CS61C Midterm 1 Review

Summer 2004

Pooya Pakzad
Ben Huang
Navtej Sadhal

Overview

- Number Representation
- C
- Memory Management
- MIPS Assembly

Changing Bases

- A number in any base can be expanded to calculate its decimal representation:

123_7 =

FD_{16} =

Changing Bases

- A number in any base can be expanded to calculate its decimal representation:

$123_7 = (1x7^2) + (2x7^1) + (3x7^0)$

$= 66_{10}$

$FD_{16} =$

Changing Bases

- A number in any base can be expanded to calculate its decimal representation:

$123_7 = (1x7^2) + (2x7^1) + (3x7^0)$

$= 66_{10}$

$FD_{16} = (15x16^1) + (13x16^0)$

$= 253_{10}$

Field width

- An N digit number in base b:
  - b^N possible values
- How many values in an 8-bit number?

Field width

- An N digit number in base b:

  b^N possible values

- How many values in an 8-bit number?

  $2^8 = 256_{10}$

Shortcuts

- It is easy to change between bases that are a power of 2:

  $F00D_{16}$

Shortcuts

- It is easy to change between bases that are a power of 2:

$F00D_{16}$

F	0	0	D
1111	0000	0000	1101

Shortcuts

- It is easy to change between bases that are a power of 2:

$F00D_{16}$

F	0	0	D
1111	0000	0000	1101

1111000000001101_2

Numbers in a Computer

- Unsigned integers - nothing tricky
- Signed Integers
  - Sign-magnitude
  - 1’s complement
  - 2’s complement
- Overflow

Signed Number Systems

Fill in the following decimal values for these systems (8 bit):

Decimal	Sign Mag.	1’s Comp.	2’s Comp.
255			
2			
1			
0			
-1			
-2			
-255			

Signed Number Systems

Fill in the following decimal values for these systems (8 bit):

Decimal	Sign Mag.	1's Comp.	2's Comp.
127_{10}	01111111	01111111	01111111
2_{10}			
1_{10}			
0_{10}			
-1_{10}			
-2_{10}			
-127_{10}			

Signed Number Systems

Fill in the following decimal values for these systems (8 bit):

Decimal	Sign Mag.	1's Comp.	2's Comp.
127_{10}	01111111	01111111	01111111
2_{10}	00000010	00000010	00000010
1_{10}	00000001	00000001	00000001
0_{10}	[0|1]0000000	[0|1]0000000	00000000
-1_{10}			
-2_{10}			
-127_{10}			

Signed Number Systems

Fill in the following decimal values for these systems (8 bit):

Decimal	Sign Mag.	1's Comp.	2's Comp.
127_{10}	01111111	01111111	01111111
2_{10}	00000010	00000010	00000010
1_{10}	00000001	00000001	00000001
0_{10}	[0|1]0000000	[0|1]0000000	00000000
-1_{10}	10000001	11111110	11111111
-2_{10}	10000010	11111101	11111110
-127_{10}	11111111	10000000	10000001

Two’s complement

- Benefits over SM and 1’s:
  - Only one 0
  - Numbers go in simple unsigned order with discontinuity only at 0
- Extremes:
  - Highest value: $2^{N-1}-1$
  - Lowest value: -2^{N-1}

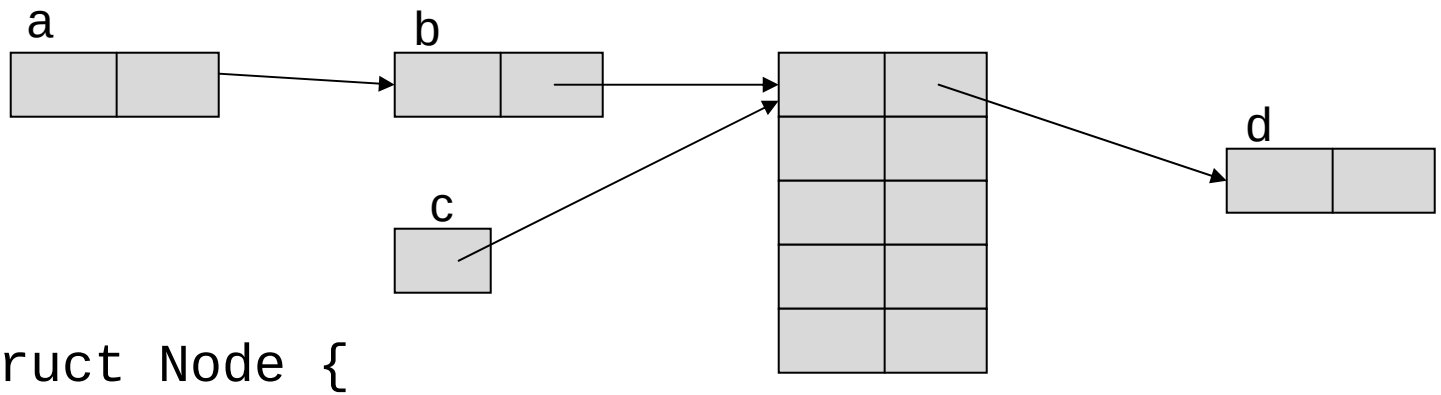
Pointers in C

- Pointers
  - A pointer contains an address of a piece of data.
  - & gets the address of a variable
  - * dereferences a pointer

```
int a;
int *b = &a;
int c = *b;
```

Pointers

How would you create this situation in C without using malloc()?




```
struct Node {
	int * i;
	struct Node * next;
};
```

Pointers

```
struct Node {
   int * i;
   struct Node * next;
};
int main() {
   struct Node a, b, c[5], d;
   a.next = &b;
   b.next = c;
   c[0].next = &d;
   return 0;
}
```

Pointers

How would you remove an element from a linked list given its value?

```
struct node {
   int * i;
   struct node * next;
};
typedef struct node Node;

void remove( … ){
…
}
```

Pointers

```
int removeNode(Node **lstPtr, int toBeRemoved){
```

Pointers

```
int removeNode(Node **lstPtr, int toBeRemoved){
      if ((*lstPtr)==NULL) {
            return 0;
      } else if ((*lstPtr)->i == toBeRemoved) {
            *lstPtr = (*lstPtr)->next;
            return 1;
      } else {
            return removeNode(
                  &((*lstPtr)->next),toBeRemoved);
      }
}
```

Malloc

- Allocates memory on the heap
- Data not disappear after function is removed from stack
- How to dynamically allocate an array of integers with a length of 10?

Malloc

- Allocates memory on the heap
- Data not disappear after function is removed from stack
- How to dynamically allocate an array of integers with a length of 10?

```
int *i=(int *)malloc(sizeof(int)*10);
```

Malloc

- Allocates memory on the heap
- Data not disappear after function is removed from stack
- How to dynamically allocate an array of integers with a length of 10?
```
int *i=(int *)malloc(sizeof(int)*10);
```
- String of length 80?

Malloc

- Allocates memory on the heap
- Data not disappear after function is removed from stack
- How to dynamically allocate an array of integers with a length of 10?

```
int *i=(int *)malloc(sizeof(int)*10);
```

- String of length 80?

```
char *str=(char*)malloc(sizeof(char)*81);
```

MIPS Assembly

- Procedure call convention:
  - Arguments in $a0,$a1…
  - Return values in $v0, $v1…
  - Callee saved registers:
    - $s0, $s1… $sp
  - Caller saved registers:
    - $a0, $a1…, $t0, $t1, … $v0, $v1…, $ra

Translating from C to MIPS

Merge sort (Fall 2003 MT1)

```
struct node {
   int value;
   struct node *next;
};

struct node *mergesort (struct node *list) {
   if (list == 0 || list->next == 0)
        return list;
   return
        merge(mergesort(evens(list)),mergesort(odds(list)));
}
```

Translating C to MIPS

```
mergesort:
   addi $sp, -12          # prologue:
   sw $ra, 0($sp)
   sw $a0, 4($sp)
```

Translating C to MIPS

```
add $v0, $a0, $zero      # body:
   beqz $a0, epi         # end test: list == 0
   lw $t0, 4($a0)        # list->next
   beqz $t0, epi         # end test: list->next == 0
   jal evens             # evens(list)
```

Translating C to MIPS

```
add $v0, $a0, $zero      # body:
   beqz $a0, epi         # end test: list == 0
   lw $t0, 4($a0)        # list->next
   beqz $t0, epi         # end test: list->next == 0
   jal evens             # evens(list)
   add $a0, $v0, $zero   # ret val becomes arg
   jal mergesort         # mergesort(evens(list))
   sw $v0, 8($sp)        # save the result
```

Translating C to MIPS

```
add $v0, $a0, $zero      # body:
   beqz $a0, epi         # end test: list == 0
   lw $t0, 4($a0)        # list->next
   beqz $t0, epi         # end test: list->next == 0
   jal evens             # evens(list)
   add $a0, $v0, $zero   # ret val becomes arg
   jal mergesort         # mergesort(evens(list))
   sw $v0, 8($sp)        # save the result
   lw $a0, 4($sp)        # list (my arg)
   jal odds              # odds(list)
```

Translating C to MIPS

```
add $v0, $a0, $zero      # body:
   beqz $a0, epi         # end test: list == 0
   lw $t0, 4($a0)        # list->next
   beqz $t0, epi         # end test: list->next == 0
   jal evens             # evens(list)
   add $a0, $v0, $zero   # ret val becomes arg
   jal mergesort         # mergesort(evens(list))
   sw $v0, 8($sp)        # save the result
   lw $a0, 4($sp)        # list (my arg)
   jal odds              # odds(list)
   add $a0, $v0, $zero   # ret val becomes arg
   jal mergesort         # mergesort(odds(list))
   add $a1, $v0, $zero   # ret val becomes 2nd arg
   lw $a0, 8($sp)        # saved result becomes arg
   jal merge             # call merge
```

Translating C to MIPS

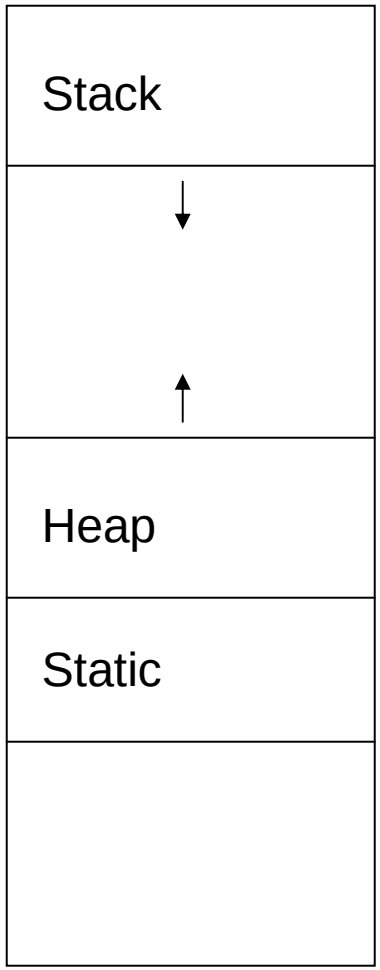
```
epi:
   lw $ra, 0($sp)          # epilogue:
   addi $sp, 12            # restore $ra and stack
   jr $ra                  # return
```

Memory Management

Stack: local variables, grows down

Heap: malloc and free, grows up

Static: global variables, fixed size



Memory (Heap) Management

- When allocating and freeing memory on the heap, we need a way to manage free blocks of memory.

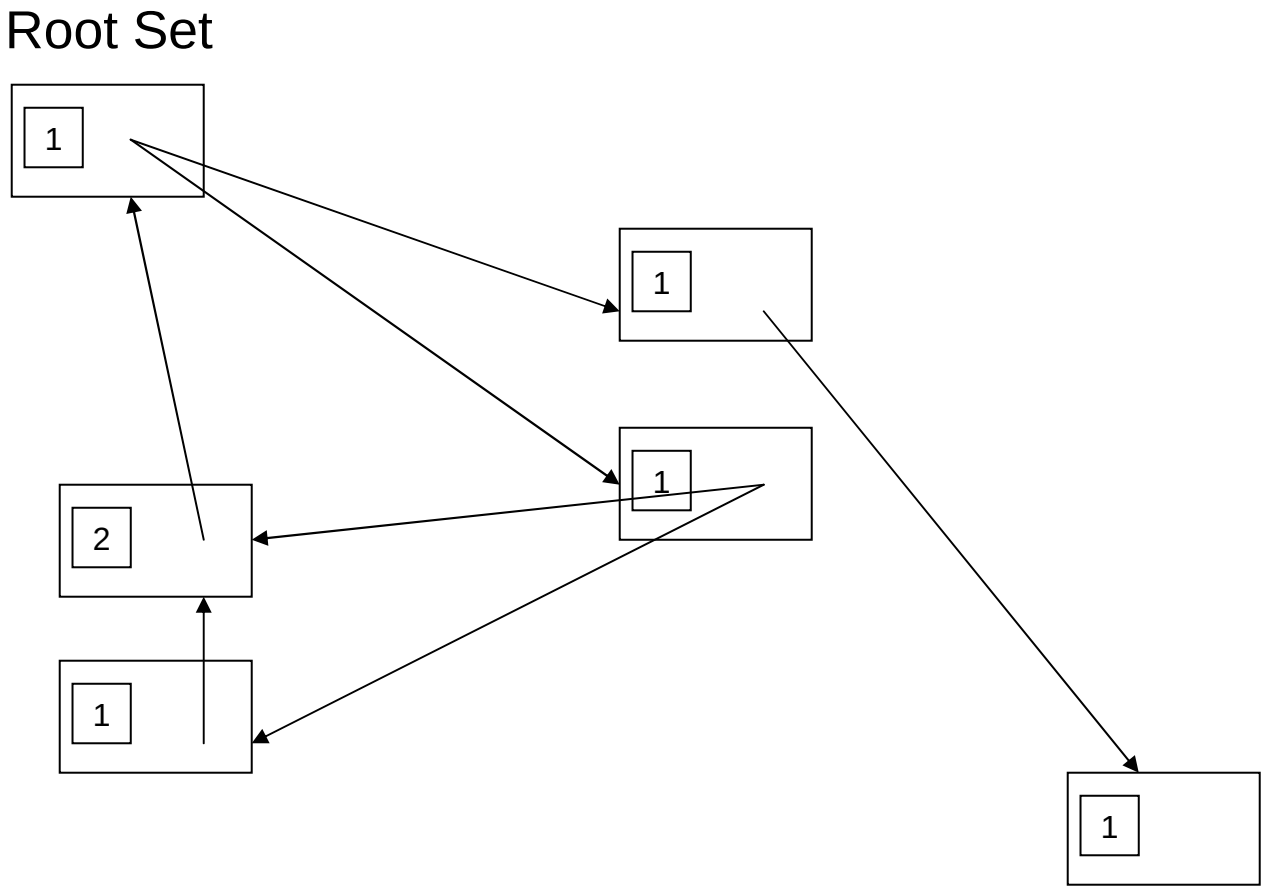
- Lecture covered two different ways to manage free blocks of memory.
    - Free List (first fit, next fit, best fit)
    - Buddy System

Garbage Collection

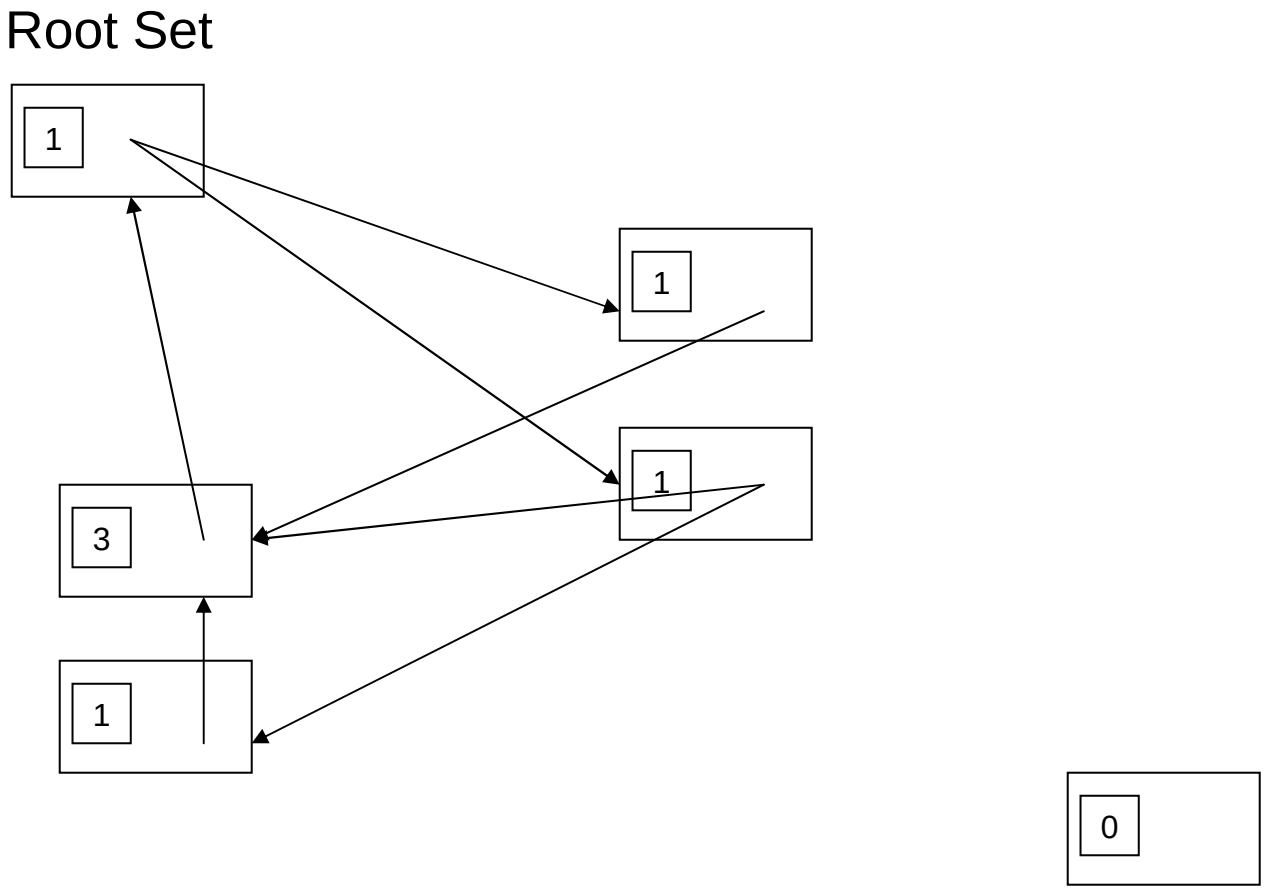
- Garbage collection is used for cleaning up the heap. Sacrifices performance for automatic memory management.

- Lecture covered three different ways to garbage collect.
    - Reference Count
    - Mark and Sweep
    - Stop and Copy

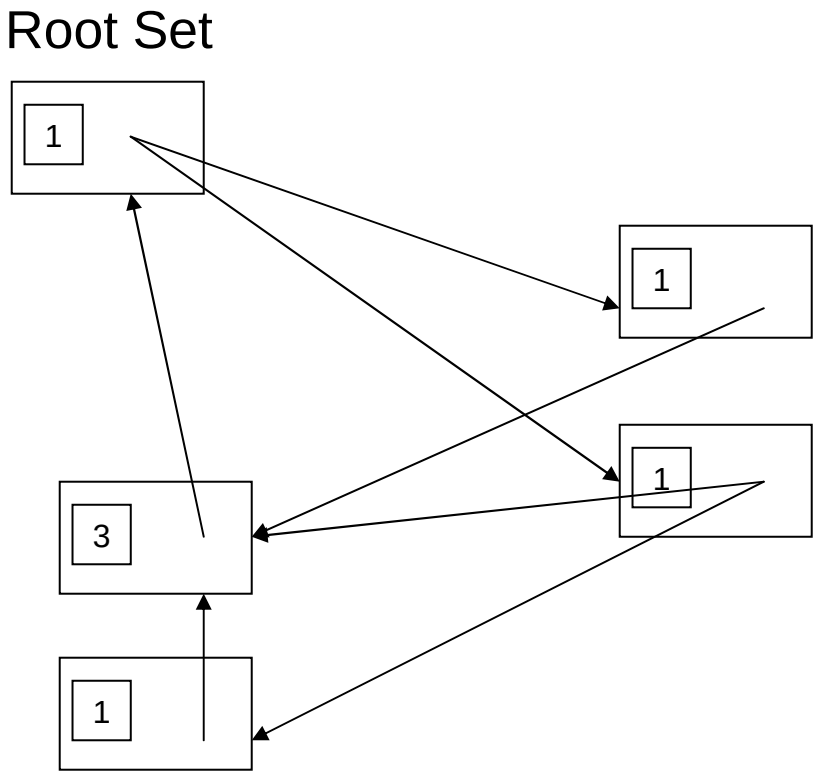
Reference Count

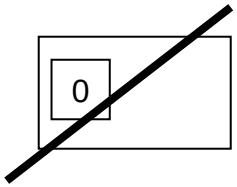


Reference Count



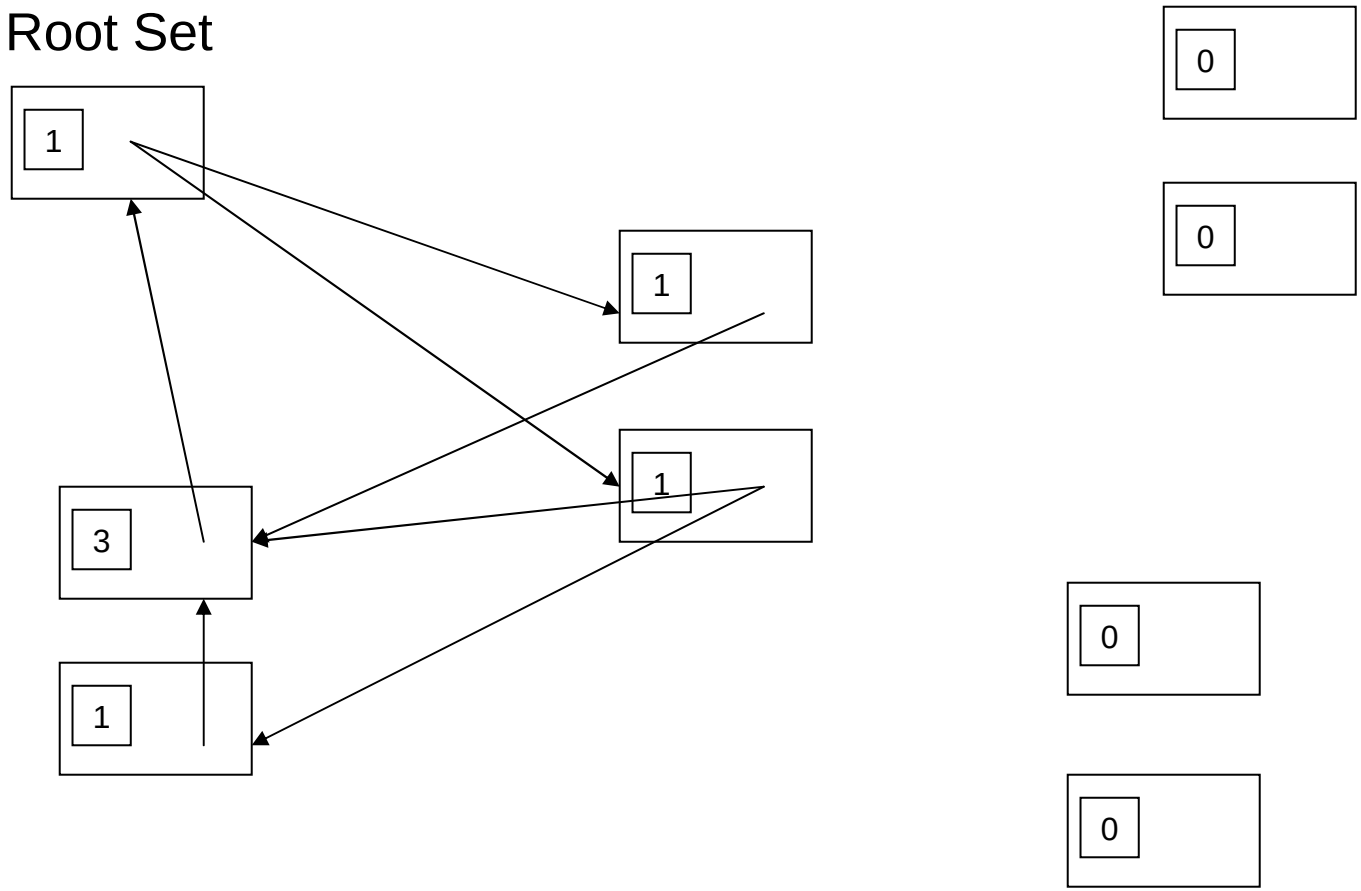
Reference Count



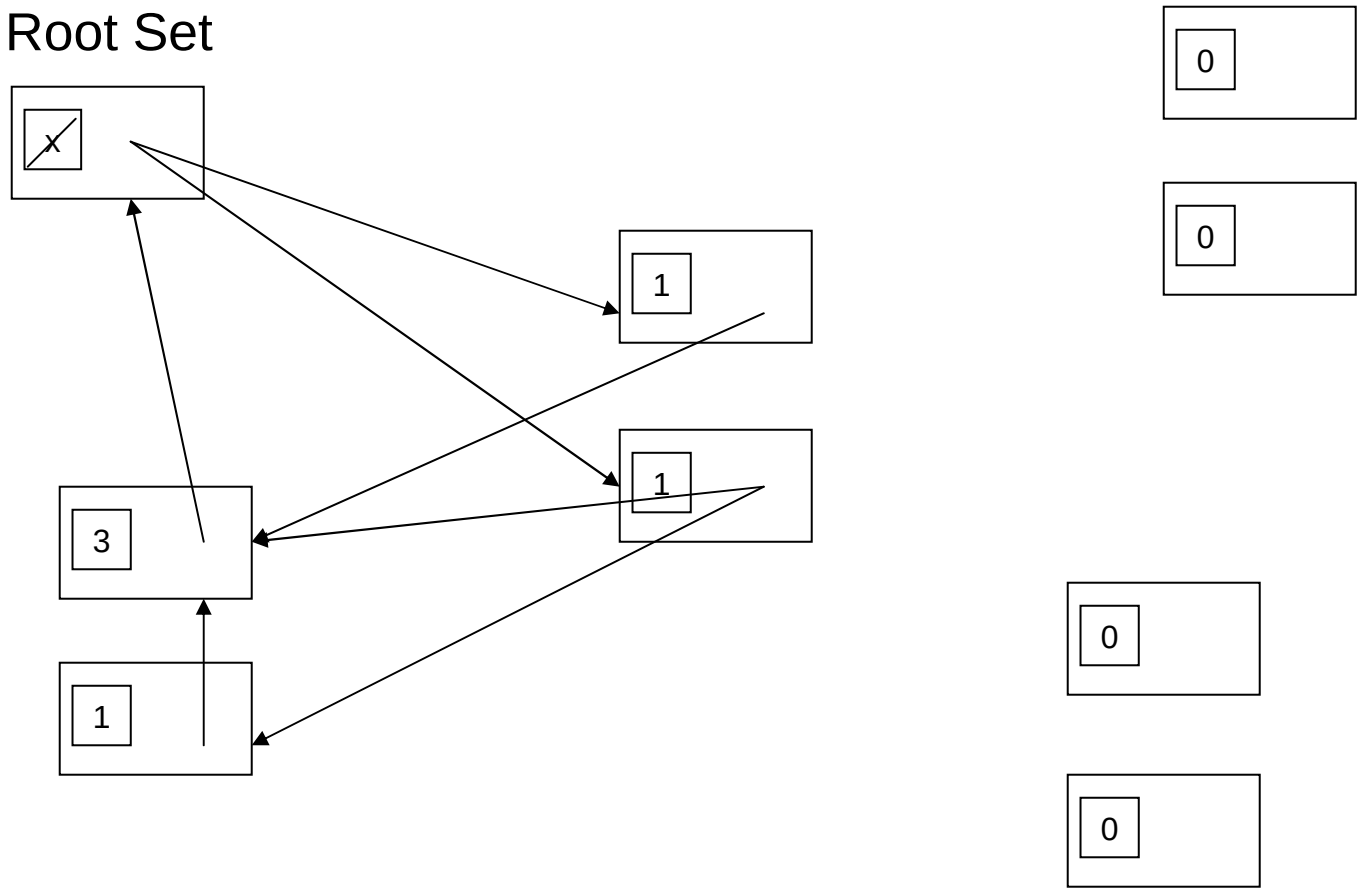


Lots of overhead – every time a pointer changes,
the count changes. Unused cycles are never retrieved!

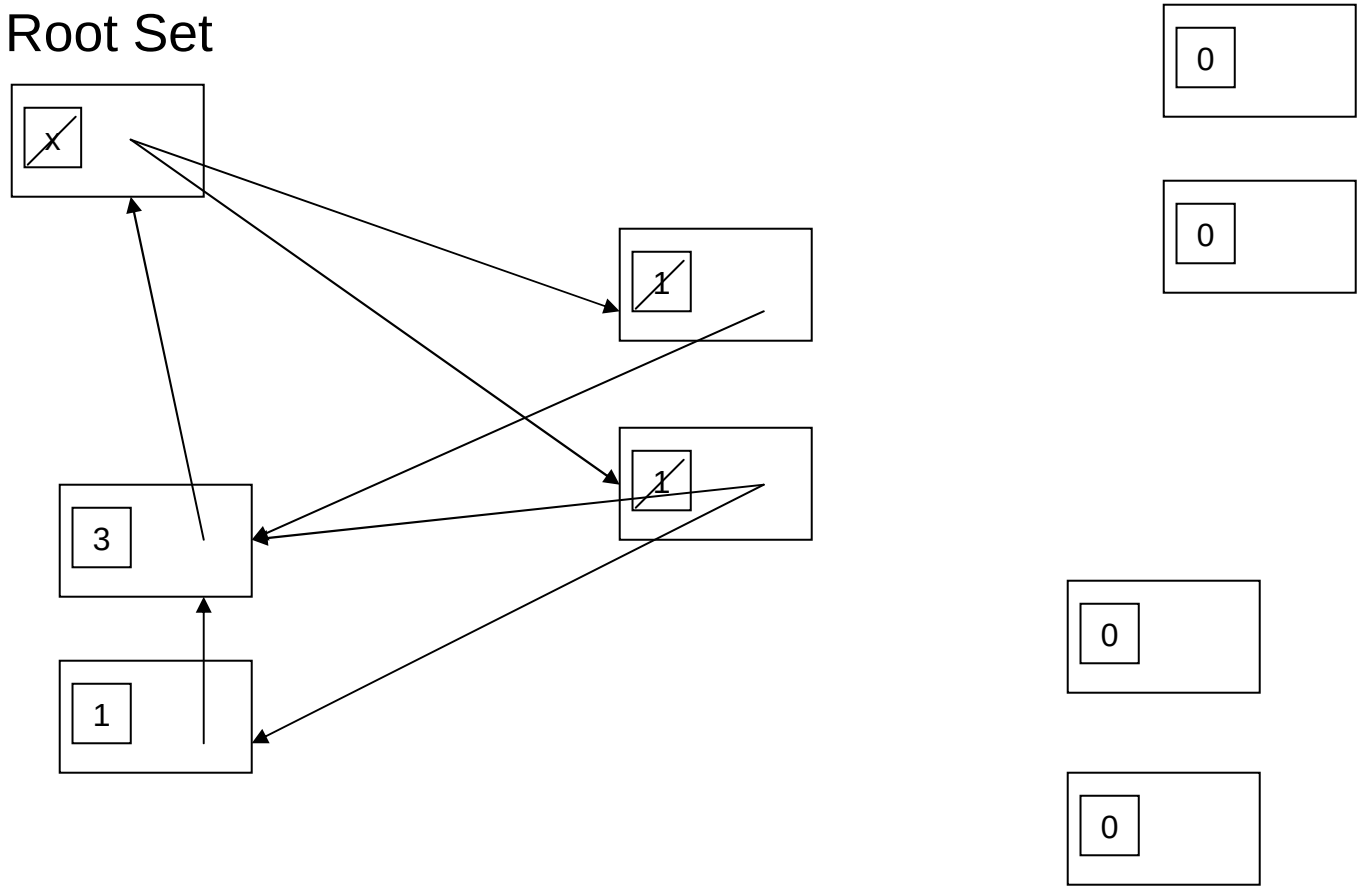
Mark and Sweep



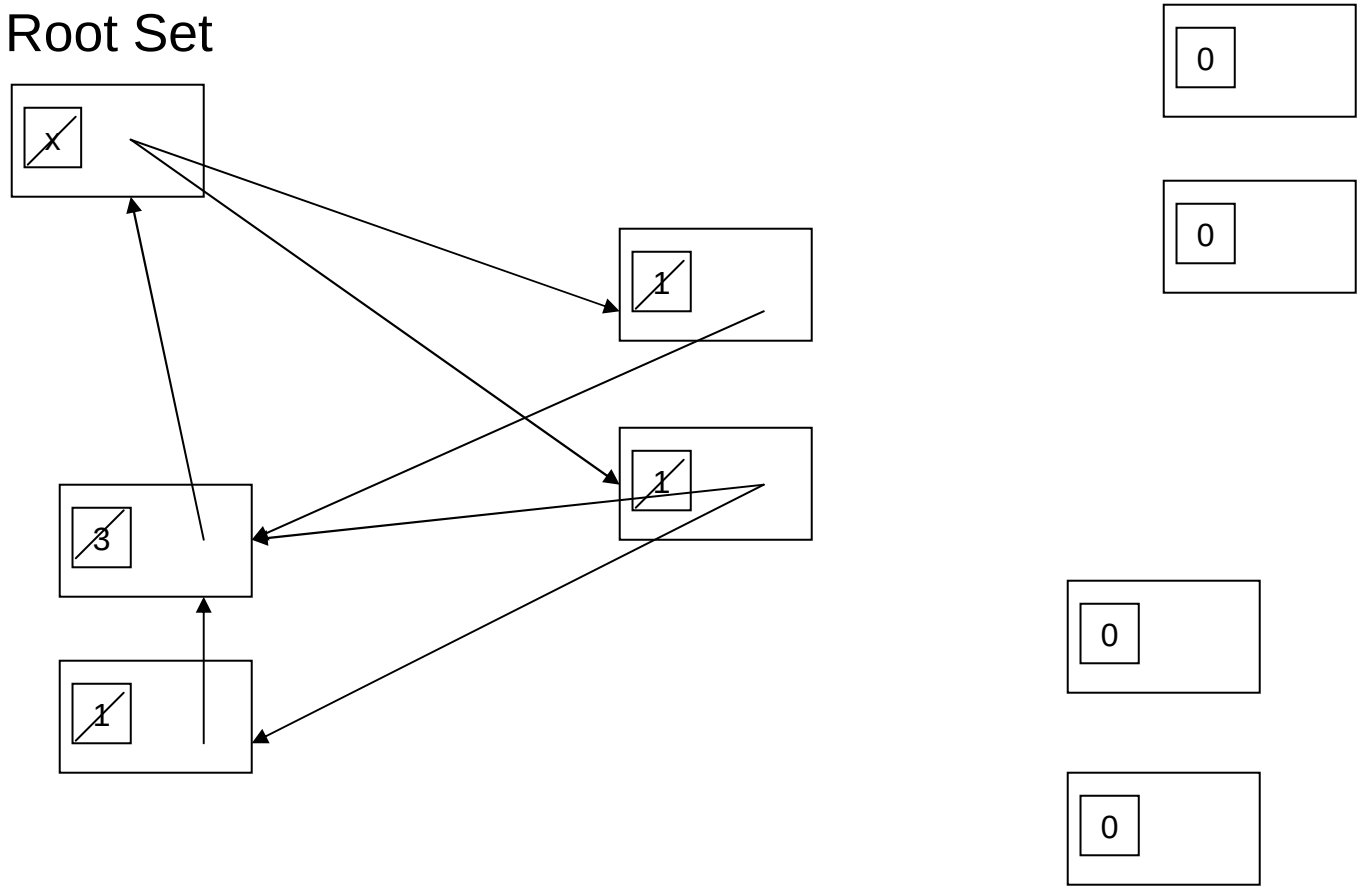
Mark and Sweep



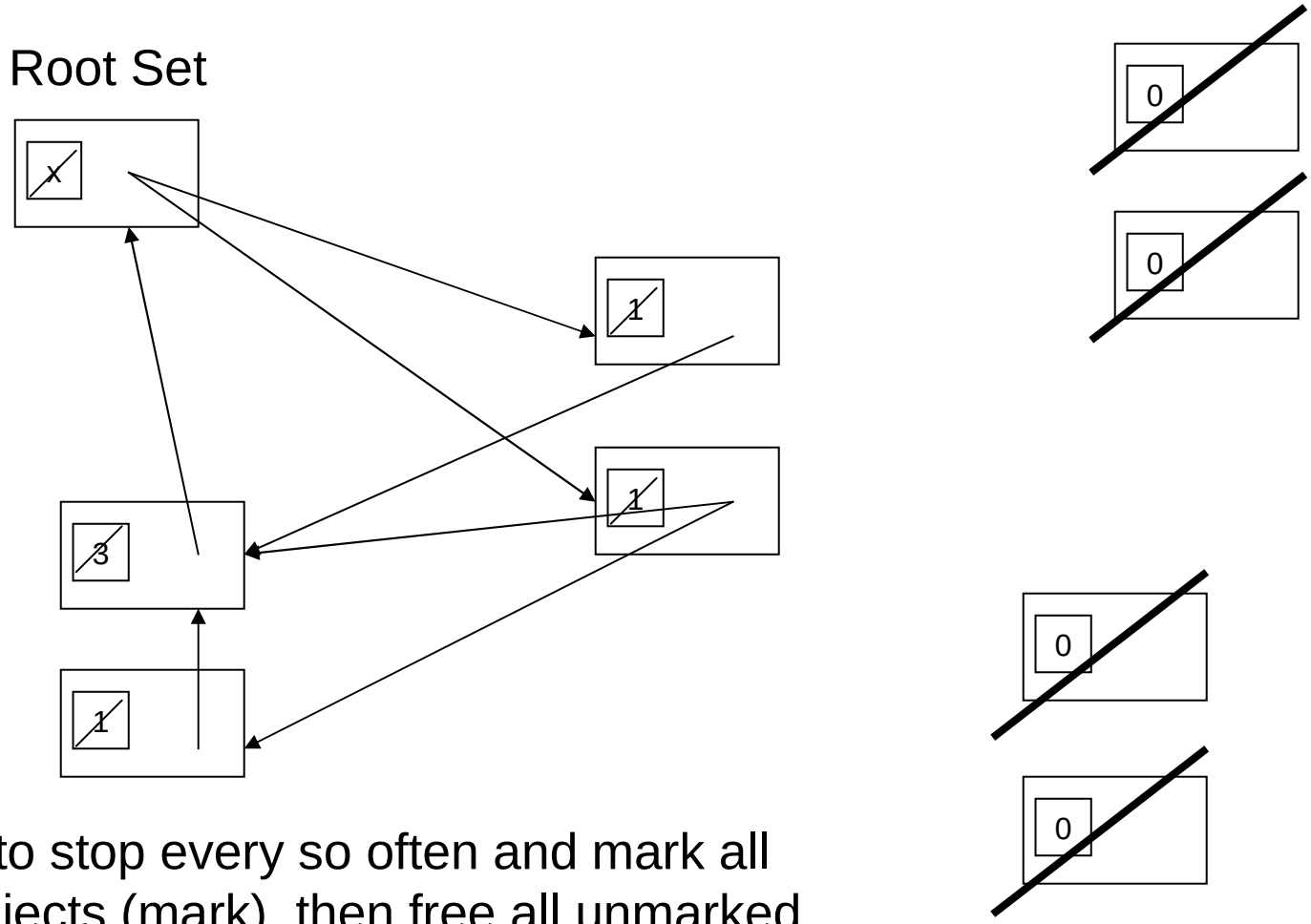
Mark and Sweep



Mark and Sweep

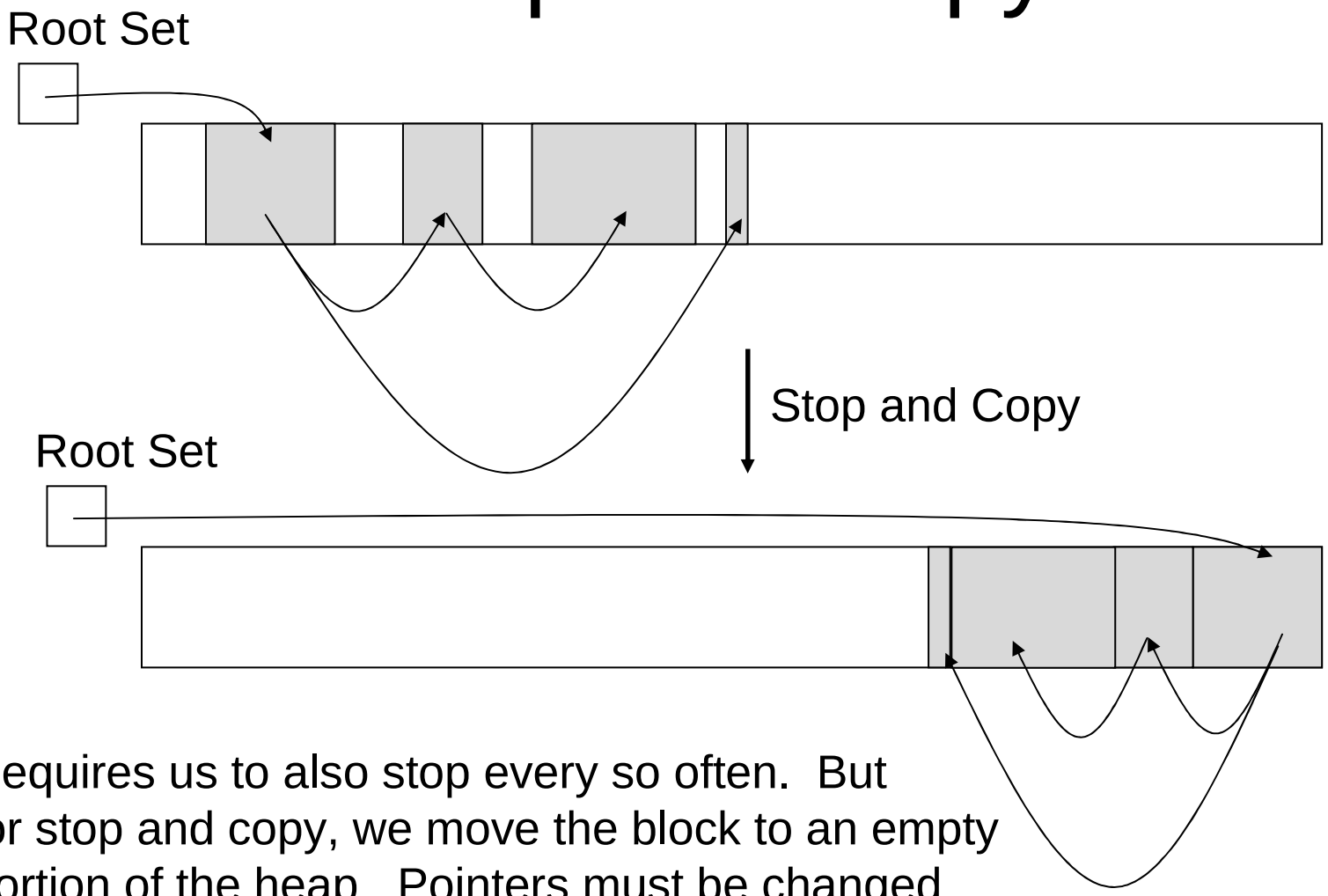


Mark and Sweep



Requires us to stop every so often and mark all reachable objects (mark), then free all unmarked blocks (sweep). Once mark and sweep is done, unmark everything!

Stop and Copy



Requires us to also stop every so often. But for stop and copy, we move the block to an empty portion of the heap. Pointers must be changed to reflect the change in block location.