Representing Data Structures

Multidimensional arrays

Structs

Array Layout and Indexing

int val[5];

+0	+4	+8	+12	+16

Write x86 code to load val[i] into %rax.
1. Assume:
   - Base address of val **is in** %rdi
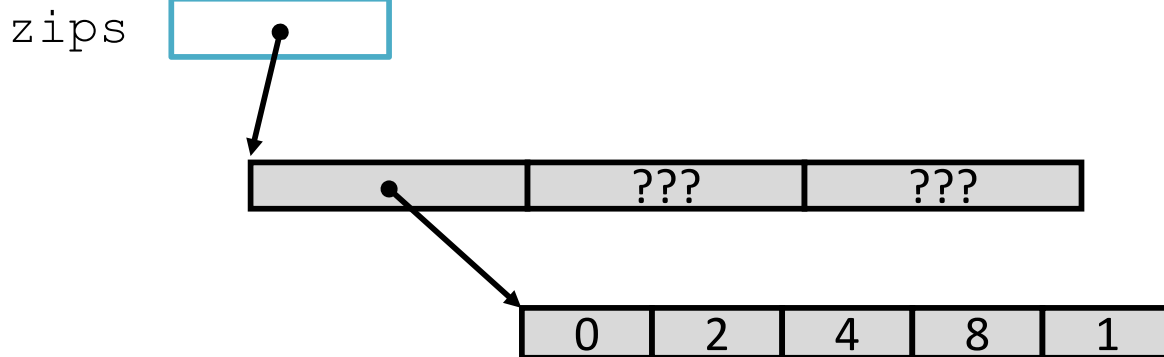   - i is in %rsi
2. Assume:
   - Base address of val **is** 28(%rsp)
   - i is in %rcx

C: Arrays of pointers to arrays of …

```
int** zips = (int**)malloc(sizeof(int*)*3);
...
zips[0] = (int*)malloc(sizeof(int)*5);
...
int* zip0 = zips[0];
zip0[0] = 0;
zips[0][1] = 2;
zips[0][2] = 4;
zips[0][3] = 8;
zips[0][4] = 1;
```

Write x86 code to implement:
```
void copyleft(int** zips, long i, long j){
    zips[i][j] = zips[i][j - 1];
}
```
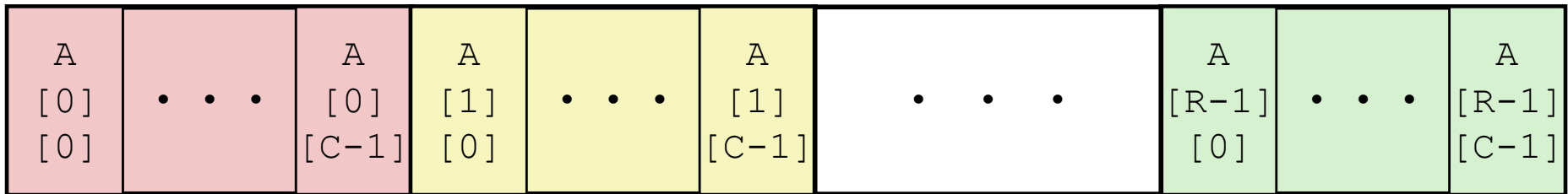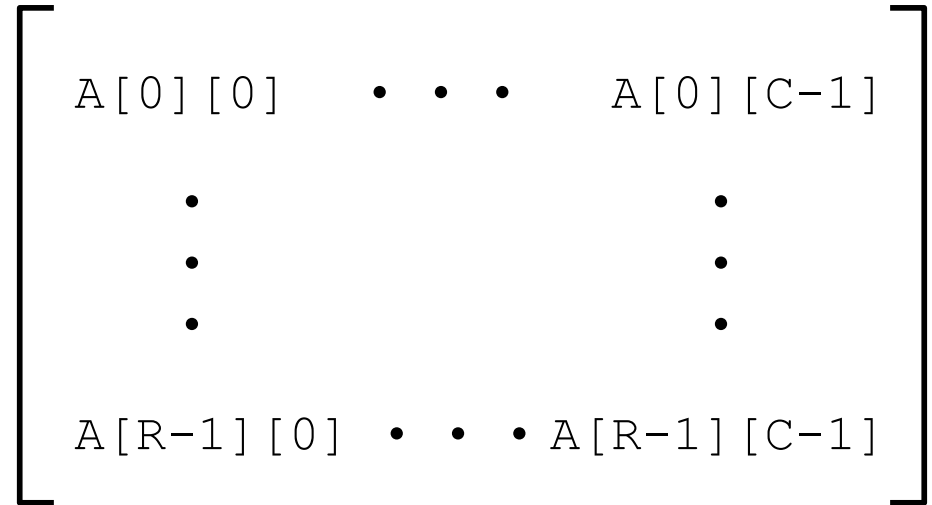
zips

	???	???

0	2	4	8	1

```
int[][] zips = new int[3][];
zips[0] = new int[5] {0, 2, 4, 8, 1};
```
Java

Row-Major Nested Arrays

$$\begin{bmatrix} A[0][0] & \cdots & A[0][C-1] \\ \vdots & & \vdots \\ A[R-1][0] & \cdots & A[R-1][C-1] \end{bmatrix}$$

```
int a[R][C];
```

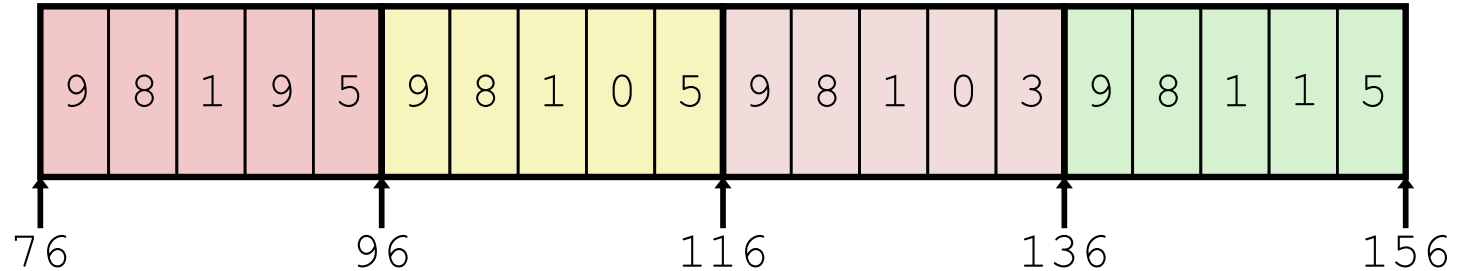| A [0] [0]	⋯	A [0] [C-1]	A [1] [0]	⋯	A [1] [C-1]	⋯	A [R-1] [0]	⋯	A [R-1] [C-1]

&a[i][j] is **a +** _____

```
int* b = (int*)a;   // Can treat as larger 1D array

&a[i][j] == &b[_____]
```

Strange Referencing Examples

`int sea[4][5];`

| | 9 | 8 | 1 | 9 | 5 | 9 | 8 | 1 | 0 | 5 | 9 | 8 | 1 | 0 | 3 | 9 | 8 | 1 | 1 | 5 | |

76   96   116   136   156

Reference	Address	Value	Guaranteed?
`sea[3][3]`	`76+20*3+4*3  = 148`	1	Yes
`sea[2][5]`			
`sea[2][-1]`			
`sea[4][-1]`			
`sea[0][19]`			
`sea[0][-1]`			

C does not do any bounds checking.

Row-major array layout is guaranteed.

C structs

```
struct rec {
    int i;
    int a[3];
    int* p;
};
```

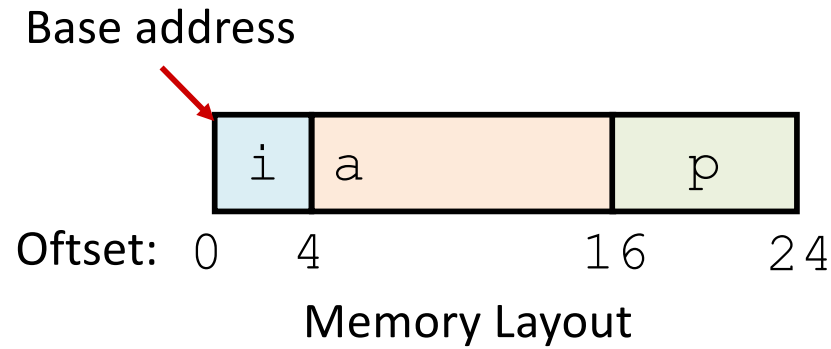Base address

i | a | p

Oftset: 0   4   16   24

Memory Layout

Like Java class/object without methods.

Compiler determines:
- Total size
- Offset of each field

```
struct rec x;
struct rec y;
x.i = 1;
x.a[1] = 2;
x.p = &(x.i);
// copy full struct:
y = x;

struct rec* z;
z = &y;
(*z).i++;
// same as:
z->i++
```
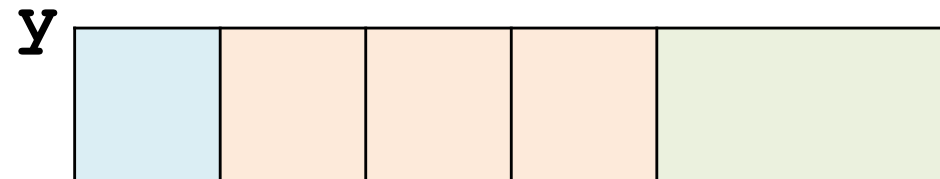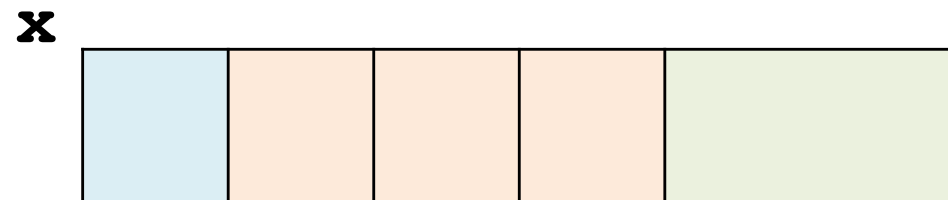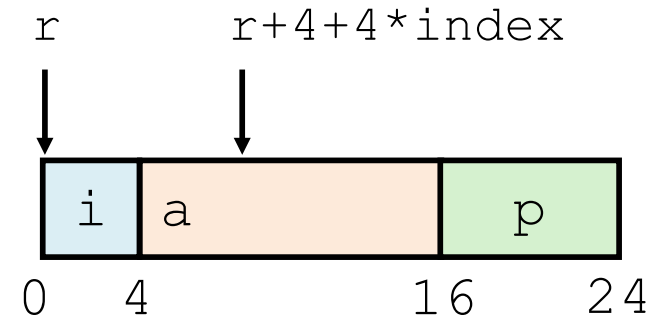
x

y

z
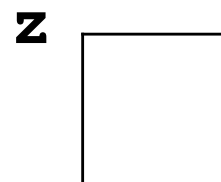
Write x86.

Accessing Struct Field

```
struct rec {
    int i;
    int a[3];
    int* p;
};
```



```
int get_i_plus_elem(struct rec* r, int index) {
    return r->i + r->a[index];
}
```

```
movl 0(%rdi),%eax            # Mem[r+0]
addl 4(%rdi,%rsi,4),%eax     # Mem[r+4*index+4]
retq
```

typedef

```
// give type T another name: U
typedef T U;


// struct types can be verbose
struct ListNode { ... };
...
struct ListNode* n = …;


// typedef can help
typedef struct list_node {

        ...

} ListNode;
...
ListNode* n = ...;
```

Struct alignment (1)

Unaligned Data


```
struct S1 {
    char c;
    double v;
    int i;
}* p;
```

Defines new struct type and declares variable p of type struct S1*
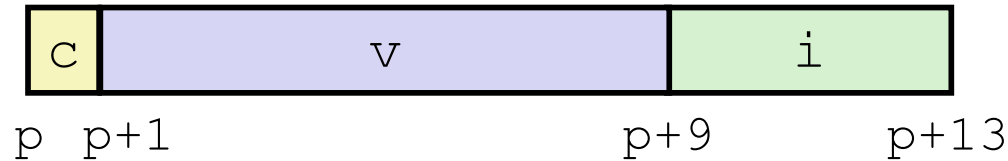
Aligned Data

Primitive data type requires **K** bytes

Address must be multiple of **K**

C: align every struct field accordingly.

internal fragmentation

Multiple of 8

Multiple of 4
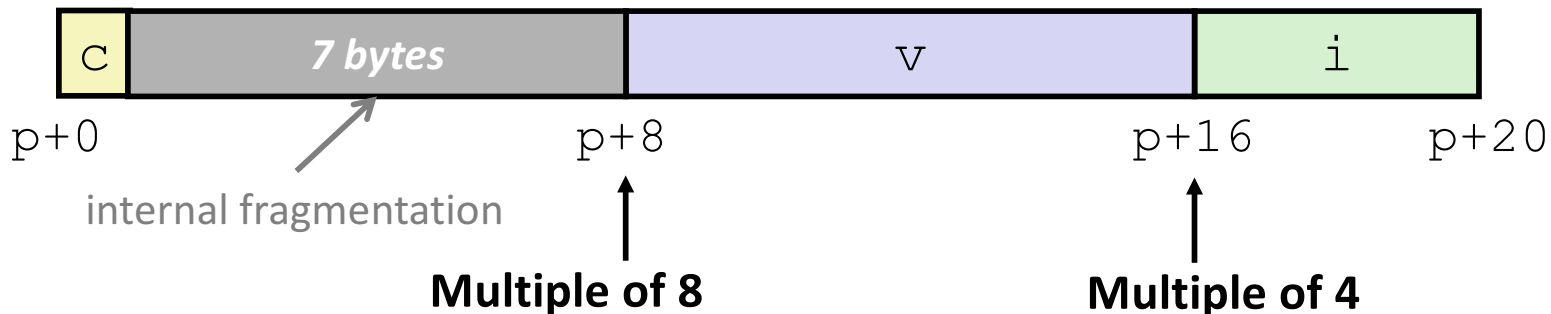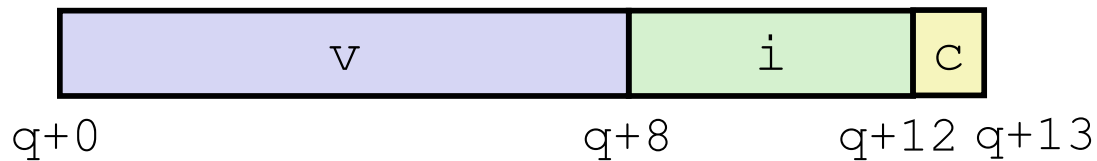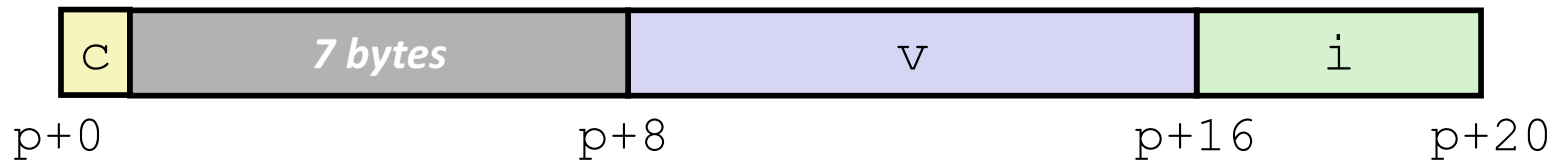
Struct packing

Put large data types first:

```
struct S1 {
    char c;
    double v;
    int i;
} * p;
```

→ programmer →

```
struct S2 {
    double v;
    int i;
    char c;
} * q;
```

c	7 bytes	v	i

p+0   p+8   p+16   p+20

v	i	c

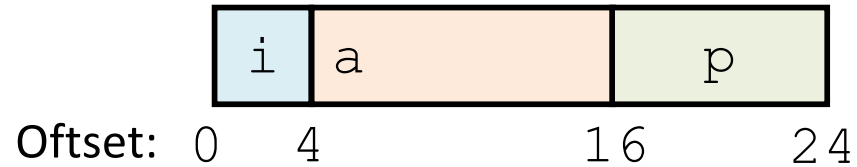q+0   q+8   q+12 q+13

But actually…

Struct alignment (full)

- Base *and total size* must align largest internal primitive type.
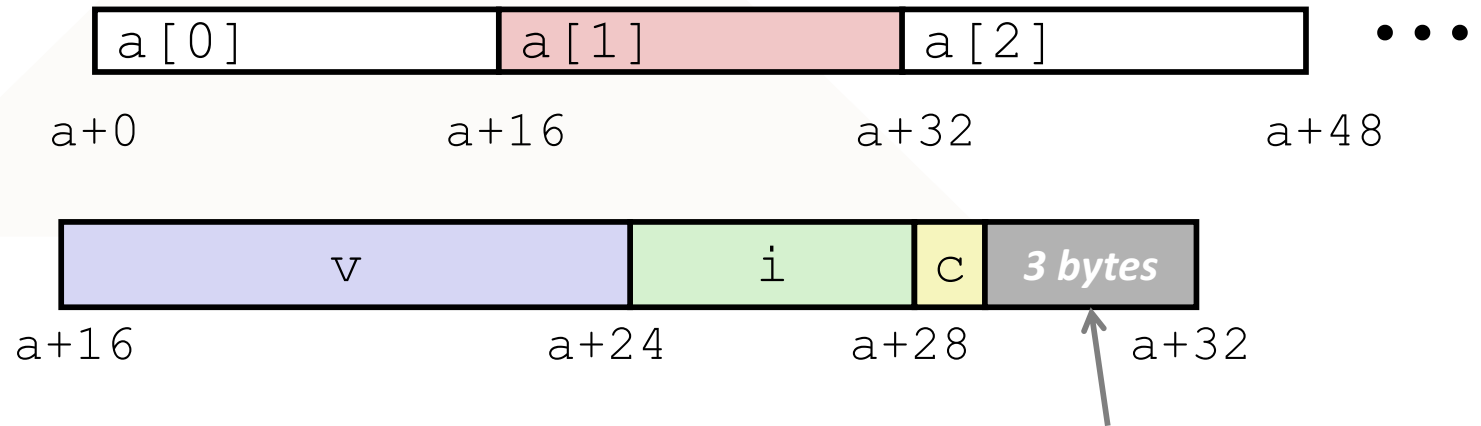- Fields must align their type's largest alignment requirement.

c	7 bytes	v	i

p+0　　　　　　　　　　p+8　　　　　　　　p+16　　　p+20

v	i	c	3 bytes

q+0　　　　　　　　q+8　　　q+12　　　q+16
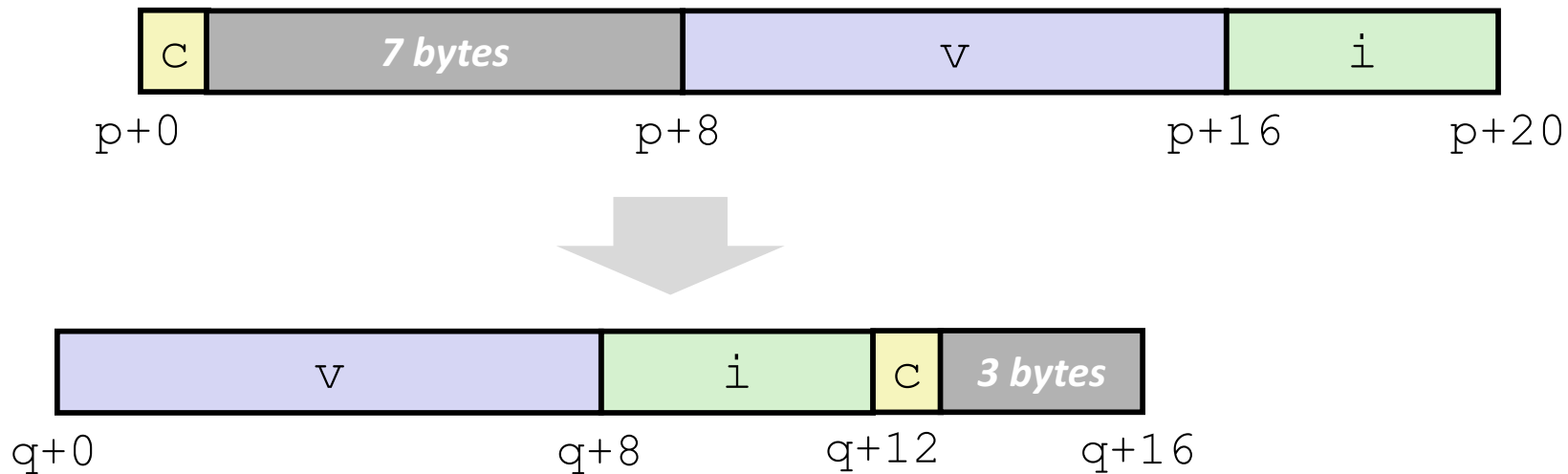
Array in struct

```
struct rec {
    int i;
    int a[3];
    int* p;
};
```

i	a	p

Oftset: 0   4   16   24

Struct in array

```
struct S2 {
    double v;
    int i;
    char c;
} a[10];
```

a[0]	a[1]	a[2]	• • •

a+0   a+16   a+32   a+48

v	i	c	3 bytes

a+16   a+24   a+28   a+32

external fragmentation

Linked Lists

head

pointers/references

```
typedef
struct ListNode {
    ListNode* next;
    int value;
} ListNode;
```
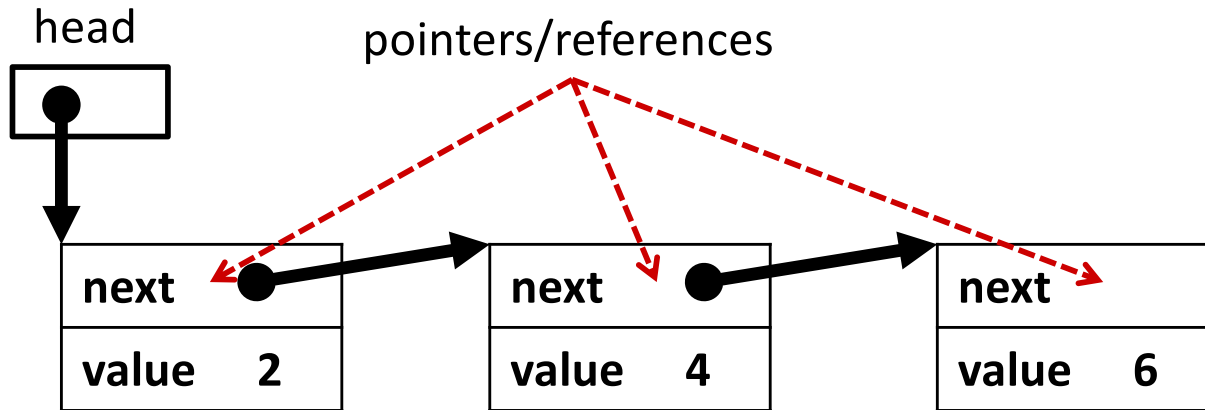

next	
value	2

next	
value	4

next	
value	6

1. Choose memory layout for ListNode

Linked Lists

head

pointers/references

```
typedef
struct ListNode {
    ListNode* next;
    int value;
} ListNode;
```
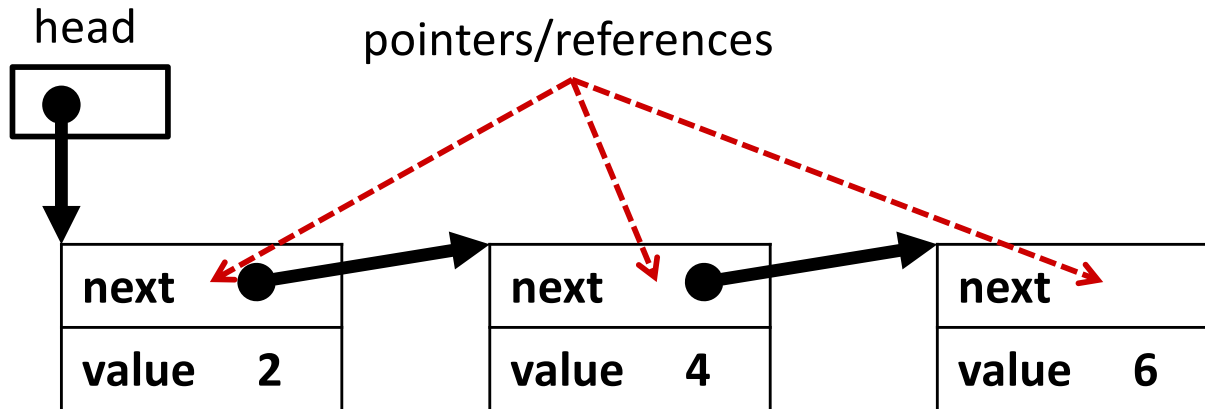
next	
value	2

next	
value	4

next	
value	6

2. Implement append in x86:

```
void append(ListNode* head, int x) { // assume head != NULL
  ListNode* cursor = head;
  while (cursor->next != NULL) {      // find tail
    cursor = cursor->next;
  }
  ListNode* n = (ListNode*)malloc(sizeof(ListNode));
  // error checking omitted for x86 simplicity
  cursor->next = n;
  n->next = NULL;
  n->value = x;
}
```

Try a recursive version too.