (define (crunch)
    (define destination 0)
    (do ((source 0))
        ((= source FREE))
      (let ((siz (vector-ref memory source)))
        (if (negative? siz)
          (let ((cell (vector-ref memory source)))
            (set! siz (+ siz MARK))
            (vector-set! memory destination siz)
            (do ((index 0 (+ index 1)))
                ((= index siz))
              (set! source (+ source 1))
              (set! destination (+ destination 1))
              (set! cell (vector-ref memory source))
              (vector-set! memory destination cell))
            (set! source (+ source 1))
            (set! destination (+ destination 1)))
          (set! source (+ source siz 1)))))
      (set! FREE destination)
      (vector-set! memory destination (- SIZE FREE 1))
      (do ((index (+ destination 1) (+ index 1)))
          ((= index SIZE))
        (vector-set! memory index VOID)))
...
  (display memory)
  (newline)
  (sweep)
  (display memory)
  (newline)
  (update)
  (display memory)
  (newline)
  (crunch)
  (display memory))

#(0 1 106 2 999 999 3 101 112 106 1 999 2 106 101 0)
#(0 114 109 2 999 999 102 -99 -98 113 1 999 108 -97 107 0)
#(0 -99 102 2 999 999 -97 100 106 102 1 999 -98 102 100 0)
#(1 102 3 100 106 102 2 102 100 6 999 999 999 999 999 999)
<undefined>
```

Traverse and Invert Memory

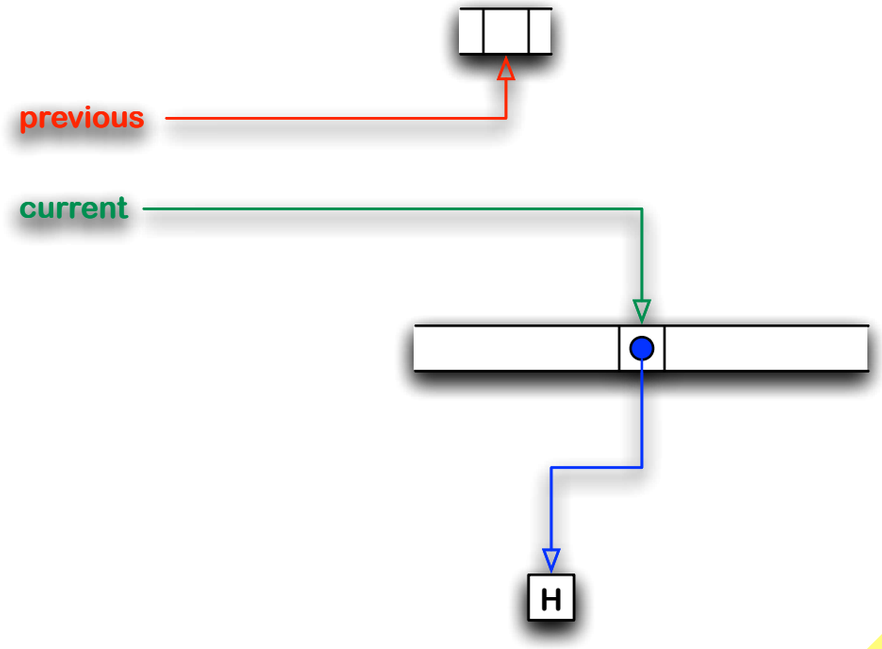


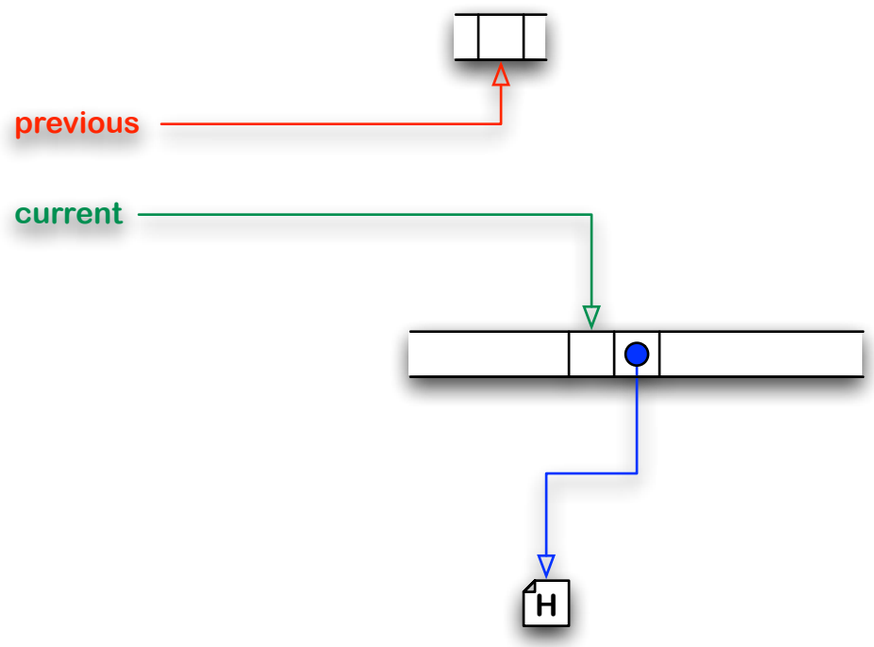

```
...
   (MARK 100)
...
  (define (chunk-size-ref chunk)
    (define index (chunk-to-number chunk))
    (vector-ref memory index))
  (define (chunk-size-set! chunk size)
    (define index (chunk-to-number chunk))
    (vector-set! memory index size))
  (define (set-root chunk)
    (set! ROOT chunk))
  (define (unmarked? size)
    (not (negative? size)))
...

  (define (sweep)
    (define (traverse chunk)
      (define size (chunk-size-ref chunk))
      (cond
        ((number? size)
          (chunk-size-set! chunk (- size MARK))
          (do ((index 0 (+ index 1))) ((= index size))
            (let ((cell (chunk-ref chunk index)))
              (if (chunk? cell)
                  (let* ((offset  (chunk-to-number chunk))
                         (pointer (number-to-chunk
                                    (+ offset index 1))))
                    (traverse cell)
                    (chunk-set! chunk index (chunk-size-ref cell))
                    (chunk-size-set! cell pointer))))))))
      (if (chunk? ROOT)
        (traverse ROOT)))
...
  (display memory)
  (newline)
  (sweep)
  (display memory)

#(0 1 106 2 999 999 3 101 112 106 1 999 2 106 101 0)
#(0 114 109 2 999 999 102 -99 -98 113 1 999 108 -97 107 0)
<undefined>
```

Non-tail-recursive process

Memory Traversal

before: <current> points to a pointer to an unvisited empty chunk

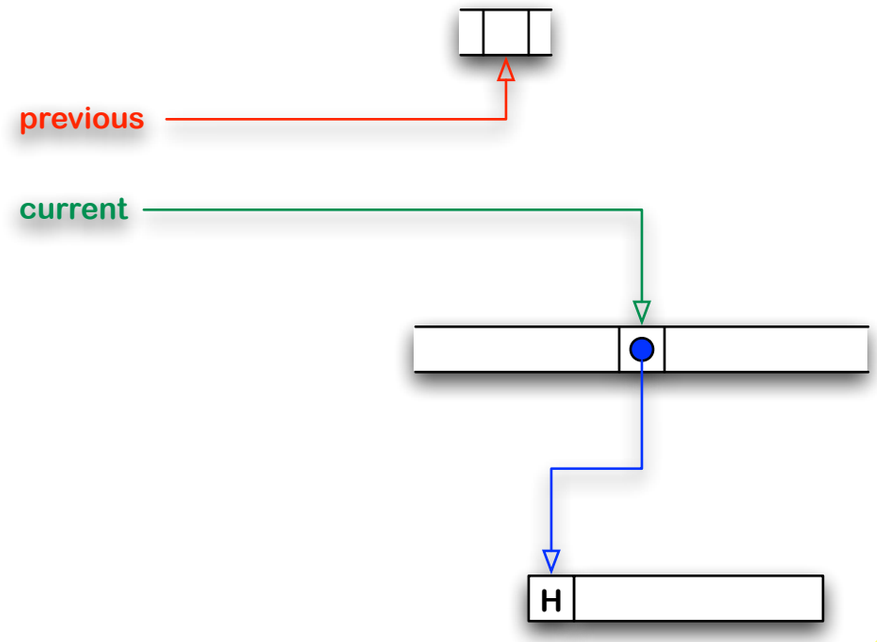


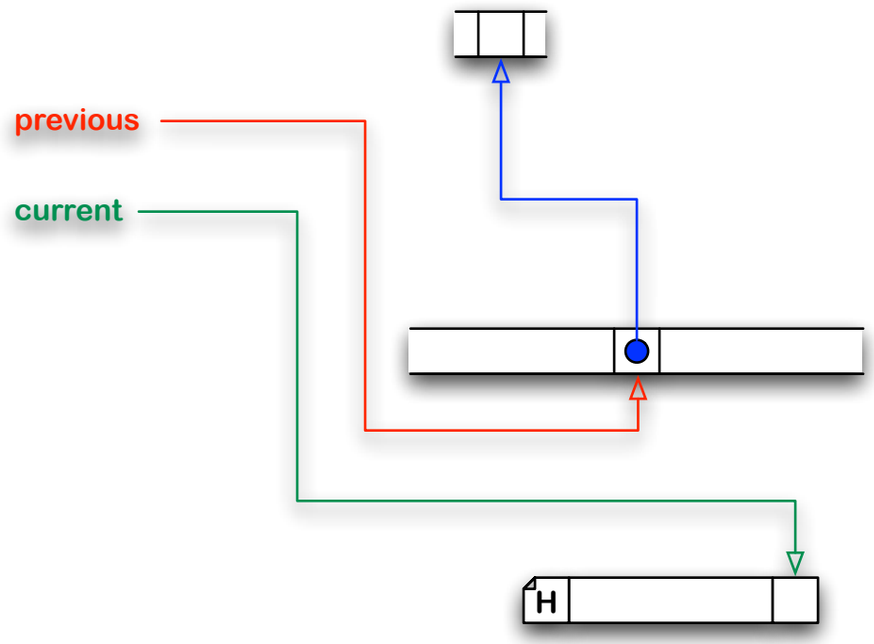


after: chunk is marked as visited and <current> is advanced

Memory Traversal

before: <current> points to a pointer to an unvisited non-empty chunk

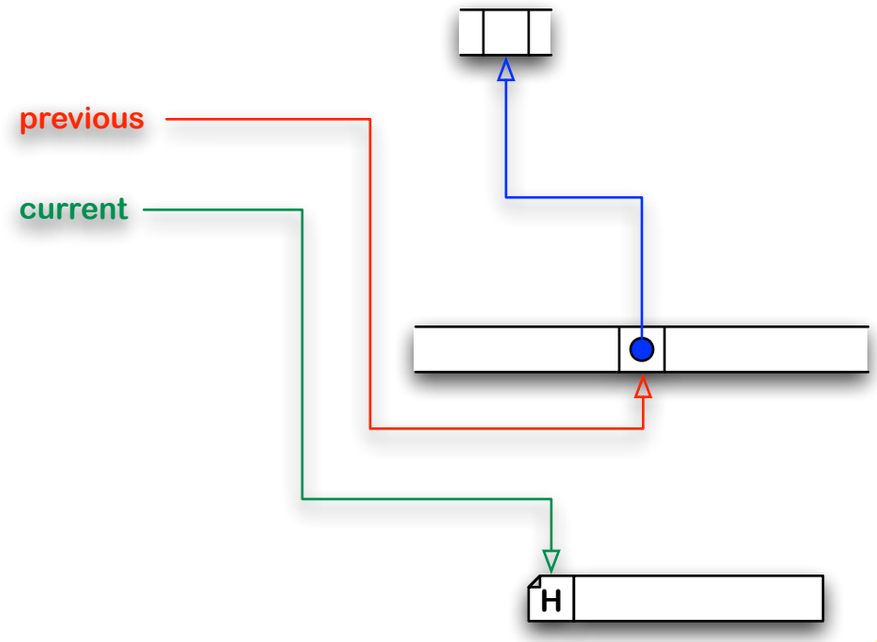


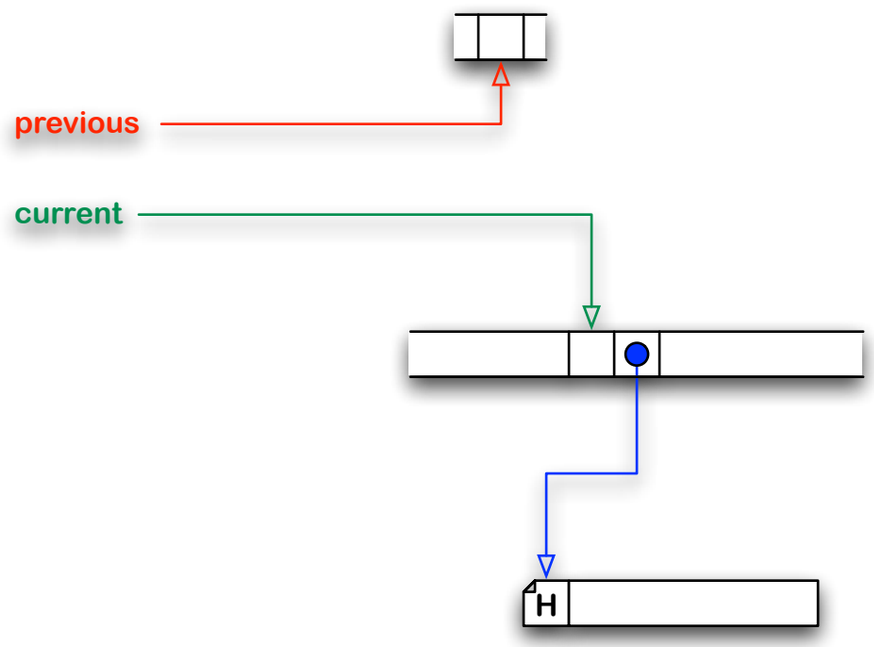


after: chunk is marked as visited and <current> is moved to the chunk's last cell; <previous> is stored instead and updated

Memory Traversal

before: <current> points to a visited chunk

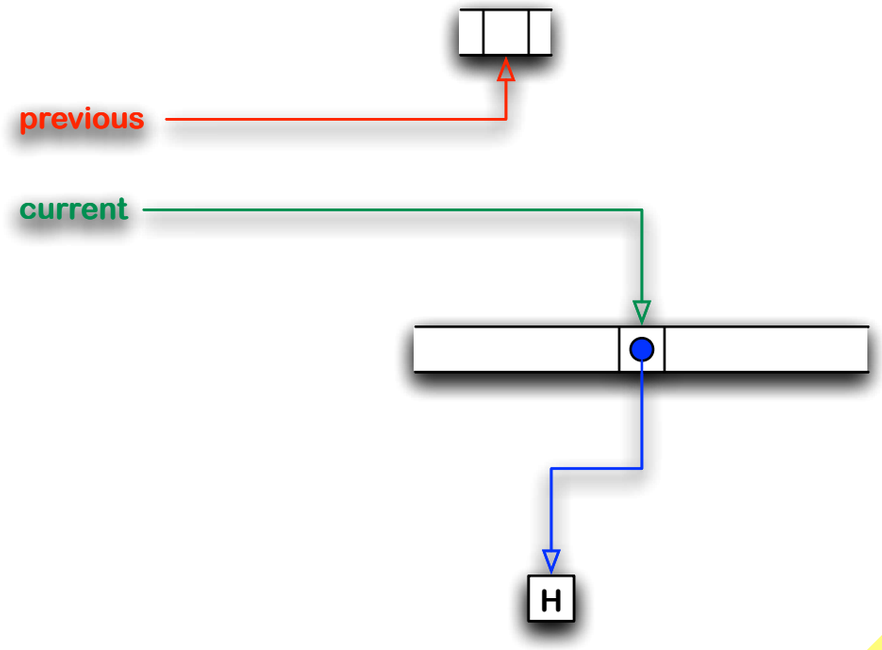


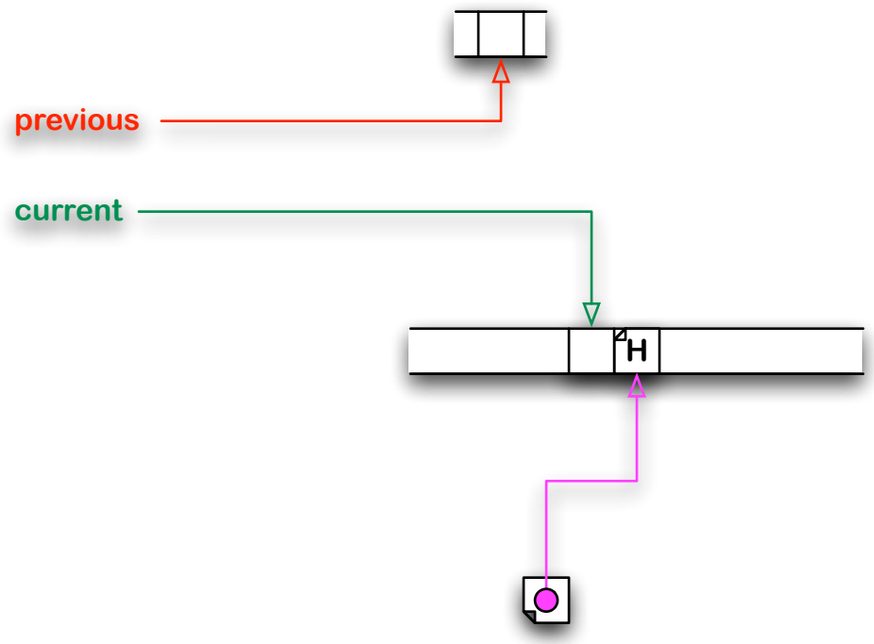


after: <current> is restored from <previous> and advanced; <previous> is restored from its original content

Memory Traversal and Pointer Inversion

before: <current> points to a pointer to an unvisited empty chunk

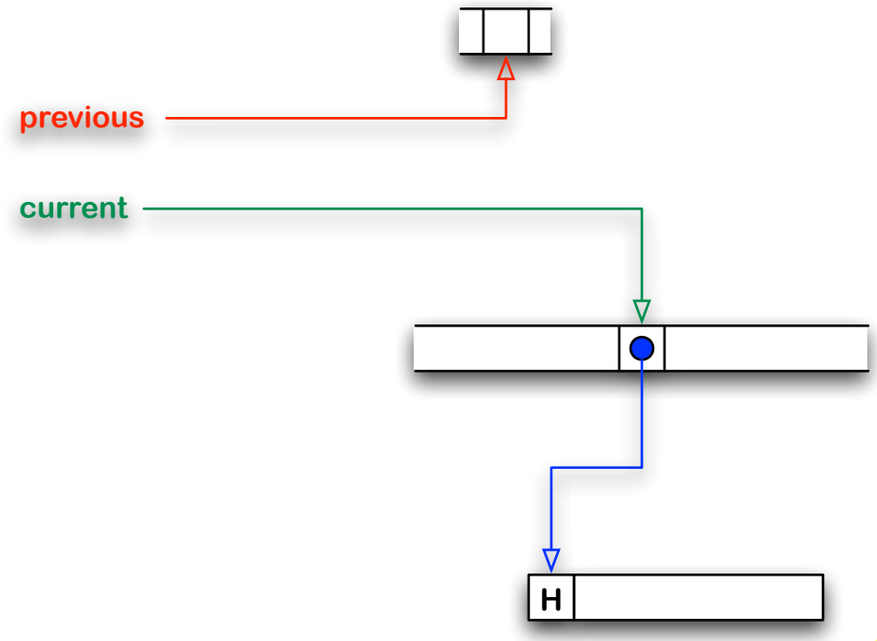


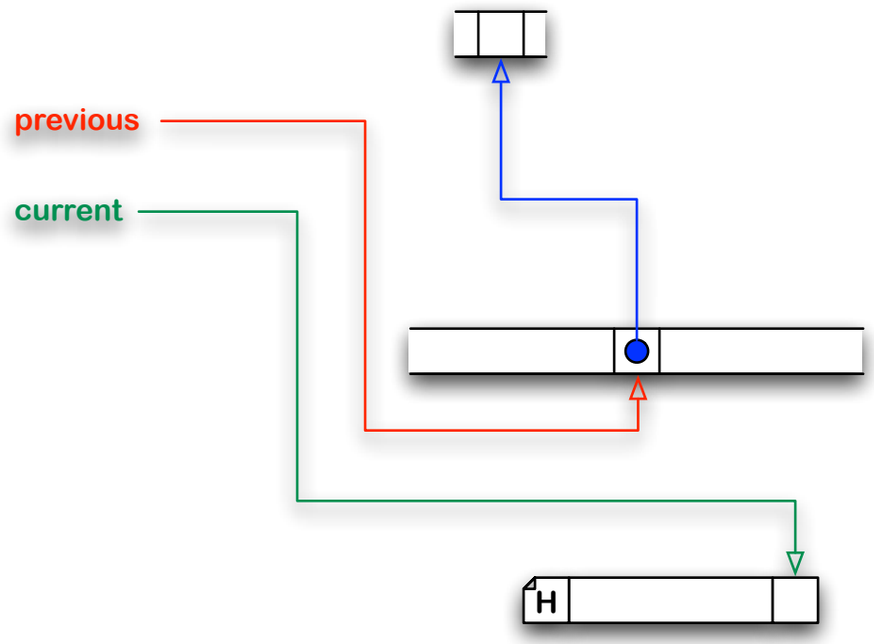


after: chunk is marked as visited, <current> is advanced and pointer is inverted

Memory Traversal and Pointer Inversion

before: <current> points to a pointer to an unvisited non-empty chunk

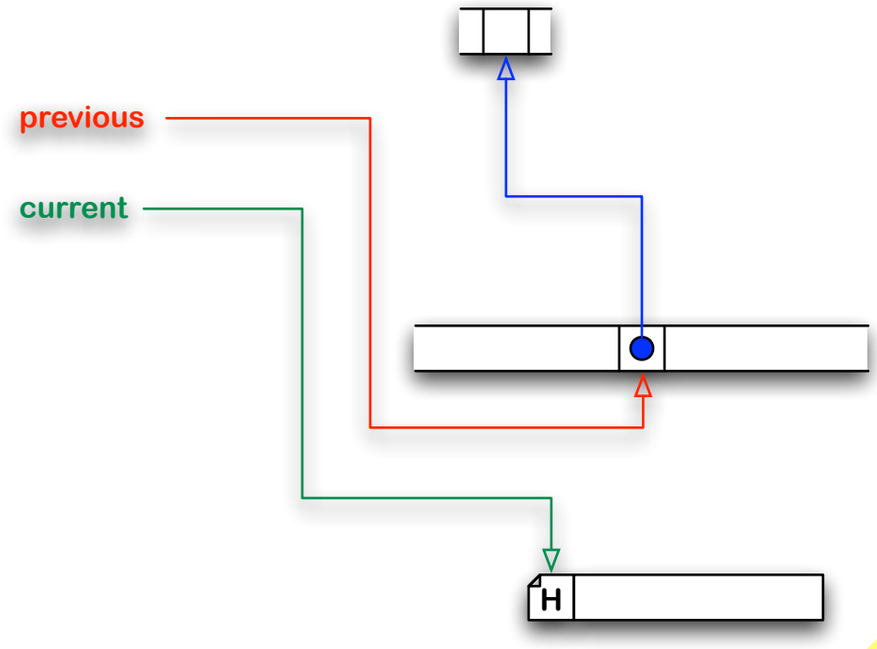


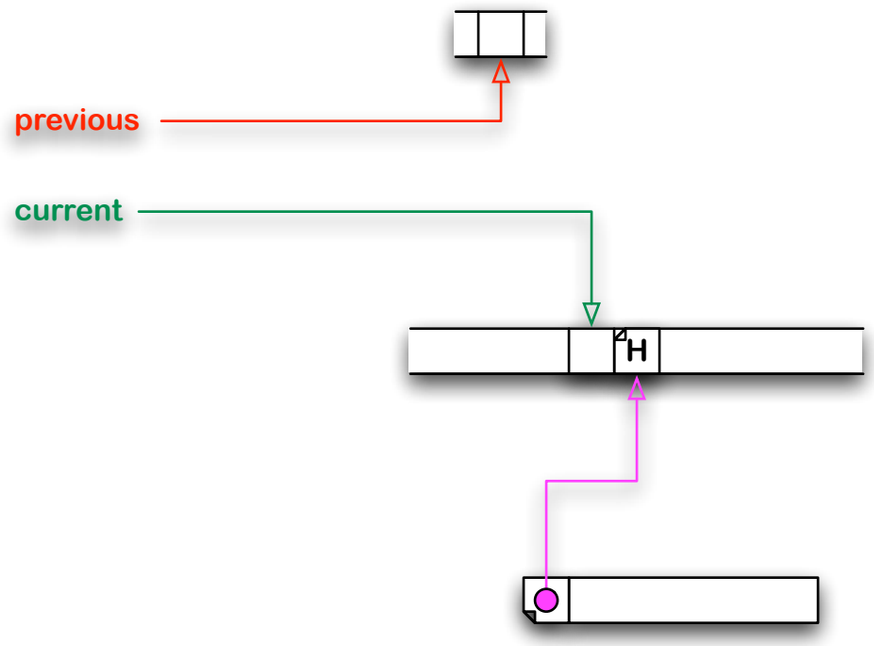


after: chunk is marked as visited and <current> is moved to the chunk's last cell; <previous> is stored instead and updated

Memory Traversal and Pointer Inversion

before: <current> points to a visited chunk

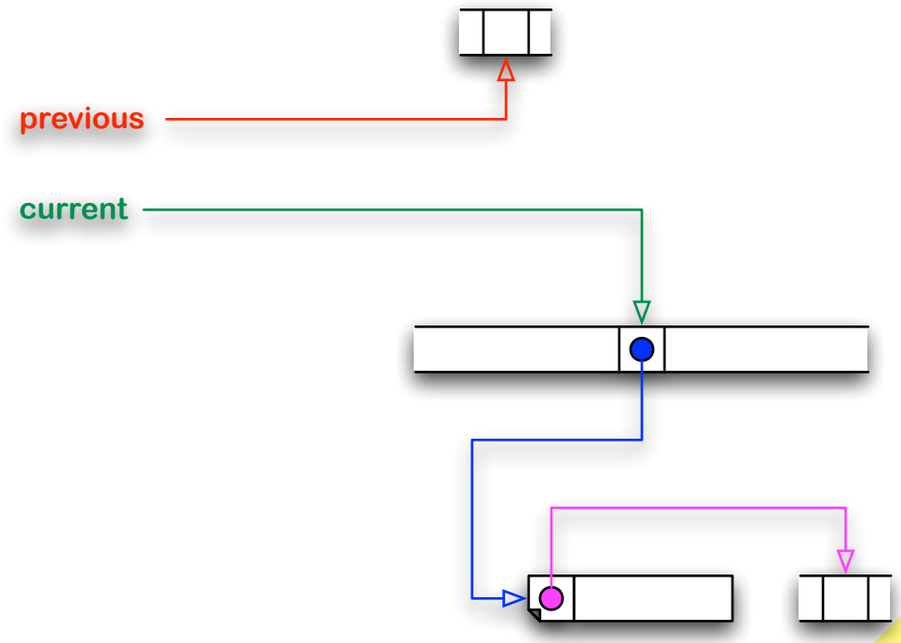


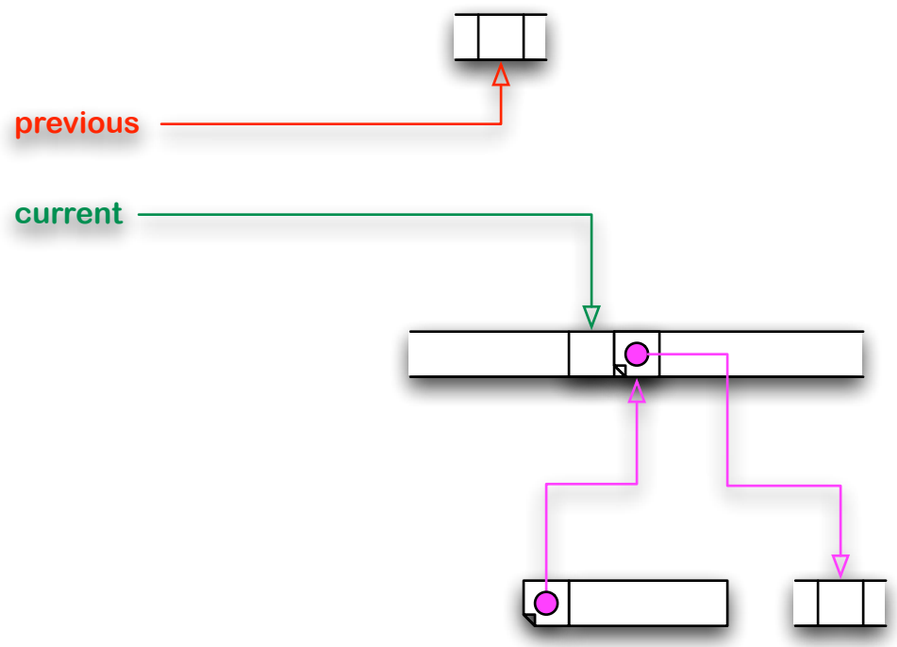


after: <current> is restored from <previous> and advanced; <previous> is restored from its original content; header is inverted

Memory Traversal and Pointer Inversion

before: <current> points to an inverted chunk

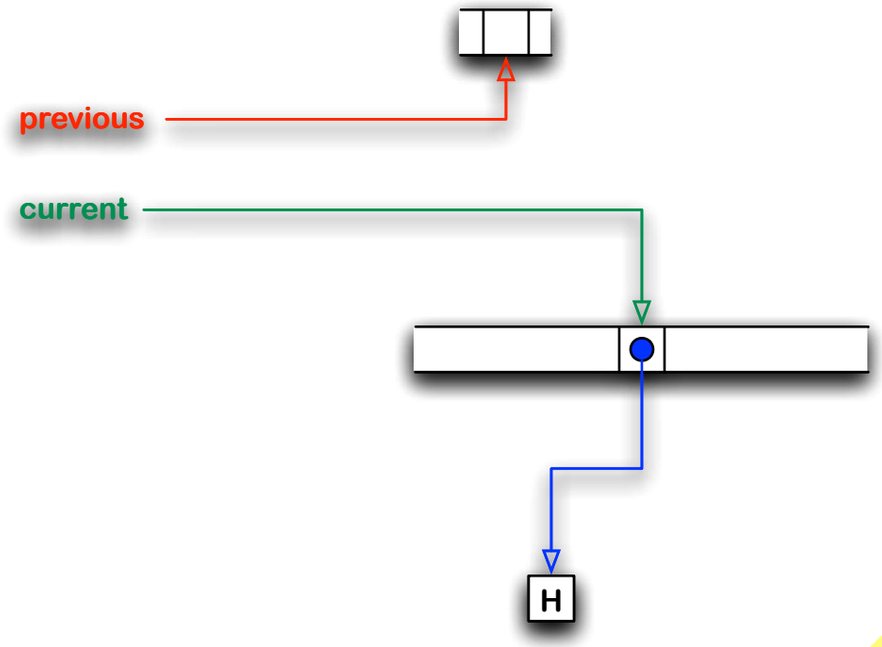


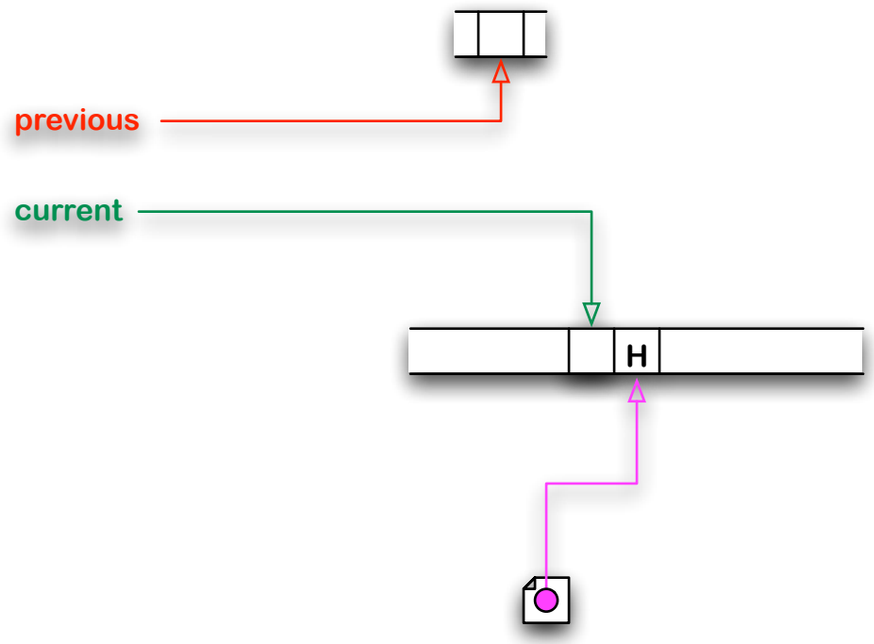


after: <current> is advanced and pointer is stacked on the inversion list

Memory Traversal and Pointer Inversion++

before: <current> points to a pointer to an unvisited empty chunk

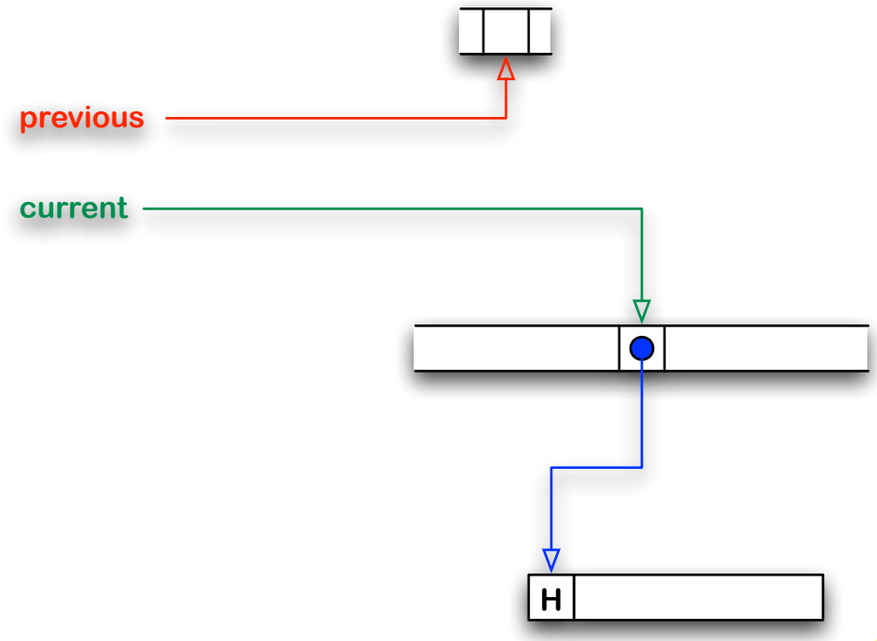


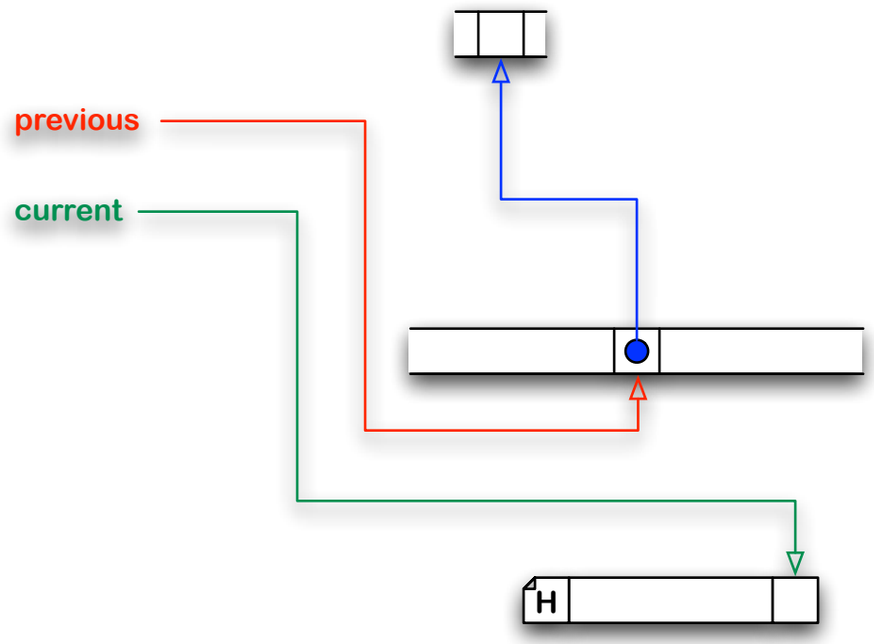


after: chunk is marked as visited, <current> is advanced and pointer is inverted

Memory Traversal and Pointer Inversion++

before: <current> points to a pointer to an unvisited non-empty chunk

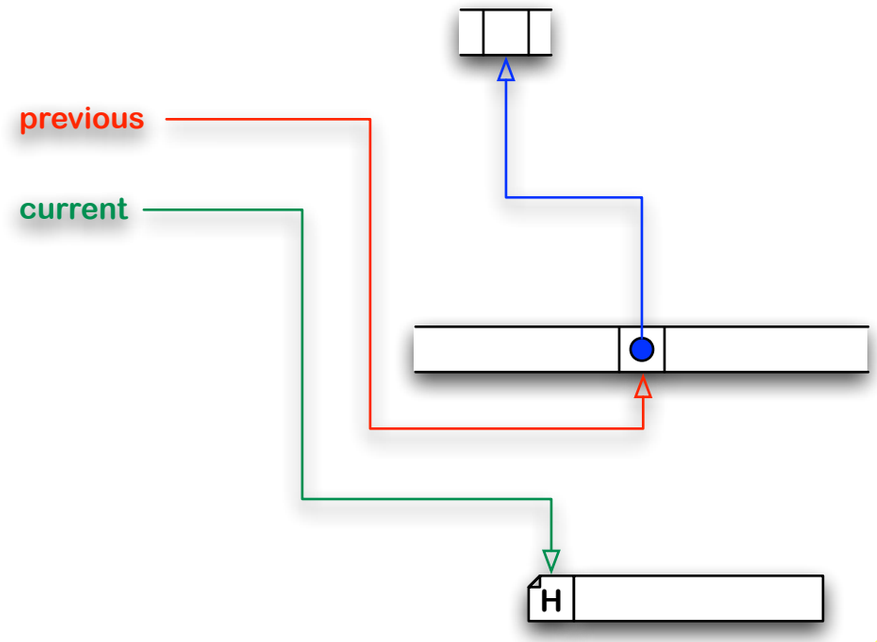


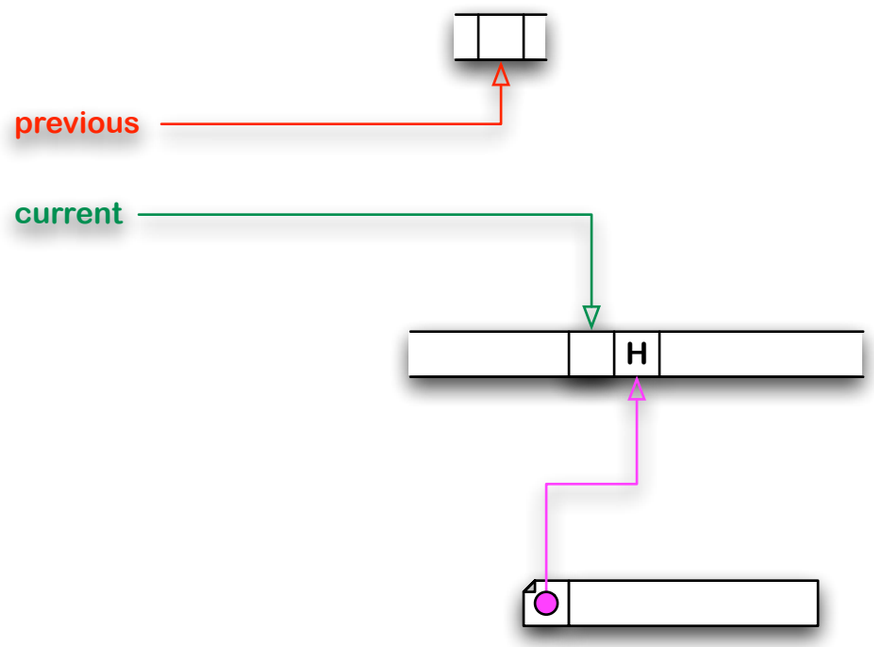


after: chunk is marked as visited and <current> is moved to the chunk's last cell; pointer is inverted

Memory Traversal and Pointer Inversion++

before: <current> points to a pointer to a visited chunk

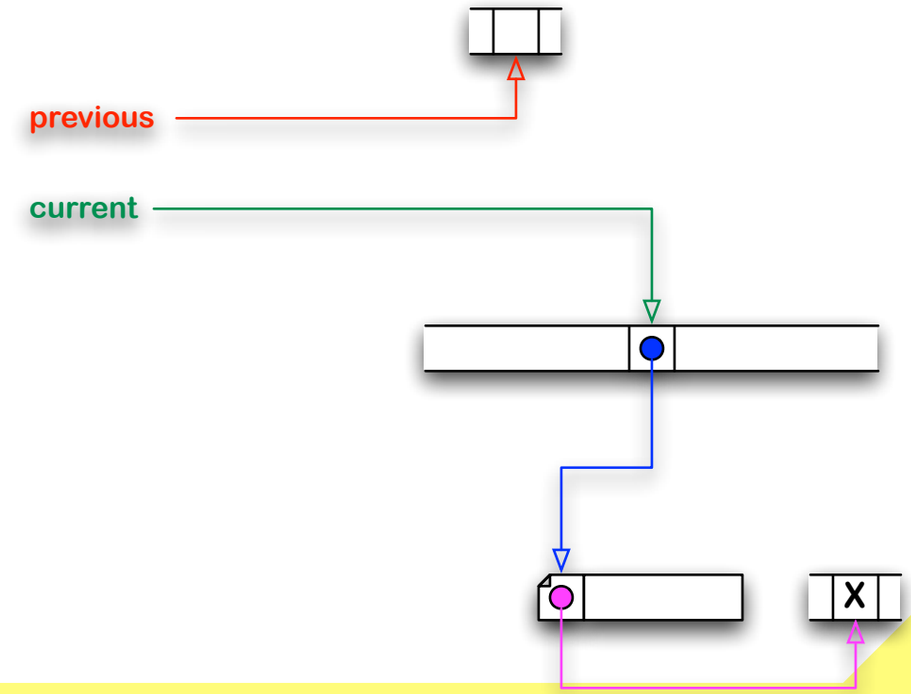


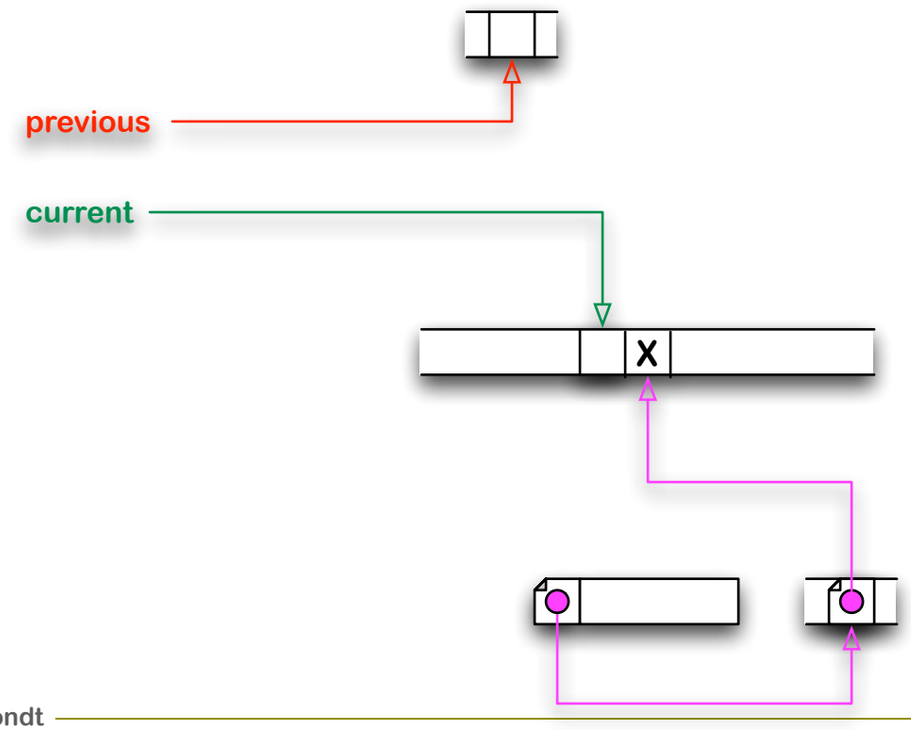


after: <current> is advanced and pointer is inverted while maintaining initial link

Memory Traversal and Pointer Inversion++

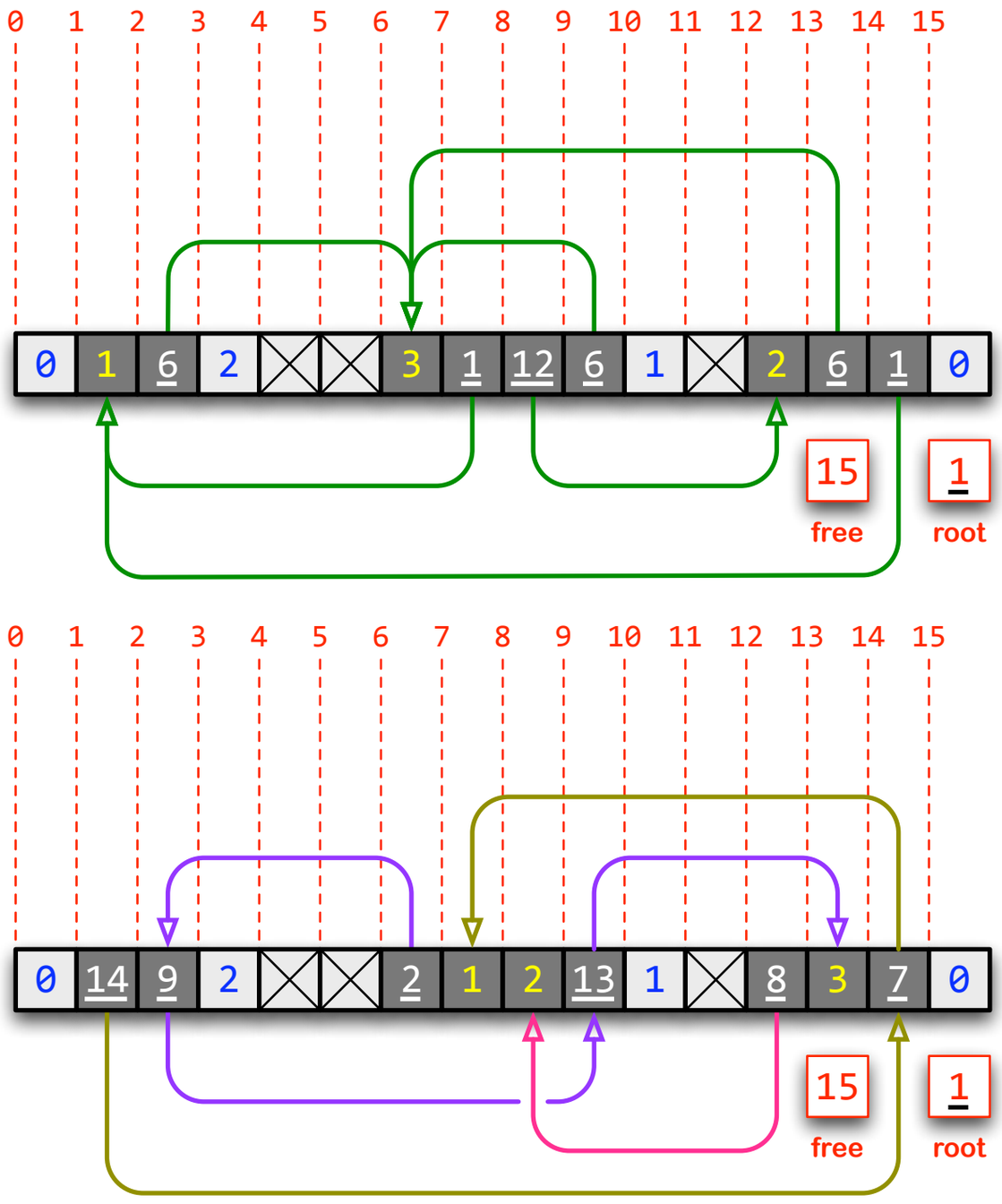
before: <current> points to a visited chunk





after: <current> is restored from the header and advanced

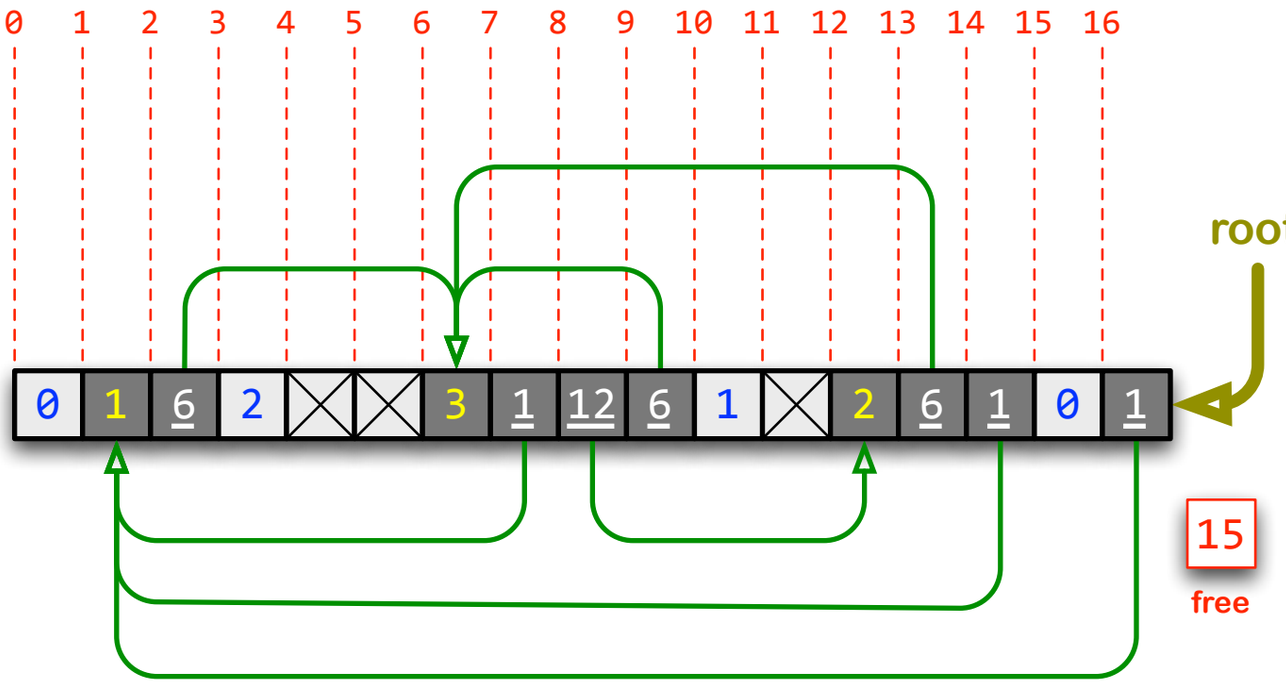
Optimized sweep operation



before ...

```
...
  (define (sweep)
    (define (traverse chunk)
      (define size (chunk-size-ref chunk))
      (cond
        ((number? size)
          (chunk-size-set! chunk (- size MARK))
          (do ((index 0 (+ index 1))) ((= index size))
            (let ((cell (chunk-ref chunk index)))
              (if (chunk? cell)
                 (let* ((offset  (chunk-to-number chunk))
                        (pointer (number-to-chunk
                                   (+ offset index 1))))
                   (traverse cell)
                   (chunk-set! chunk index (chunk-size-ref cell))
                   (chunk-size-set! cell pointer))))))))
    (if (chunk? ROOT)
      (traverse ROOT)))
...

#(0 1 106 2 999 999 3 101 112 106 1 999 2 106 101 0)
#(0 114 109 2 999 999 102 -99 -98 113 1 999 108 -97 107 0)
#(0 -99 102 2 999 999 -97 100 106 102 1 999 -98 102 100 0)
#(1 102 3 100 106 102 2 102 100 6 999 999 999 999 999 999)
<undefined>
```

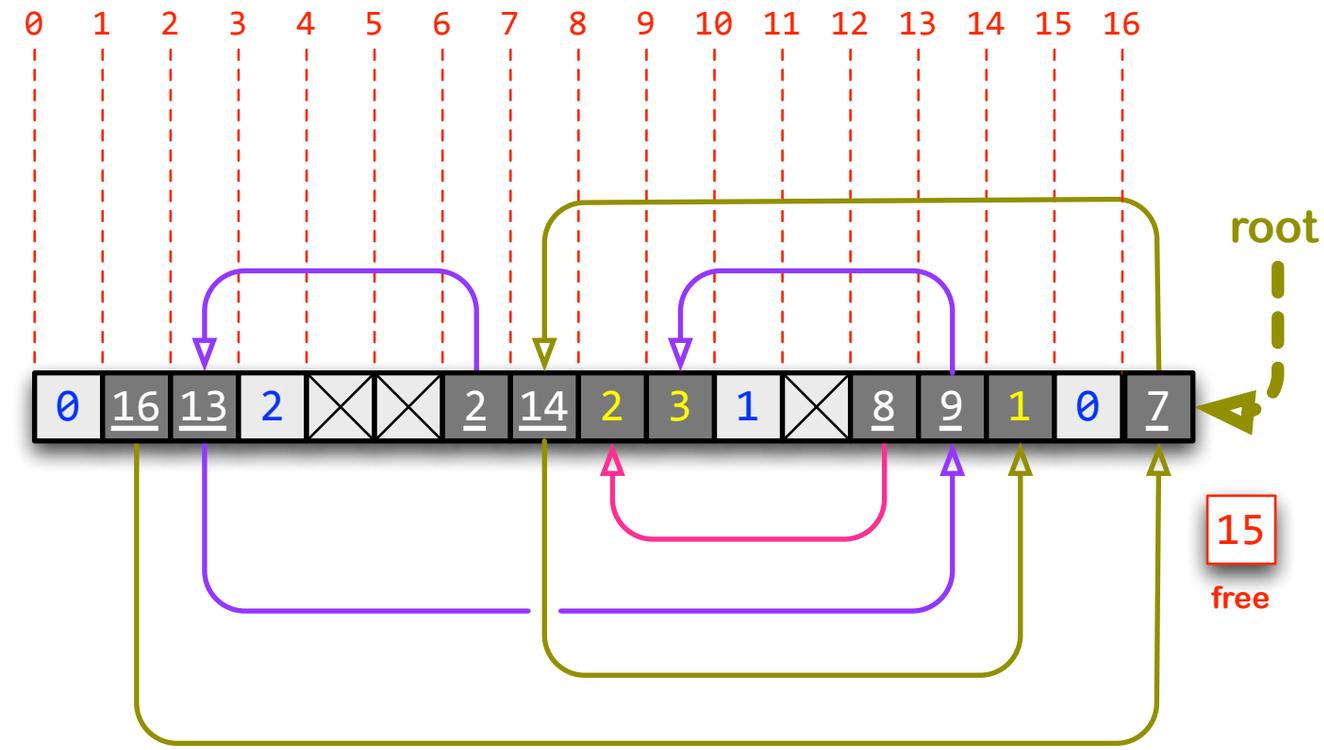
Optimized sweep operation



... after

```
(let*
  ((SIZE 16)
   (BIAS 100)
   (VOID 999)
   (MARK 100)
   (FREE 0)
   (ROOT SIZE))

  (define memory (make-vector (+ SIZE 1) VOID))
  (define (memory-set! offset value)
    (vector-set! memory offset value))
  (define (memory-ref offset)
    (vector-ref memory offset))

...

  (define (make-chunk size)
    (define total (memory-ref FREE))
    (define chunk (number-to-chunk FREE))
    (memory-set! FREE size)
    (set! FREE (+ FREE size 1))
    (memory-set! FREE (- total size 1))
    chunk)
(define (chunk-set! chunk offset value)
  (define index (chunk-to-number chunk))
  (memory-set! (+ index offset 1) value))
(define (chunk-ref chunk offset)
  (define index (chunk-to-number chunk))
  (memory-ref (+ index offset 1)))
(define (set-root chunk)
  (memory-set! ROOT chunk))

(define (mark cell)
  (- (+ cell 1)))
(define (unmark cell)
  (- (- cell) 1))
(define (marked? cell)
  (negative? cell))
```

Optimized sweep operation



... after

```
(define (sweep)
    (define (traverse current)
      (define cell (memory-ref current))
      (if (marked? cell)
        (let* ((chunk (unmark cell))
               (offset (chunk-to-number chunk)))
          (if (not (= offset ROOT))
              (traverse (- offset 1))))
        (if (chunk? cell)
          (let* ((chunk cell)
                 (offset (chunk-to-number chunk))
                 (cell (memory-ref offset)))
            (if (marked? cell)
              (let* ((chunk (unmark cell))
                     (next (chunk-to-number chunk))
                     (any (memory-ref next)))
                (let* ((chunk (number-to-chunk current))
                       (cell  (mark chunk)))
                  (memory-set! current any)
                  (memory-set! next cell)
                  (traverse (- current 1))))
              (let* ((chunk (number-to-chunk current))
                     (size cell)
                     (cell (mark chunk)))
                (memory-set! current size)
                (memory-set! offset cell)
                (if (zero? size)
                  (traverse (- current 1))
                  (traverse (+ offset size))))))
            (traverse (- current 1)))))
    (traverse ROOT))
...

#(0 1 106 2 999 999 3 101 112 106 1 999 2 106 101 0 101)
#(0 -117 -114 2 999 999 -103 -115 2 3 1 999 -109 -110 1 0 -108)
#(0 -2 102 2 999 999 -4 100 106 102 1 999 -3 102 100 0 100)
#(1 102 3 100 106 102 2 102 100 6 999 999 999 999 999 999 100)
<undefined>
```

Design of a Garbage Collector

- Use “critical sections”
- Anticipate collection
- Use immediate values

Design of a Garbage Collector

to coincide with evaluate/ continue functions

- **Use “critical sections”**
- **Anticipate collection**
- **Use immediate values**

Design of a Garbage Collector

to coincide with evaluate/ continue functions

- Use “critical sections”
- **Anticipate collection**
- Use immediate values

by claiming memory in advance

Design of a Garbage Collector

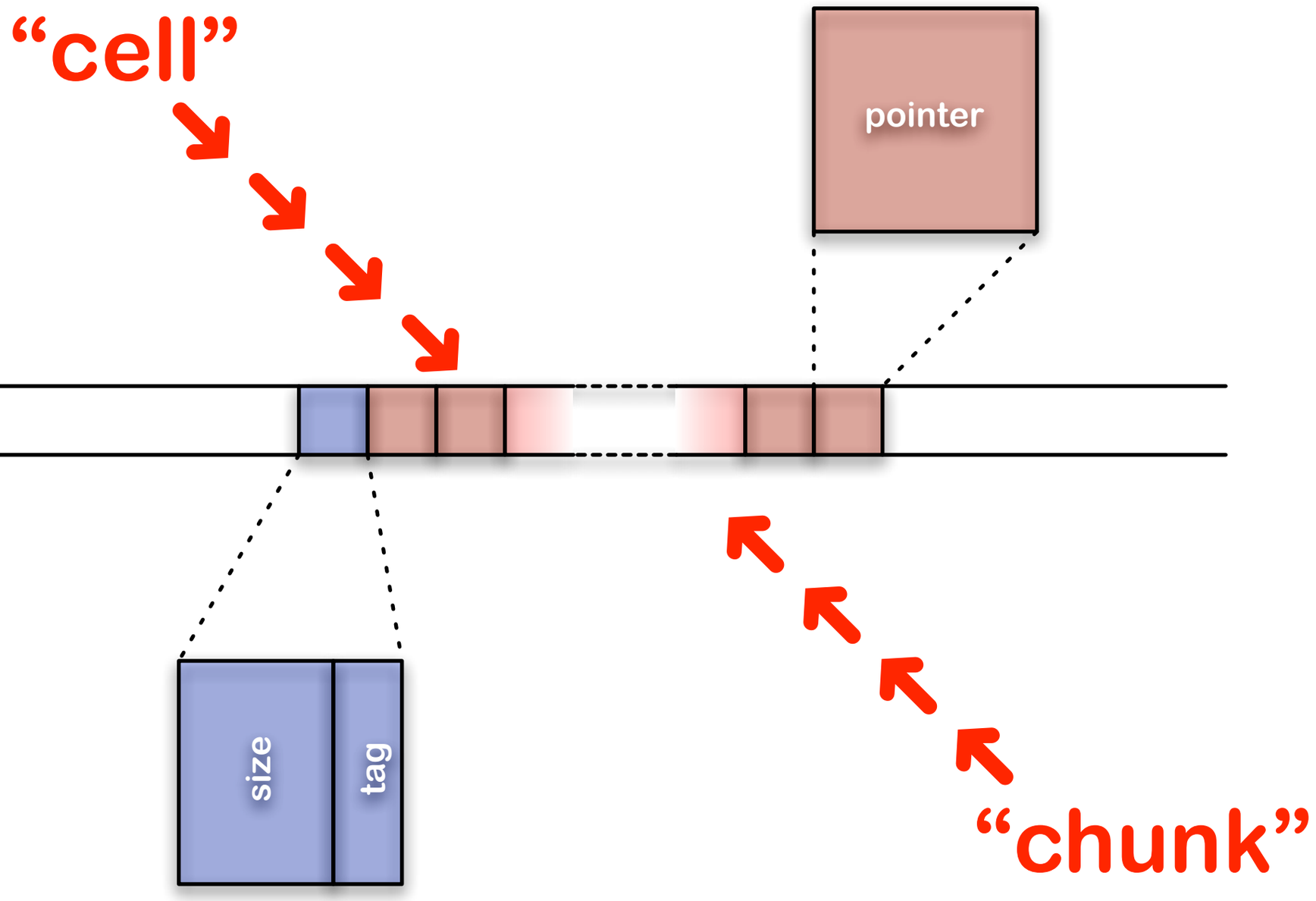
to coincide with evaluate/ continue functions

- Use “critical sections”
- Anticipate collection
- Use immediate values

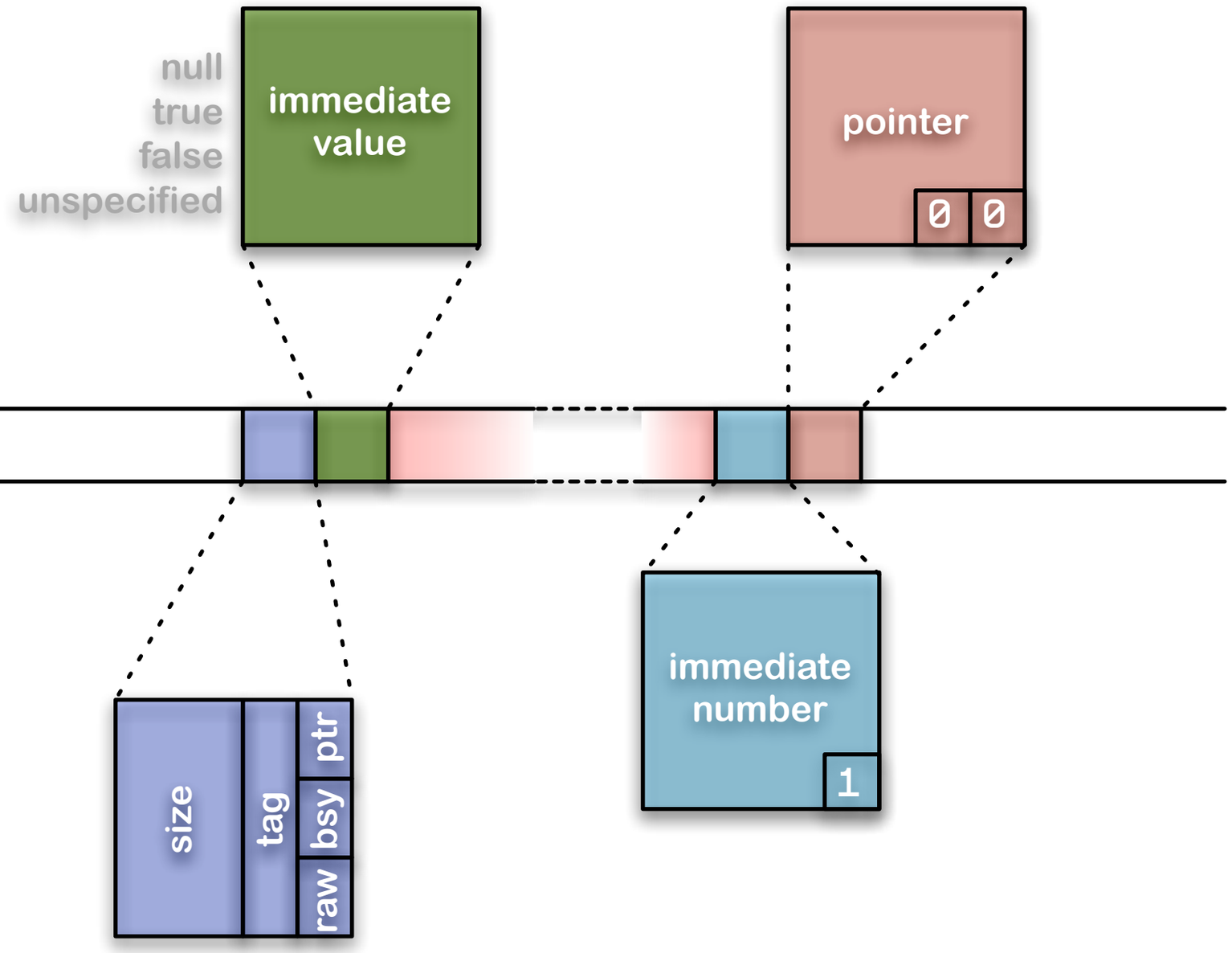
by claiming memory in advance

use “small integers” and inline pointers

The Naive Memory Model



A GC-oriented Memory Model



A GC-oriented Memory Model (cont'd)

```
enum { Immediate_null_value         = Memory_Void_Value + 0*Memory_Cell_Bias,
       Immediate_true_value         = Memory_Void_Value + 1*Memory_Cell_Bias,
       Immediate_false_value        = Memory_Void_Value + 2*Memory_Cell_Bias,
       Immediate_unspecified_value = Memory_Void_Value + 3*Memory_Cell_Bias,
       Immediate_boundary           = Memory_Void_Value + 3*Memory_Cell_Bias };
```

A GC-oriented Memory Model (cont'd)

```
enum { Memory_Cell_Bias          = 0x00000004,
       Memory_Immediate_Maximum = 0x3FFFFFFF,
       Memory_Void_Value         = 0x00000000 };
```

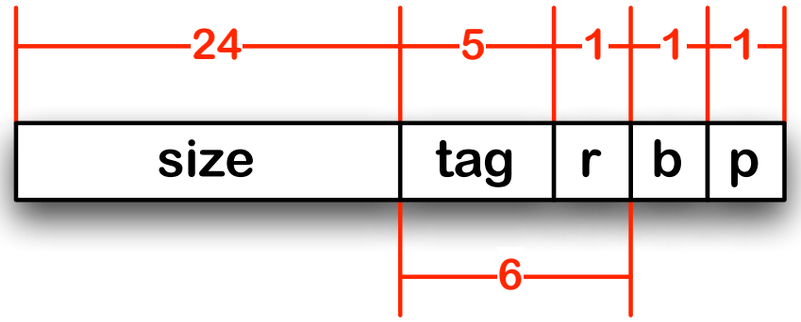
```
enum { Immediate_null_value        = Memory_Void_Value + 0*Memory_Cell_Bias,
       Immediate_true_value        = Memory_Void_Value + 1*Memory_Cell_Bias,
       Immediate_false_value       = Memory_Void_Value + 2*Memory_Cell_Bias,
       Immediate_unspecified_value = Memory_Void_Value + 3*Memory_Cell_Bias,
       Immediate_boundary          = Memory_Void_Value + 3*Memory_Cell_Bias };
```

A GC-oriented Memory Model (cont'd)

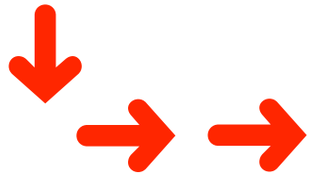
```
enum { Memory_Cell_Bias          = 0x00000004,
       Memory_Immediate_Maximum = 0x3FFFFFFF,
       Memory_Void_Value         = 0x00000000 };

enum { Immediate_null_value        = Memory_Void_Value + 0*Memory_Cell_Bias,
       Immediate_true_value        = Memory_Void_Value + 1*Memory_Cell_Bias,
       Immediate_false_value       = Memory_Void_Value + 2*Memory_Cell_Bias,
       Immediate_unspecified_value = Memory_Void_Value + 3*Memory_Cell_Bias,
       Immediate_boundary          = Memory_Void_Value + 3*Memory_Cell_Bias };

TAG_type Tag_of(EXP_type Expression)
  { UNS_type number;
    if (Memory_Is_Immediate(Expression))
      return NBR_tag;
    number = (UNS_type)Expression;
    if (number <= Immediate_boundary)
      switch (number)
        { case Immediate_null_value:
            return NUL_tag;
          case Immediate_true_value:
            return TRU_tag;
          case Immediate_false_value:
            return FLS_tag;
          case Immediate_unspecified_value:
            return USP_tag; }
    return (TAG_type)Memory_Get_Tag((PTR_type)Expression); }
```

Optimized Abstract Grammar

```
typedef enum { APL_tag = 0x00<<1 | 0x0,
               BEG_tag = 0x01<<1 | 0x0,
               CNT_tag = 0x02<<1 | 0x0,
               DFF_tag = 0x03<<1 | 0x0,
               DFV_tag = 0x04<<1 | 0x0,
               DFZ_tag = 0x05<<1 | 0x0,
               ENV_tag = 0x06<<1 | 0x0,
               FRM_tag = 0x07<<1 | 0x0,
               GLB_tag = 0x08<<1 | 0x0,
               IFF_tag = 0x09<<1 | 0x0,
               IFZ_tag = 0x0A<<1 | 0x0,
               LCL_tag = 0x0B<<1 | 0x0,
               LMB_tag = 0x0C<<1 | 0x0,
               LMZ_tag = 0x0D<<1 | 0x0,
               PAI_tag = 0x0E<<1 | 0x0,
               PRC_tag = 0x0F<<1 | 0x0,
               PRZ_tag = 0x10<<1 | 0x0,
               QUO_tag = 0x11<<1 | 0x0,
               STG_tag = 0x12<<1 | 0x0,
               STL_tag = 0x13<<1 | 0x0,
               VEC_tag = 0x14<<1 | 0x0,
               WHI_tag = 0x15<<1 | 0x0,
```



r = raw
b = busy
p = pointer

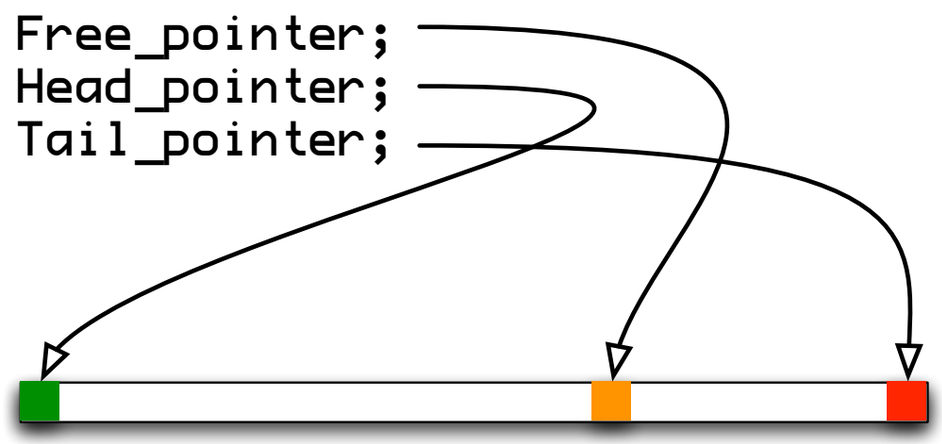


```
               CHA_tag = 0x00<<1 | 0x1,
               FLS_tag = 0x01<<1 | 0x1,
               NAT_tag = 0x02<<1 | 0x1,
               NUL_tag = 0x03<<1 | 0x1,
               NBR_tag = 0x04<<1 | 0x1,
               REA_tag = 0x05<<1 | 0x1,
               STR_tag = 0x06<<1 | 0x1,
               SYM_tag = 0x07<<1 | 0x1,
               TRU_tag = 0x08<<1 | 0x1,
               USP_tag = 0x09<<1 | 0x1 } TAG_type;
```

Garbage Collection

```
PTR_type Memory_Collect(PTR_type Root)
  { Head_pointer->ptr = Root;
    sweep_and_mark();
    traverse_and_update();
    traverse_and_compact();
    return Head_pointer->ptr; }
```

```
static PTR_type Free_pointer;
static PTR_type Head_pointer;
static PTR_type Tail_pointer;
```



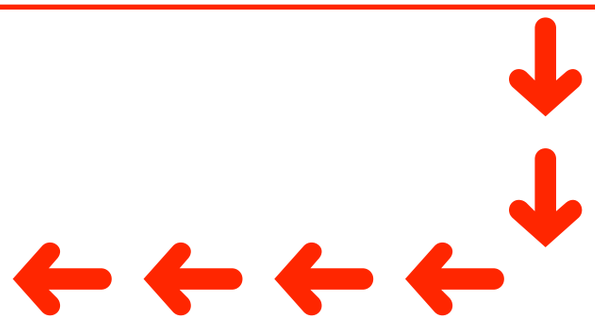
Garbage Collection (cont'd)

```
static NIL_type sweep_and_mark(NIL_type)
  { CEL_type bits;
    PTR_type current,
             pointer;
    UNS_type size;
    for (current = Head_pointer;;)
      { bits = current->cel;
        switch (get_last_2_bits(bits))
          { case bp_bits:
              pointer = current->ptr;
              if ((pointer < Head_pointer) ||
                  (pointer > Tail_pointer))
                { current--;
                  continue; }
              bits = pointer->cel;
              switch (get_last_3_bits(bits))
                { case rbp_bits:
                    pointer->cel = make_busy(current);
                    current->cel = bits;
                    size = get_size(bits);
                    if (size)
                      current = pointer + size;
                    else
                      current--;
                    continue;
```

Garbage Collection (cont'd)

```
static NIL_type sweep_and_mark(NIL_type)
  { CEL_type bits;
    PTR_type current,
             pointer;
    UNS_type size;
    for (current = Head_pointer;;)
      { bits = current->cel;
        switch (get_last_2_bits(bits))
          { case bp_bits:
              pointer = current->ptr;
              if ((pointer < Head_pointer) ||
                  (pointer > Tail_pointer))
                { current--;
                  continue; }
its = pointer->cel;
witch (get_last_3_bits(bits))
 { case rbp_bits:
     pointer->cel = make_busy(current);
     current->cel = bits;
     size = get_size(bits);
     if (size)
       current = pointer + size;
     else
       current--;
     continue;
```

```
      case rBp_bits:
      case RBp_bits:
        pointer = make_free(bits);
        bits = pointer->cel;
      case Rbp_bits:
        pointer->cel = make_busy(current);
        current->cel = bits;
        current--; }
  continue;
case Bp_bits:
  current = make_free(bits);
  if (current == Head_pointer)
    return;
case bP_bits:
case BP_bits:
  current--; }}}
```



Garbage Collection (cont'd)

```
static const CEL_type Busy_mask      = 0x00000002;
static const CEL_type Free_mask      = 0xFFFFFFFD;
static const CEL_type Immediate_mask = 0x00000001;
static const UNS_type Size_max       = 0x00FFFFFF;
static const CEL_type Tag_mask       = 0x000000FC;
static const CEL_type Three_bit_mask = 0x00000007;
static const CEL_type Two_bit_mask   = 0x00000003;

enum { Busy_shift      = 1,
       Immediate_shift = 1,
       Size_shift      = 8,
       Tag_shift       = 2 };

enum { bp_bits = 0x0,
       bP_bits = 0x1,
       Bp_bits = 0x2,
       BP_bits = 0x3 };

enum { rbp_bits = 0x0,
       rBp_bits = 0x2,
       rBP_bits = 0x3,
       Rbp_bits = 0x4,
       RBp_bits = 0x6,
       RBP_bits = 0x7 };
```

Garbage Collection (cont'd)

```
static CEL_type make_header(BYT_type Tag,
                            UNS_type Size)
  { return ((CEL_type)Size << Size_shift) | ((CEL_type)Tag << Tag_shift); }

static BLN_type is_busy(CEL_type Cell)
  { return (Cell & Busy_mask) >> Busy_shift; }

static CEL_type free_size(UNS_type Size)
  { return (CEL_type)Size << Size_shift; }

static BYT_type get_last_2_bits(CEL_type Cell)
  { return Cell & Two_bit_mask; }

static BYT_type get_last_3_bits(CEL_type Cell)
  { return Cell & Three_bit_mask; }

static UNS_type get_size(CEL_type Cell)
  { return (UNS_type)(Cell >> Size_shift); }

static CEL_type make_busy(PTR_type Pointer)
  { return (CEL_type)Pointer | Busy_mask; }

static PTR_type make_free(CEL_type Cell)
  { return (PTR_type)(Cell & Free_mask); }
```

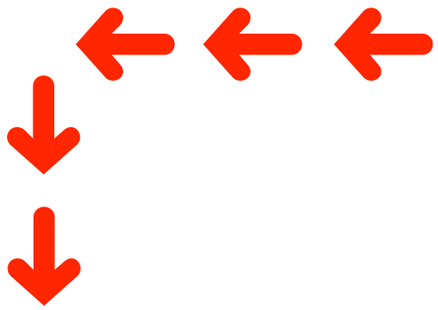
Garbage Collection (cont'd)

```
static NIL_type traverse_and_compact(NIL_type)
  { CEL_type bits;
    PTR_type destination,
             source;
    UNS_type size;
    source = destination = Head_pointer + 1;
    while (source < Free_pointer)
      { bits = (source++)->cel;
        size = get_size(bits);
        if (is_busy(bits))
          { (destination++)->ptr = make_free(bits);
            while (size--)
              (destination++)->cel = (source++)->cel; }
        else
          source += size; }
    Free_pointer = destination; }
```

Garbage Collection (cont'd)

```
static NIL_type traverse_and_update(NIL_type)
  { CEL_type bits;
    PTR_type destination,
             free,
             source,
             this;
    UNS_type accumulated_size,
             size;
    source = destination = Head_pointer + 1;
    while (source < Free_pointer)
      { bits = source->cel;
        if (is_busy(bits))
          { do
              { this = make_free(bits);
                bits = this->cel;
                this->ptr = destination; }
            while (is_busy(bits));
            source->cel = make_busy((PTR_type)bits);
            size = get_size(bits) + 1;
            source += size;
            destination += size; }
        else
          { free = source;
            accumulated_size = 0;
```

Garbage Collection (cont'd)



```
static NIL_type traverse_and_update(NIL_type)
  { CEL_type bits;
    PTR_type destination,
             free,
             source,
             this;
    UNS_type accumulated_size,
             size;
    source = destination = Head_pointer + 1;
    while (source < Free_pointer)
      { bits = source->cel;
        if (is_busy(bits))
          { do
              { this = make_free(bits);
                bits = this->cel;
                this->ptr = destination; }
            while (is_busy(bits));
            source->cel = make_busy((PTR_type)bits);
            size = get_size(bits) + 1;
                 size;
            on += size; }

            urce;
            ed_size = 0;
```

```
do
  { size = get_size(bits) + 1;
    if ((accumulated_size + size) >= Size_max)
      break;
    accumulated_size += size;
    source += size;
    if (source >= Free_pointer)
      break;
    bits = source->cel; }
while (!is_busy(bits));
free->cel = free_size(accumulated_size - 1); }}}
```