Algorithms, Data Structures, and Problem Solving

Masoumeh Taromirad

Hamlstad University

HÖGSKOLAN I HALMSTAD

DT4002, Fall 2016

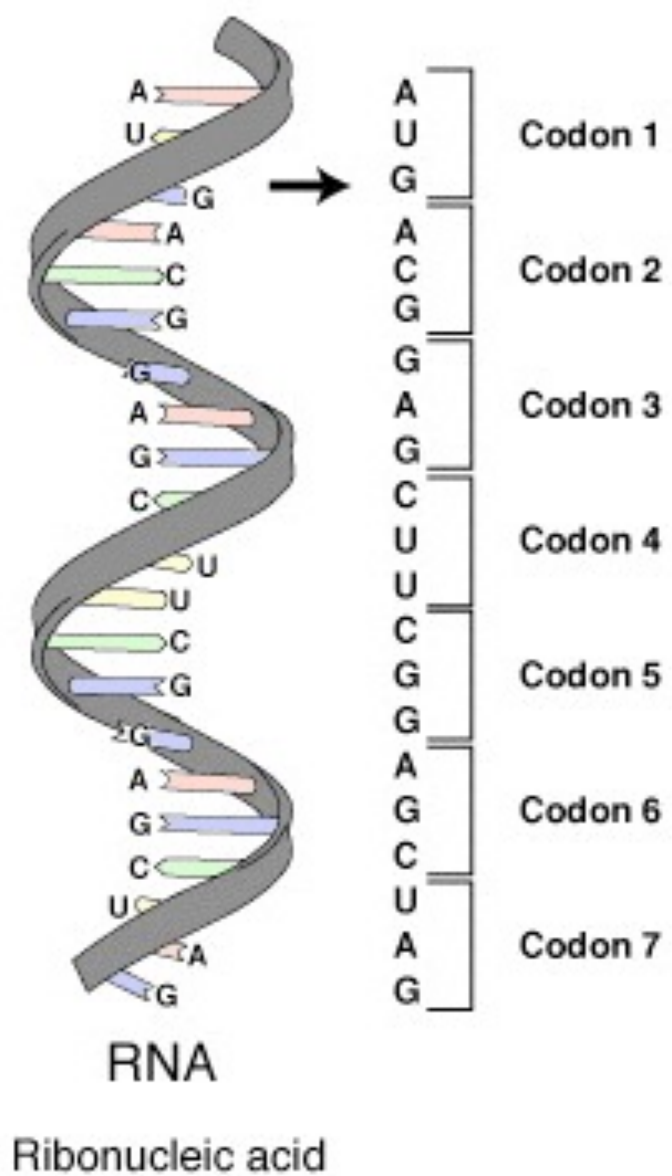
Container Concepts

- containers store data
- container operations:
  - insertion
  - retrieval
  - removal
  - iteration
- possible organization types
  - sequential
  - associative
  - “unorganized”

Conceptual Overview

Sequence Containers

http://en.wikipedia.org/wiki/File:RNA-codons.png



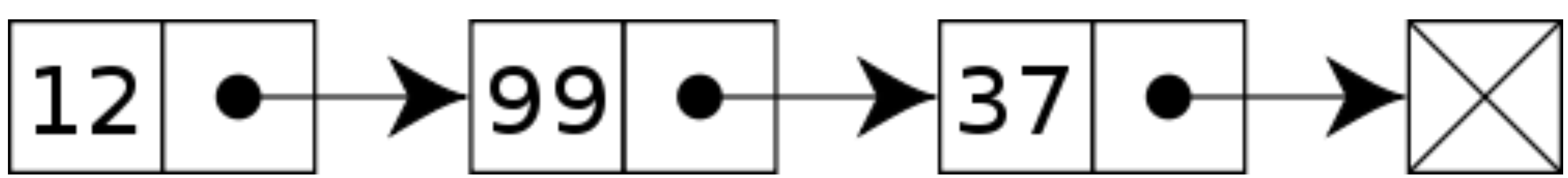
http://en.wikipedia.org/wiki/File:Toppledominos.jpg

serial arrangement of items

Conceptual Overview

Sequence Containers

- common implementation approaches:
  - array
  - linked list
    - singly or doubly linked
    - linear or circular



http://en.wikipedia.org/wiki/File:Singly-linked-list.svg

Conceptual Overview

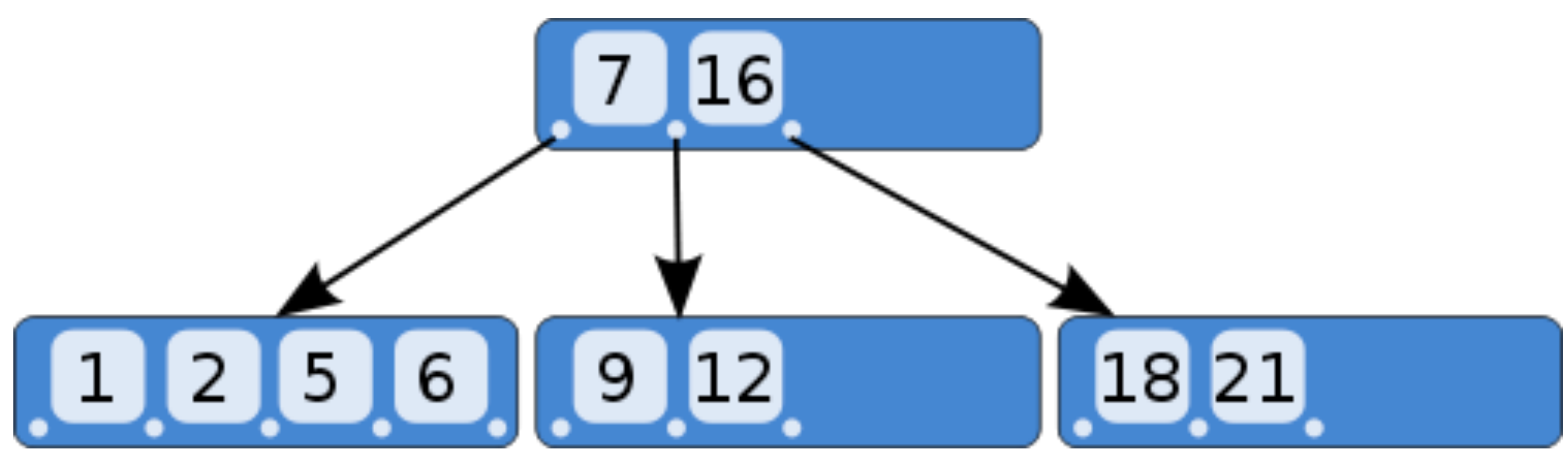
Associative Containers

connect each item with a key

Conceptual Overview

Associative Containers

- common implementation approaches:
  - tree
    - binary, k-ary, multiway, ...
    - balanced / complete / ...
  - hash table
  - ...



http://en.wikipedia.org/wiki/File:B-tree.svg

Conceptual Overview

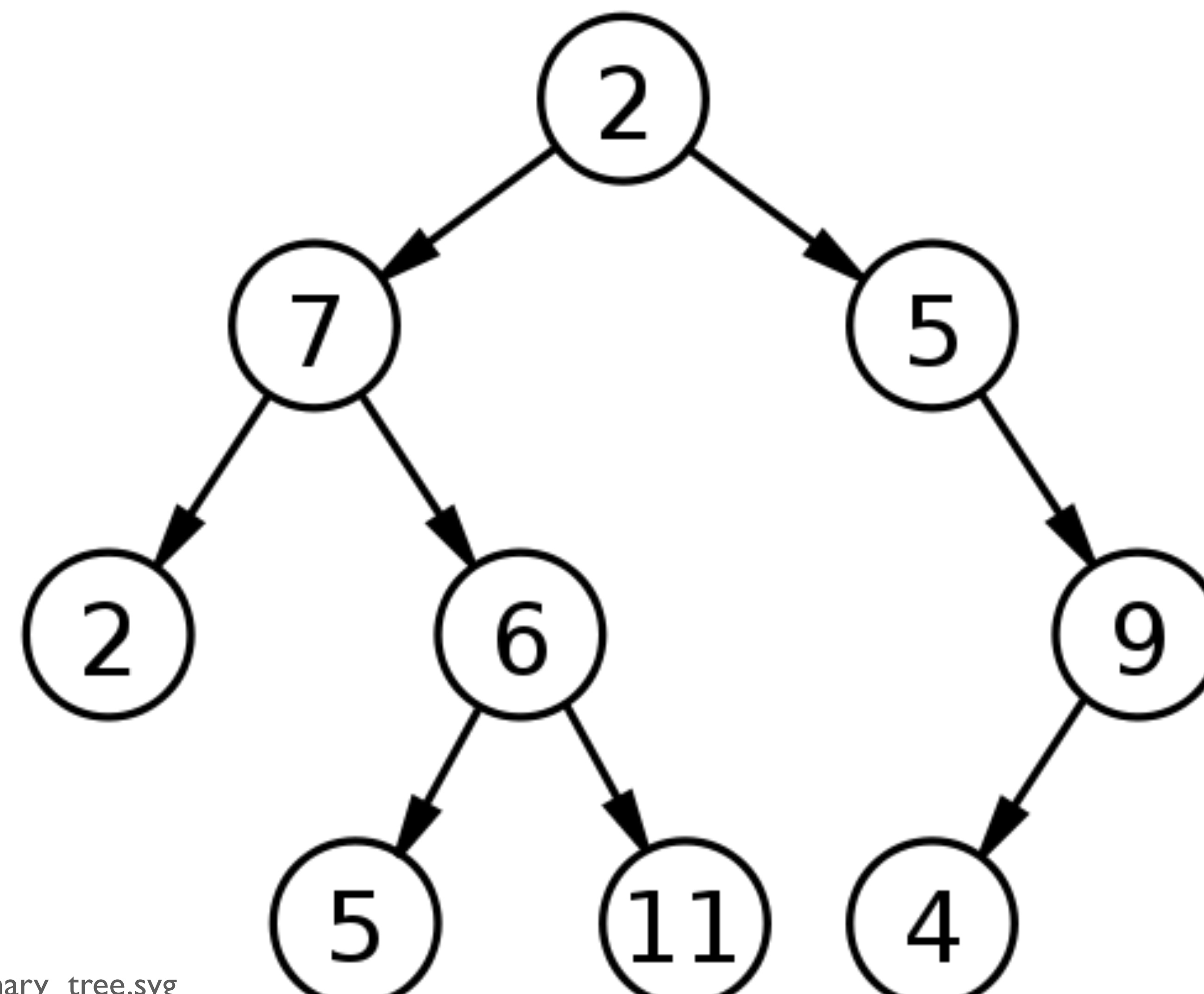
"Unorganized" Containers

- no particular sequence or association
- internal structure depends on desired properties
- example: each item should be unique

Conceptual Overview

"Unorganized" Containers

- common implementation approaches:
  - tree
  - hash table
  - array
  - list



http://en.wikipedia.org/wiki/File:Binary_tree.svg

Conceptual Overview

Iteration

- a generic way of visiting each item
- the iteration order depends on:
  - container organization type
  - container implementation
  - iteration algorithm
- classification:
  - uni-directional
  - bi-directional
  - random access

Today

Common Sequence Containers

Vectors and Lists

- vectors are arrays that grow and shrink
- lists are containers where each item points to its successor *(and sometimes predecessor)*

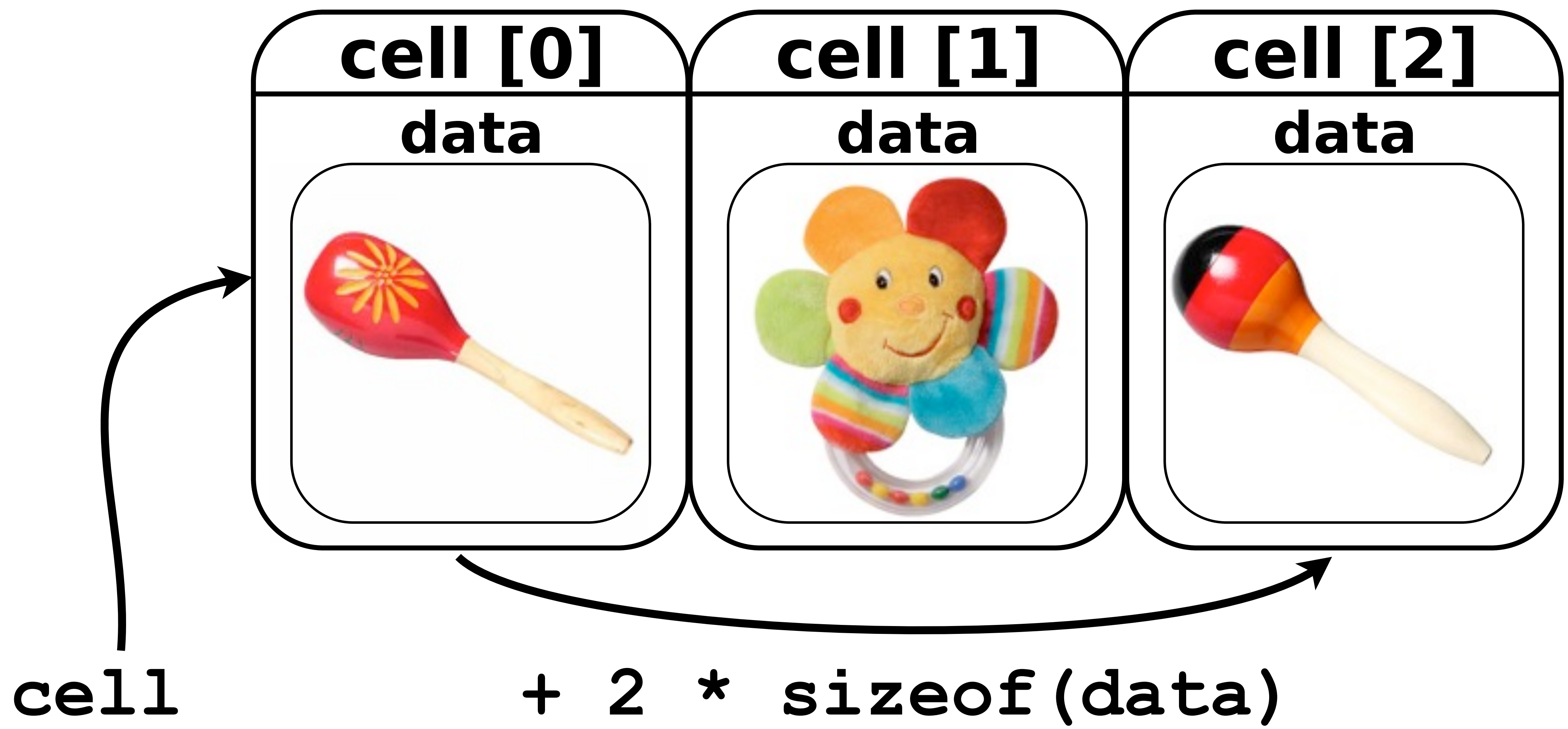
Implementation Overview

Arrays

- store user data in consecutive cells (memory locations)
- pre-allocate enough space
- the array "is" the address of the first item
- address offset of each item
  `offset = index * sizeof(item)`

Implementation Overview

Arrays



Implementation Overview

Arrays

- advantages:
  - very simple
  - lightweight
  - random access
- potential drawbacks:
  - fixed capacity
  - insertion and removal can be costly

Implementation Overview

Vectors (dynamic arrays)

- advantages:
  - very simple
  - lightweight
  - <u>random access</u>
- potential drawbacks:
  - ~~fixed capacity~~
  - insertion and removal can be costly

Implementation Overview

Vectors

- dynamic size:
  - get more memory when the array grows
  - *(optional)* give back memory when it shrinks

▶ we need to:

- track the capacity
- track the current size
- manage memory
- *(maybe)* copy the contents after growing

Implementation Overview

Vectors

- dynamic size:
  - get more memory when the array grows
  - *(optional)* give back memory when it shrinks

▸ we need to:
  - track the capacity
  - track the current size
  - manage memory → **`malloc`, `realloc`, `free`**
  - *(maybe)* copy the contents after growing → **`memcpy`**

Implementation Sketch

Vectors

```
unsigned int cap;
unsigned int len;
int * arr;

/* ... */

if (len >= cap) {
  newcap = 2*cap;
  newarr = malloc (newcap*sizeof(int));
  memcpy (newarr, arr, len*sizeof(int));
  free (arr);
  arr = newarr;
  cap = newcap;
}
arr[len] = val;
++len;
```

Implementation Sketch

Vectors

```
unsigned int cap;
unsigned int len;
int * arr;

/* ... */

if (len >= cap) {
  newcap = 2*cap;
  newarr = malloc (newcap*sizeof(int));
  memcpy (newarr, arr, len*sizeof(int));
  free (arr);
  arr = newarr;
  cap = newcap;
}
arr[len] = val;
++len;
```

vector contents:

index	0	1	2	
value	42	77	-123	

```
unsigned int cap;
unsigned int len;
int * arr;

/* ... */

if (len >= cap) {
   newcap = 2*cap;
   newarr = malloc (newcap*sizeof(int));
   memcpy (newarr, arr, len*sizeof(int));
   free (arr);
   arr = newarr;
   cap = newcap;
}
arr[len] = val;
++len;
```

address	value	
0124	4	=cap
0128	3	=len
012c	3b50	=arr

this is a **pointer**

address	value
3b50	42
3b54	77
3b58	-123
3b60	

vector contents:

index	0	1	2	
value	42	77	-123	

```
unsigned int cap;
unsigned int len;
int * arr;
```

address	value	
0124	4	=cap
0128	3	=len
012c	3b50	=arr

address	value
3b50	42
3b54	77
3b58	-123
3b60	

let's append 321

false

```
if (len >= cap) {
   newcap = 2*cap;
   newarr = malloc (newcap*sizeof(int));
   memcpy (newarr, arr, len*sizeof(int));
   free (arr);
   arr = newarr;
   cap = newcap;
}
arr[len] = val;
++len;
```

vector contents:

index	0	1	2	
value	42	77	-123	

```
unsigned int cap;
unsigned int len;
int * arr;
```

address	value	
0124	4	=cap
0128	**4**	=len
012c	3b50	=arr

let's append 321

address	value
3b50	42
3b54	77
3b58	-123
3b60	**321**

```
if (len >= cap) {
   newcap = 2*cap;
   newarr = malloc (newcap*sizeof(int));
   memcpy (newarr, arr, len*sizeof(int));
   free (arr);
   arr = newarr;
   cap = newcap;
}
arr[len] = val;
++len;
```

vector contents:

index	0	1	2	3
value	42	77	-123	321

```
unsigned int cap;
unsigned int len;
int * arr;
```

address	value	
0124	4	=cap
0128	4	=len
012c	3b50	=arr

address	value
3b50	42
3b54	77
3b58	-123
3b60	321

let's append -21

```
if (len >= cap) {
  newcap = 2*cap;
  newarr = malloc (newcap*sizeof(int));
  memcpy (newarr, arr, len*sizeof(int));
  free (arr);
  arr = newarr;
  cap = newcap;
}
arr[len] = val;
++len;
```

true

vector contents:

index	0	1	2	3
value	42	77	-123	321

```
unsigned int cap;
unsigned int len;
int * arr;
```

address	value	
0124	4	=cap
0128	4	=len
012c	3b50	=arr

let's append -21

```
if (len >= cap) {
   newcap = 2*cap;
   newarr = malloc (newcap*sizeof(int));
   memcpy (newarr, arr, len*sizeof(int));
   free (arr);
   arr = newarr;
   cap = newcap;
}
arr[len] = val;
++len;
```

address	value
3b50	42
3b54	77
3b58	-123
3b60	321

address	value
a534	
a538	
a53c	
a540	
a544	
a548	
a54c	
a550	

vector contents:

index	0	1	2	3
value	42	77	-123	321

```
unsigned int cap;
unsigned int len;
int * arr;
```

address	value	
0124	4	=cap
0128	4	=len
012c	3b50	=arr

address	value
3b50	42
3b54	77
3b58	-123
3b60	321

address	value
a534	**42**
a538	**77**
a53c	**-123**
a540	**321**
a544	
a548	
a54c	
a550	

let's append -21

```
if (len >= cap) {
   newcap = 2*cap;
   newarr = malloc (newcap*sizeof(int));
   memcpy (newarr, arr, len*sizeof(int));
   free (arr);
   arr = newarr;
   cap = newcap;
}
arr[len] = val;
++len;
```

vector contents:

index	0	1	2	3
value	42	77	-123	321

```
unsigned int cap;
unsigned int len;
int * arr;
```

address	value	
0124	4	=cap
0128	4	=len
012c	3b50	=arr

address	value
a534	42
a538	77
a53c	-123
a540	321
a544	
a548	
a54c	
a550	

let's append -21

```
if (len >= cap) {
   newcap = 2*cap;
   newarr = malloc (newcap*sizeof(int));
   memcpy (newarr, arr, len*sizeof(int));
   free (arr);
   arr = newarr;
   cap = newcap;
}
arr[len] = val;
++len;
```

vector contents:

index	0	1	2	3
value	42	77	-123	321

```
unsigned int cap;
unsigned int len;
int * arr;
```

address	value	
0124	**8**	=cap
0128	4	=len
012c	**a534**	=arr

address	value
a534	42
a538	77
a53c	-123
a540	321
a544	
a548	
a54c	
a550	

let's append -21

```
if (len >= cap) {
   newcap = 2*cap;
   newarr = malloc (newcap*sizeof(int));
   memcpy (newarr, arr, len*sizeof(int));
   free (arr);
   arr = newarr;
   cap = newcap;
}
arr[len] = val;
++len;
```

vector contents:

index	0	1	2	3			
value	42	77	-123	321			

```
unsigned int cap;
unsigned int len;
int * arr;
```

address	value	
0124	8	=cap
0128	**5**	=len
012c	a534	=arr

address	value
a534	42
a538	77
a53c	-123
a540	321
a544	**-21**
a548	
a54c	
a550	

let's append -21

```
if (len >= cap) {
   newcap = 2*cap;
   newarr = malloc (newcap*sizeof(int));
   memcpy (newarr, arr, len*sizeof(int));
   free (arr);
   arr = newarr;
   cap = newcap;
}
arr[len] = val;
++len;
```

vector contents:

index	0	1	2	3	4		
value	42	77	-123	321	-21		

remember to clean up
after yourself!
if you forget to **`free`**,
you'll create a <u>leak</u>

```
unsigned int cap;
unsigned int len;
int * arr;

/* ... */

if (len >= cap) {
   newcap = 2*cap;
   newarr = malloc (newcap*sizeof(int));
   memcpy (newarr, arr, len*sizeof(int));
   free (arr);
   arr = newarr;
   cap = newcap;
}
arr[len] = val;
++len;
```

vector contents:

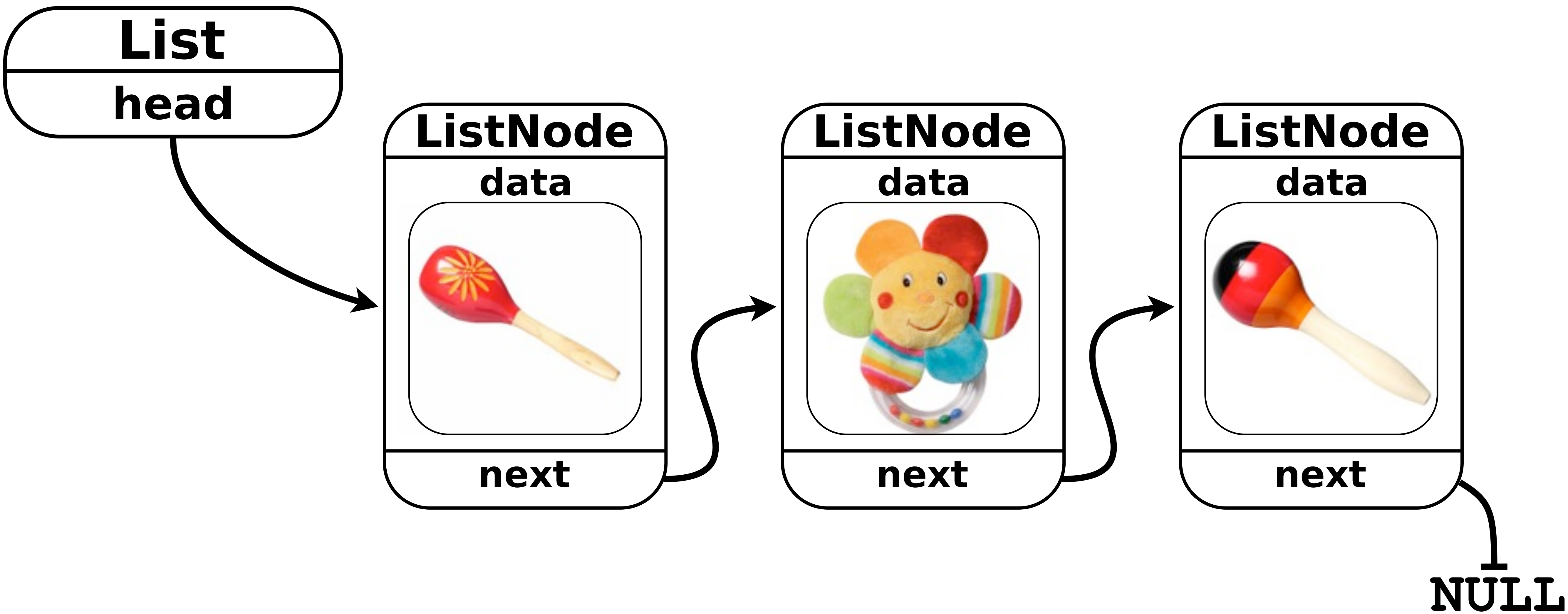
index	0	1	2	3	4		
value	42	77	-123	321	-21		

Implementation Overview
Linked Lists

- wrap user data into list nodes *("items")*
  - links between nodes
  - links to first (and last) node
  - *links are implemented with pointers*
- look at three examples:
  - simply linked list
  - doubly linked list
  - circular list

Implementation Overview

Linked Lists



Implementation Overview

Linked Lists

- advantages:
  - simple
  - lightweight
  - “unlimited” capacity
- potential drawback:
  - no random access

Intermezzo

Important Elements of C

- **structs**
  - encapsulate information required to manage a data structure
  - we'll frequently <u>store pointers</u>
- **functions**
  - encapsulate the computations required for operations on a data structure
  - *introduced last week, practice this week*
  - we'll frequently <u>pass pointers</u>

Structs Define New Types

```
struct name_s {
   char *first, *last;
};

struct date_s {
   int year, day;
   char *month;
};

struct person {
   struct name_s name;
   struct date_s birthday;
};
```

structs contains **fields**, each field has a **type** and a **name**

structs can be **nested**

Structs Storage Examples

```
struct name_s {
   char *first, *last;
};

struct date_s {
   int year, day;
   char *month;
};

struct person_s {
   struct name_s name;
   struct date_s birthday;
};
```

	address	value
first	4108	44b4
last	410c	4800

address	value
44b4	'B'
44b5	'o'
44b6	'b'
44b7	0

address	value
4800	'D'
4801	'o'
4802	'e'
4803	0

Structs Storage Examples

```
struct name_s {
  char *first, *last;
};

struct date_s {
  int year, day;
  char *month;
};

struct person_s {
  struct name_s name;
  struct date_s birthday;
};
```

		address	value
name	*first*	7290	44b4
	last	7294	4800
birthday	*year*	7298	1999
	day	729c	17
	month	72a0	49a4

Field Access and Pointers

```
struct person_s bob;
bob.name.first = "Bob";
bob.birthday.year = 1999;
/* etc */

struct person_s *alice;
alice = malloc (sizeof(*alice));
alice->name.first = "Alice";
/* etc */

struct person_s *p1, *p2;
p1 = &bob;
p2 = alice;
```

Field Access and Pointers

```
struct person_s bob;
bob.name.first = “Bob”;
bob.birthday.year = 1999;
/* etc
```

dot notation for value-variables

```
struct
alice
```

arrow notation for pointer-variables

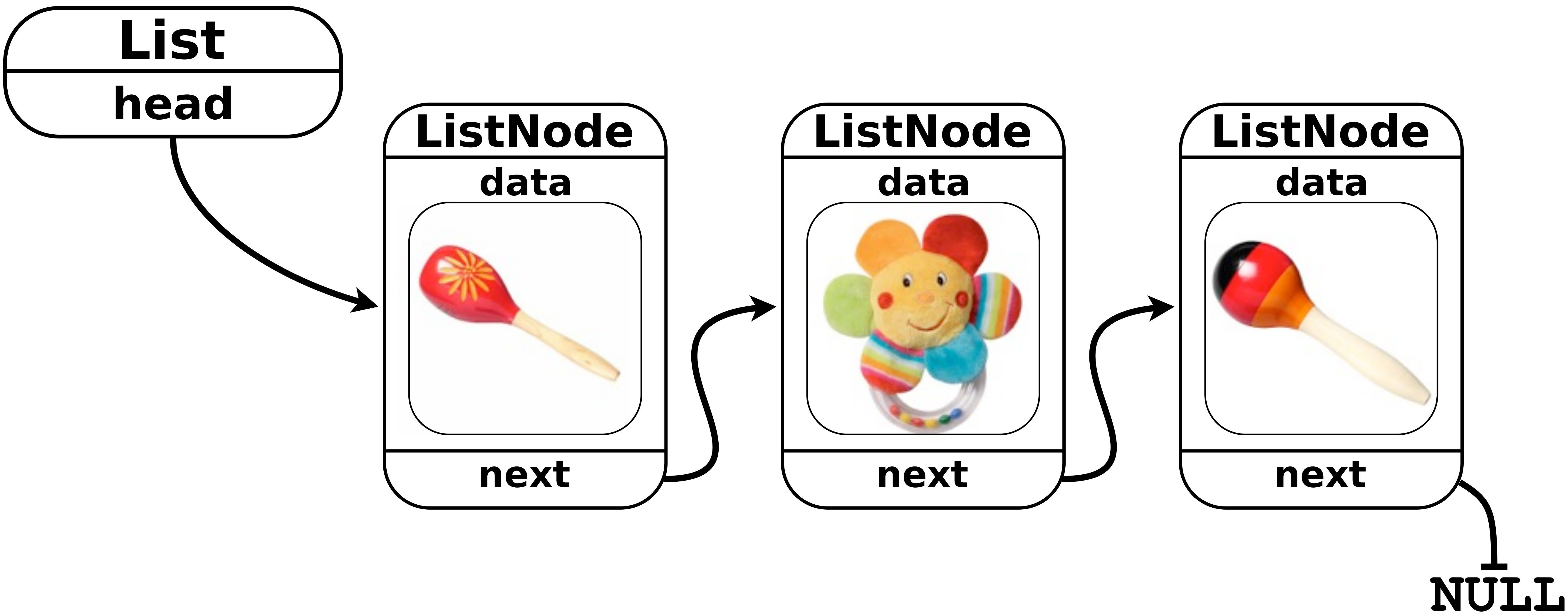
```
alice->name.first = “Alice”;
/* etc
```

Note: `alice->name` is a **value** here

```
struct person_s *p1, *p2;
p1 = &bob;
p2 = alice;
```

...back to

Linked Lists



Implementation Sketch

Lists

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;

/* ... */

struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;

/* ... */

struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

variables

	address	value
head	0124	NULL
tail	0128	NULL
it	012c	NULL

list contents: ↘ *null*

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;
```

variables

	address	value
head	0124	NULL
tail	0128	NULL
it	012c	**21c8**

let's append 42

```
struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

	address	value
value	21c8	
next	21cc	

list contents: ↘ *null*

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;
```

let's append 42

```
struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

true

variables

	address	value
head	0124	NULL
tail	0128	NULL
it	012c	21c8

	address	value
value	21c8	**42**
next	21cc	**NULL**

list contents: *null*

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;
```

let's append 42

```
struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

variables

	address	value
head	0124	**21c8**
tail	0128	**21c8**
it	012c	21c8

	address	value
value	21c8	42
next	21cc	NULL

list contents: → 42 → *null*

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;
```

variables

	address	value
head	0124	21c8
tail	0128	21c8
it	012c	21c8

let's append 17

```
struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

	address	value
value	21c8	42
next	21cc	NULL

list contents: → 42 → *null*

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;
```

let's append 17

```
struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

false

variables

	address	value
head	0124	21c8
tail	0128	21c8
it	012c	**4f10**

	address	value
value	21c8	42
next	21cc	NULL

	address	value
value	4f10	**17**
next	4f14	**NULL**

list contents: → 42 → *null*

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;
```

let's append 17

```
struct item * it;
it = malloc (sizeof (*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

variables

	address	value
head	0124	21c8
tail	0128	21c8
it	012c	4f10

	address	value
value	21c8	42
next	21cc	NULL

	address	value
value	4f10	17
next	4f14	NULL

list contents: → 42 → *null*

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;
```

variables

	address	value
head	0124	21c8
tail	0128	21c8
it	012c	4f10

let's append 17

```
struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

	address	value
value	21c8	42
next	21cc	NULL

	address	value
value	4f10	17
next	4f14	NULL

list contents: → 42 → *null*

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;
```

let's append 17

```
struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

variables

	address	value
head	0124	21c8
tail	0128	21c8
it	012c	4f10

	address	value
value	21c8	42
next	21cc	NULL

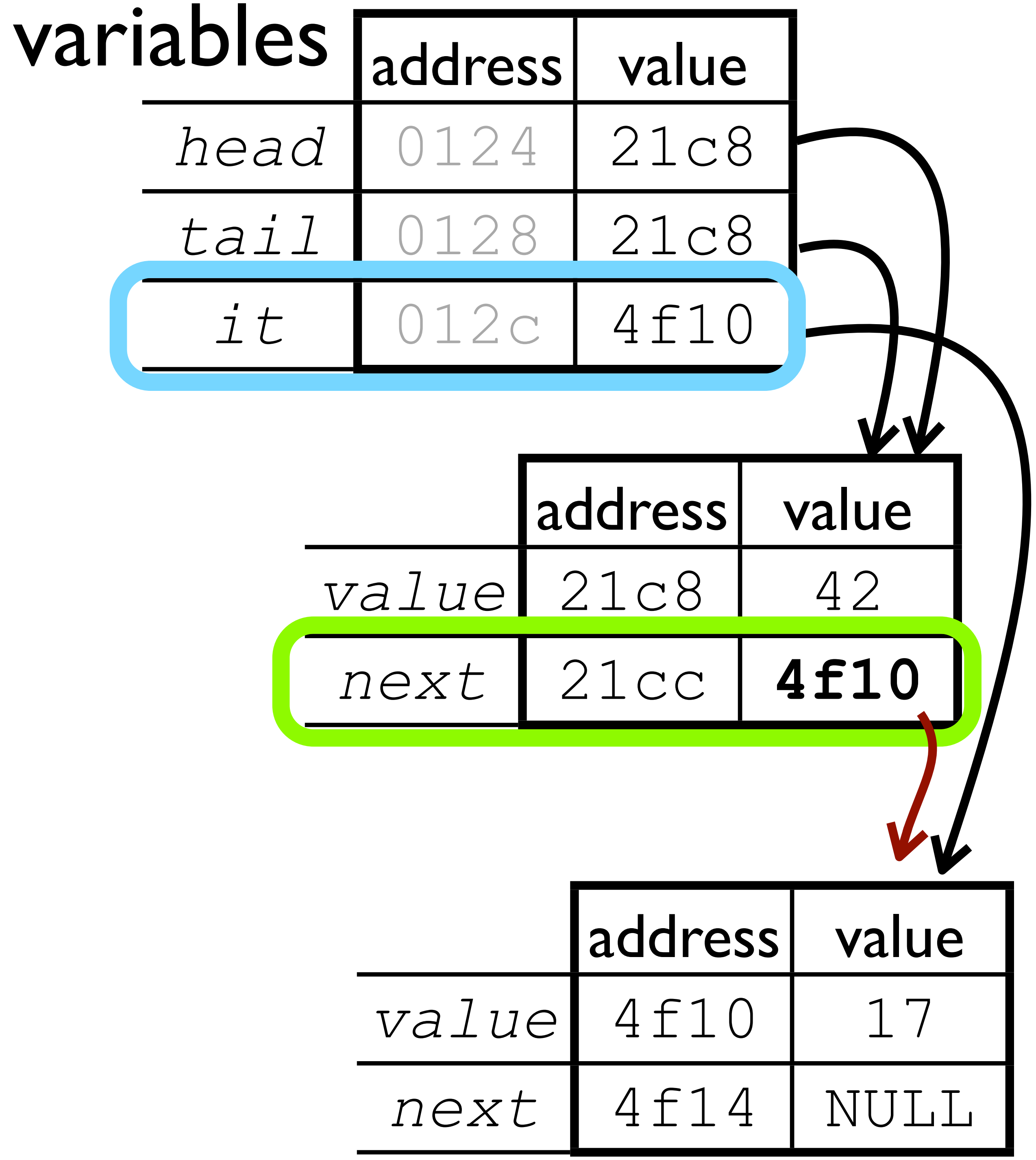
	address	value
value	4f10	17
next	4f14	NULL

list contents: → 42 → *null*

```c
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;
```

let's append 17

```c
struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

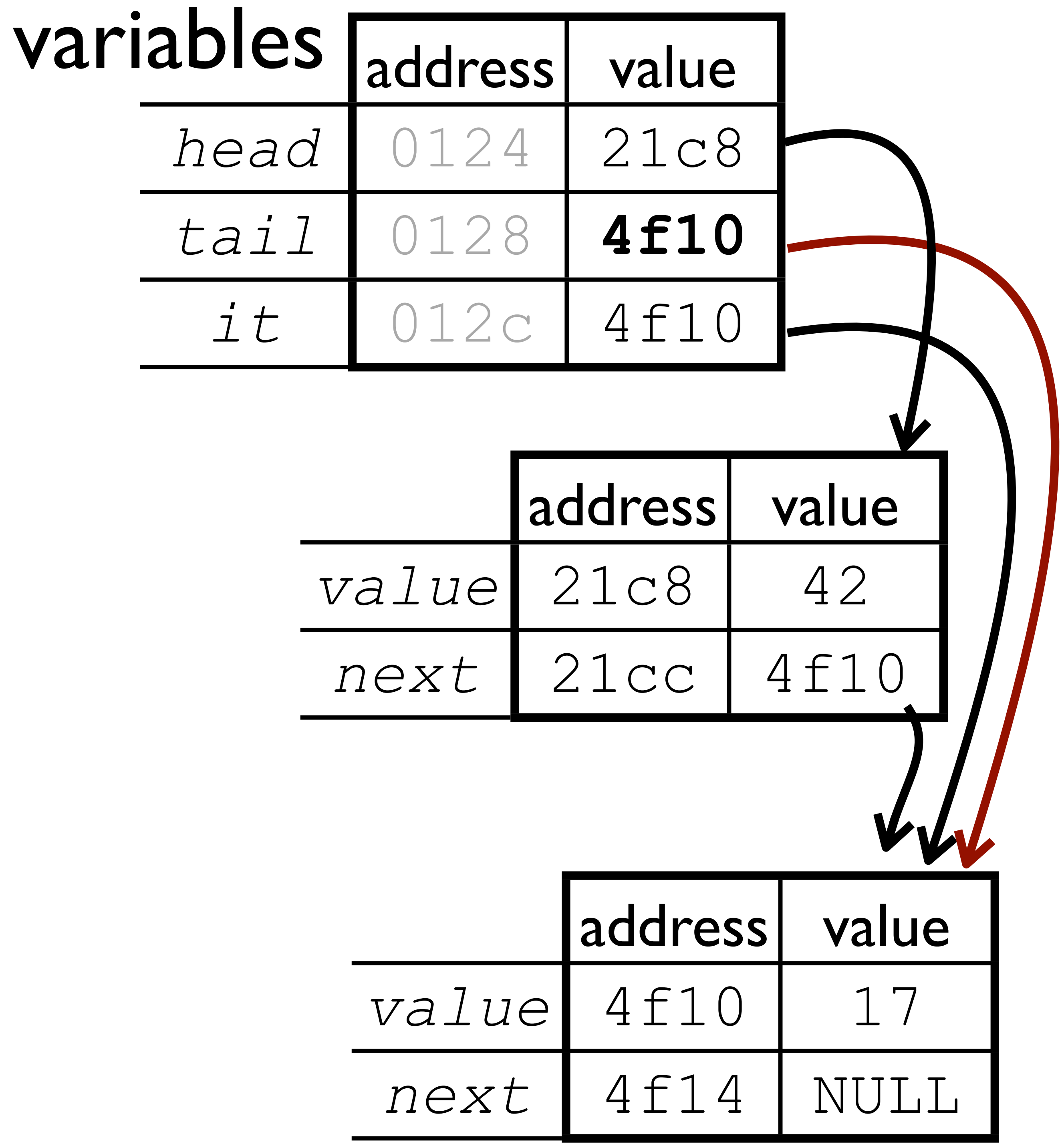


list contents: → 42 → 17 → *null*

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;
```

let's append 17

```
struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

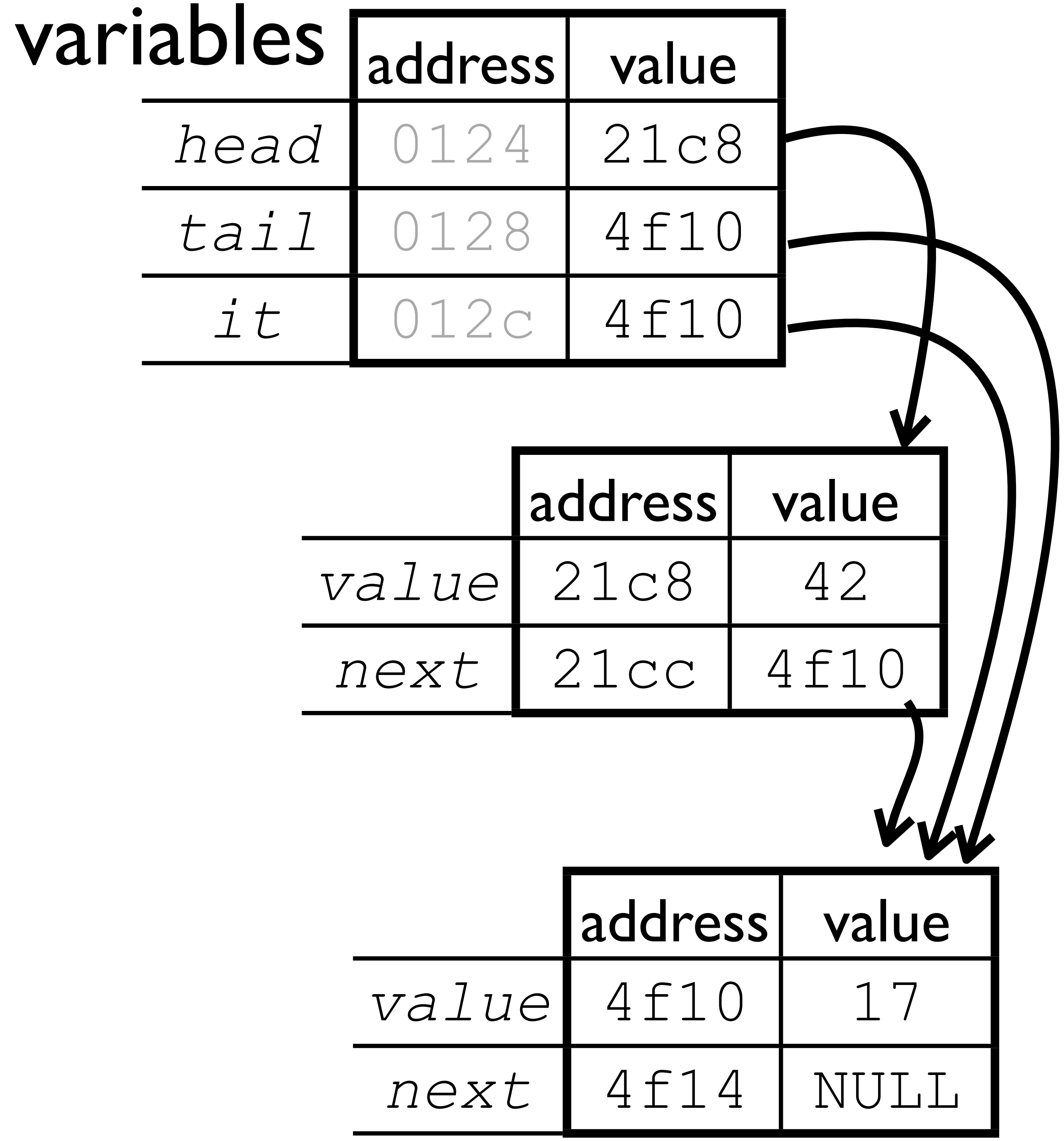


list contents: → 42 → 17 → *null*

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;

/* ... */

struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```



list contents: → 42 → 17 → *null*

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;

/* ... */

struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

sooner or later we must `free` the memory allocated here

Implementation Sketch

Lists

```
struct item {
   int value;
   struct item * next;
};
struct item *head, *tail;

/* ... */

struct item * it;
it = malloc (sizeof(*it));
it->value = value;
it->next = NULL;
if (NULL == head) {
   head = it;
   tail = it;
}
else {
   tail->next = it;
   tail = it;
}
```

A Typical List Operation **Function**

```
int list_append (struct list_s * list, int val)
{
   struct item * it;
   if (NULL == (it = malloc (sizeof(*it))))
      return -1;
   it->value = value;
   it->next = NULL;
   if (NULL == list->head) {
      list->head = it;
      list->tail = it;
   }
   else {
      list->tail->next = it;
      list->tail = it;
   }
   return 0;
}
```

A Typical List Operation Function

```
int list_append (struct list_s * list, int val)
{
   struct i
   if (NULL == (it = malloc (sizeof(*it))))
      return -1;
   it->valu
   it->nex
   if (NULL == list->head) {
      list->head = it;
      list->tail = it;
   }
   else {
      list->tail->next = it;
      list->tail = it;
   }
   return 0;
}
```

pointer to the list that should get modified

return value indicates success / failure

Typical List **Declarations**

```
typedef struct item_s {
  int value;
  struct item_s *next;
} Item;

typedef struct list_s {
  Item *head, *tail;
} List;

void list_init (List *ll);
void list_destroy (List *ll);
int list_append (List *ll, int val);
int list_insert (List *ll, Item *pos, int val);
int list_remove (List *ll, Item *it);

/* etc... */
```

Typical List **Declarations**

```
typedef struct item_s {
   int value;
   struct item_s *next;
} Item;

typedef struct list_s {
   Item *head, *tail;
} List;

void list_init (List *ll);
void list_destroy (List *ll);
int list_append (List *ll, int val);
int list_insert (List *ll, Item *pos, int val);
int list_remove (List *ll, Item *it);

/* etc... */
```

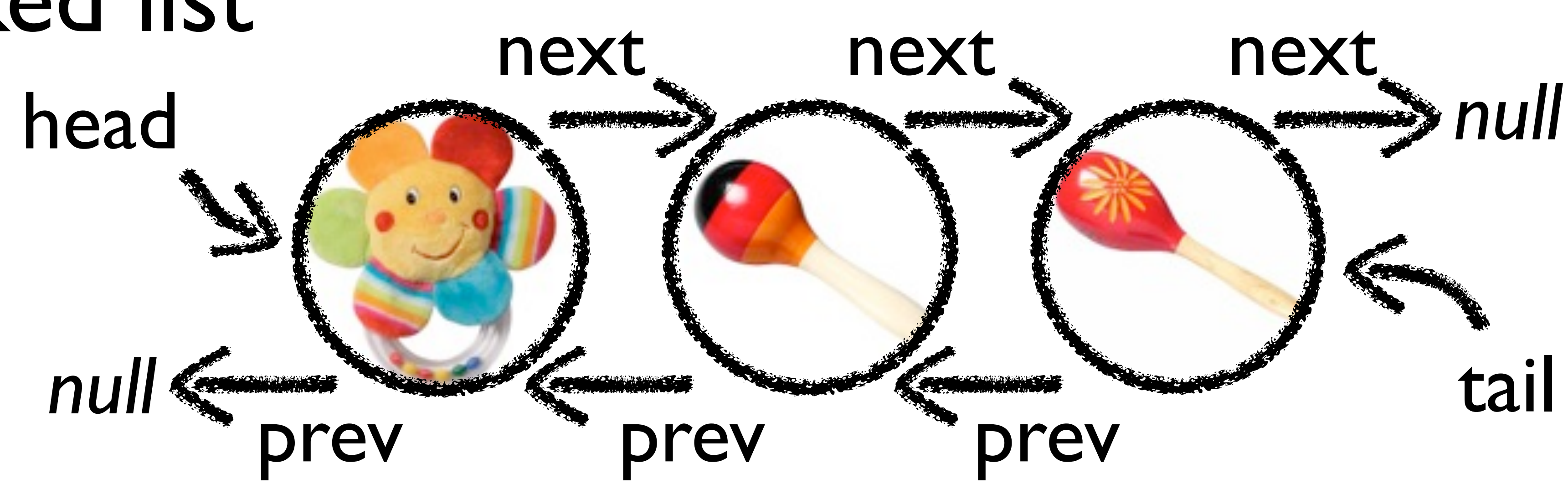
these have to `free` memory allocated elsewhere

Common List Variations

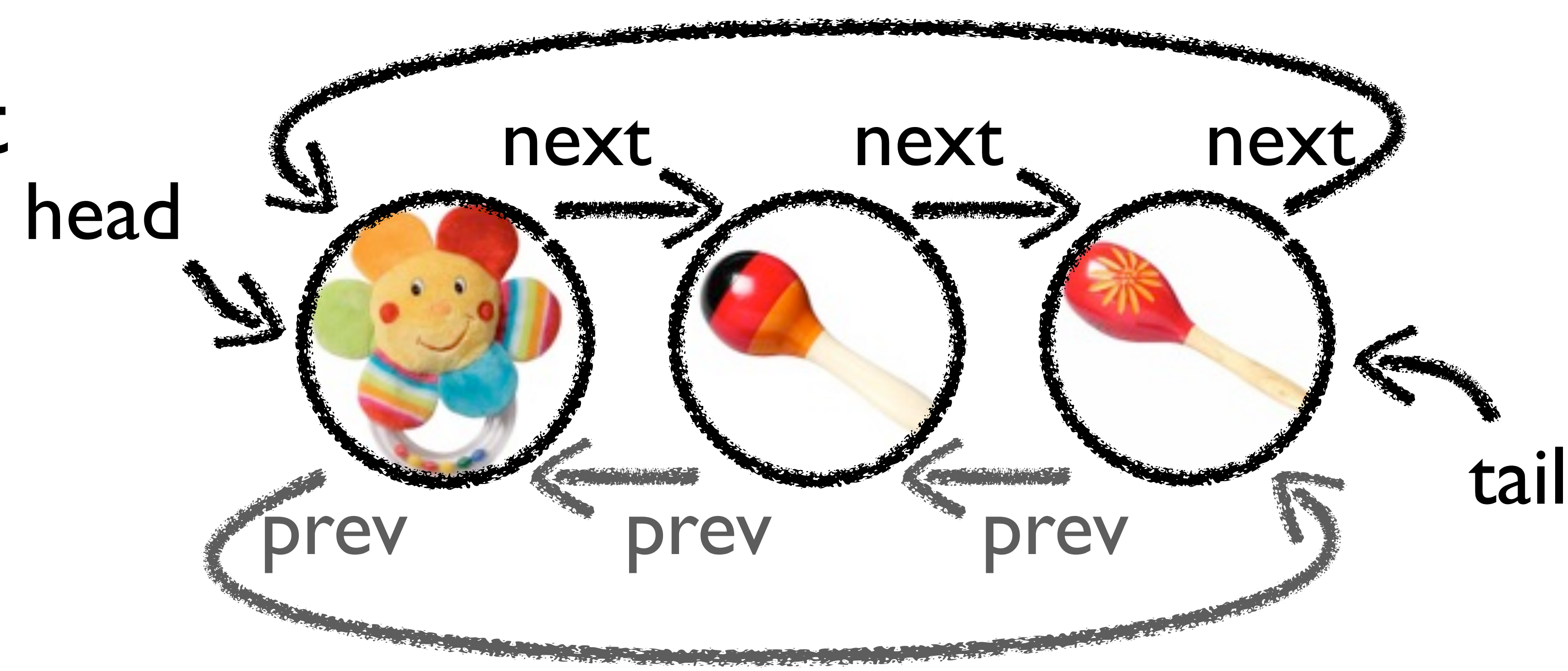
- doubly-linked list
    - each item also knows its predecessor
- circular list
    - the tail's successor is the first node
    - the head's predecessor is the last node
      (for doubly-linked circular lists)

Common List Variations

- doubly-linked list



- circular list



Common List Specializations

- <u>stack</u> (LIFO - last in first out):
  - push, pop, top
- <u>queue</u> (FIFO - first in first out):
  - enqueue *(insert at the end)*
  - dequeue *(remove from the front)*
  - get *(retrieve from the front)*
- <u>deque</u> *(double-ended queue):*
  - enqueue at front or end
  - dequeue at front or end
  - get at front or end

Iteration

- visit each item of a container
- vectors:
  - just use array index
  - can go forward, backward
  - can even "jump around" *(random access)*
- lists:
  - just use list nodes
  - can go forward
  - doubly-linked nodes can also go backward
  - *no random access*

Vector Iteration

```
struct vector_s {
  int *arr;
  unsigned long cap;
  unsigned long len;
};
struct vector_s *vec;

/* ... */
unsigned long i;
for (i = 0; i < vec->len; ++i)
  printf ("%d\n", vec->arr[i]);
```

List Iteration

```
struct item_s {
  int value;
  struct item_s * next;
};
struct list_s {
  struct item_s * head;
};
struct list_s * list;

/* ... */
struct item_s * it;
for (it = list->head; NULL != it; it = it->next)
  printf (“%d\n”, it->value);
```

...almost done...

Some Lose Ends

- store something other than integers
  - copy-paste-adapt *(easy but tedious)*
  - generics *(reusable but trickier)*
    - void pointers
    - pointer casts

- common list implementation glitches
  - order of link reassignments
  - list insertion after tail or before head

Changing the Item Type

```
struct int_vector_s {
  int *arr;
  size_t len, cap;
};

int int_vector_append (
        struct int_vector_s *vec,
        int val);
```

Changing the Item Type

```
struct complex_s {
  double real, imag;
};

struct cpx_vector_s {
  struct complex_s *arr;
  size_t len, cap;
};

int cpx_vector_append (
        struct cpx_vector_s *vec,
        struct complex_s val);
```

Changing the Item Type

```
struct complex_s {
  double real, imag;
};

struct complex_vector_s {
  struct complex_s *arr;
  size_t len, cap;
};

int complex_vector_append (
        struct complex_vector_s *vec,
        struct complex_s val);
```

this is trivial only for item types that can be bitwise copied by the compiler

Changing the Item Type

```
struct person_vector_s {
  struct person_s **arr;
  size_t len, cap;
};
```

storing pointers to items is more generic than storing their values

```
int person_vector_append (
        struct person_vector_s *vec,
        struct person_s *val);
```

but now ownership management becomes really important *(more details later)*

The Same for Lists

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

```
struct person_item_s {
  struct person_s *value;
  struct person_item_s *next;
};

struct person_list_s {
  struct person_item_s *head, *tail;
};

int person_list_append (
         struct person_list_s * list,
         struct person_s *value);
```

Generic Item Type

```
struct item_s {
  void *value;
  struct item_s *next;
};
struct list_s {
  struct item_s *head, *tail;
};

int list_append (
         struct list_s * list,
         void *value);

struct list_s * list;
struct person_s * bob;
/* ... */
list_append (list, bob);
```

Generic Item Type

```
struct item_s {
  void *value;
  struct item_s *next;
};
struct list_s {
  struct item_s *head, *tail;
};

int list_append (
         struct list_s * list,
         void *value);

struct list_s * list;
struct person_s * bob;
/* ... */
list_append (&list, &bob);
```

to retrieve values, we need to cast from void* to the correct type

and more importantly, the memory occupied by **`bob`** needs to remain valid until **`list`** is destroyed!

Take-Home Message

- containers serve to store, look up, remove, and iterate over data items
- sequence containers store a serial arrangement of data items
- vectors and linked lists are the fundamental sequence container types
- stacks (LIFO), queues (FIFOs), and other variations are easily built on top of lists