

Linked data structures

Johan Montelius

KTH

VT26

a class/record/object/struct

A data structure with a fixed set of (named) properties.
Properties could be of different types.

```
struct person {  
    char *name;  
    char *adress;  
    int age;  
  
    :  
    :  
}
```

a class/record/object/struct

A data structure with a fixed set of (named) properties.
Properties could be of different types.

```
struct person {  
    char *name;  
    char *adress;  
    int  age;  
  
    :  
    :  
}
```

a class/record/object/struct

A data structure with a fixed set of (named) properties.
Properties could be of different types.

```
struct person {  
    char *name;  
    char *adress;  
    int age;  
  
    :  
    :  
}
```

a class/record/object/struct

Structures can be created and their properties used.

```
struct person anders = {"Anders", "Storgatan 12", 27, ...}  
:  
printf("Hej  \n", anders.name);  
:
```

Nothing new, you all know this.

a class/record/object/struct

Structures can be created and their properties used.

```
struct person anders = {"Anders", "Storgatan 12", 27, ...}  
:  
printf("Hej  \\\n", anders.name);  
:
```

Nothing new, you all know this.

let's play some cards

```
enum Suit {  
    HEART,  
    SPADE,  
    DIAMOND,  
    CLUB  
};
```

let's play some cards

```
struct card {  
    enum Suite suite  
    int value;  
}
```

```
struct card *create_card(enum Suit suit, int val) {  
    struct card *new = malloc(sizeof(struct card));  
    new->suit = suit;  
    new->value = val;  
    return new;  
}
```

a deck of cards

```
#define SIZE 8

struct deck {
    struct card **cards;
    int size;
    int last;
}

struct deck *create_deck() {
    struct deck *new = malloc(sizeof(struct deck));
    deck->cards = malloc(sizeof(struct card)*SIZE);
    deck->size = SIZE;
    deck->last = 0;
}
```

We've seen this before.

a deck of cards

```
#define SIZE 8

struct deck {
    struct card **cards;
    int size;
    int last;
}

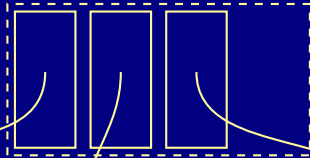
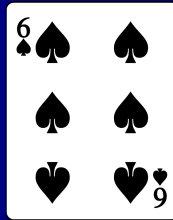
struct deck *create_deck() {
    struct deck *new = malloc(sizeof(struct deck));
    deck->cards = malloc(sizeof(struct card)*SIZE);
    deck->size = SIZE;
    deck->last = 0;
}
```

We've seen this before.

a deck of cards

.cards

.last 3



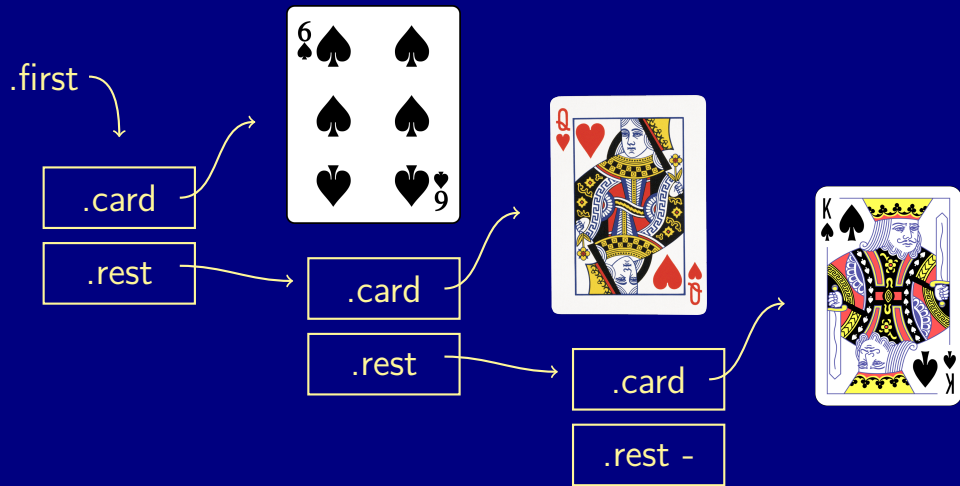
how about this

```
struct cell {  
    struct card *card;  
    struct cell *rest;  
}
```

```
struct deck {  
    cell *first;  
}
```

```
struct deck *create_deck() {  
    struct deck *new = malloc(sizeof(struct deck));  
    deck->first = NULL;  
    return deck;  
}
```

how about this



pros and cons

Access the n 'th card.

- The list of cards has an $O(n)$ access operation.
- The array of cards has an $O(1)$ access operation.

pros and cons

Access the n 'th card.

- The list of cards has an $O(n)$ access operation.
- The array of cards has an $O(1)$ access operation.

pros and cons

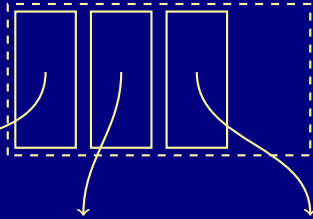
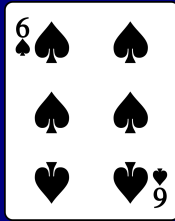
Access the n 'th card.

- The list of cards has an $O(n)$ access operation.
- The array of cards has an $O(1)$ access operation.

adding a card to an array of cards

.cards

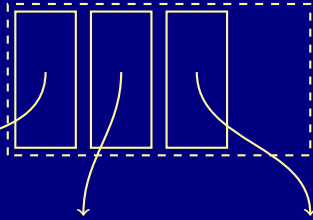
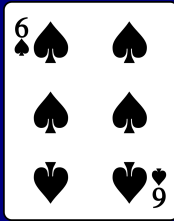
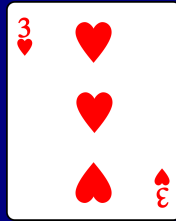
.first 3



adding a card to an array of cards

.cards

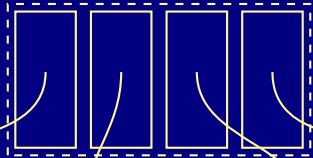
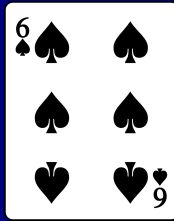
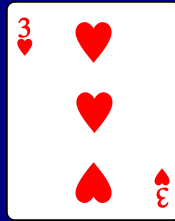
.first 3



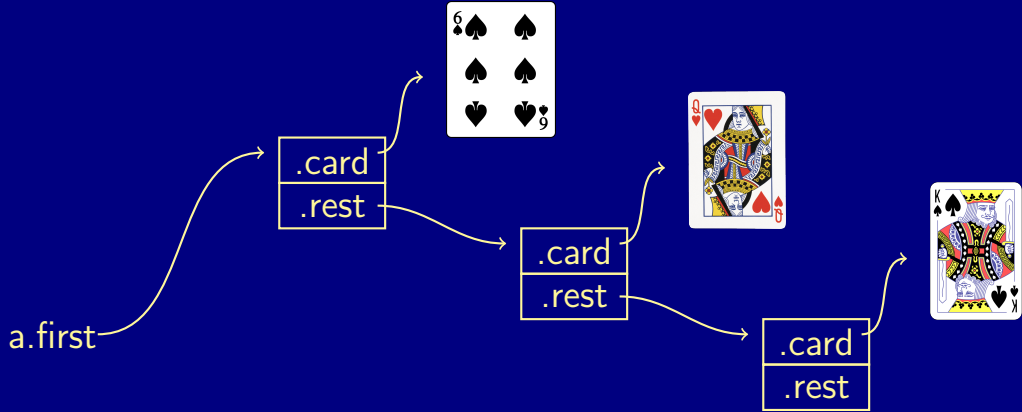
adding a card to an array of cards

.cards

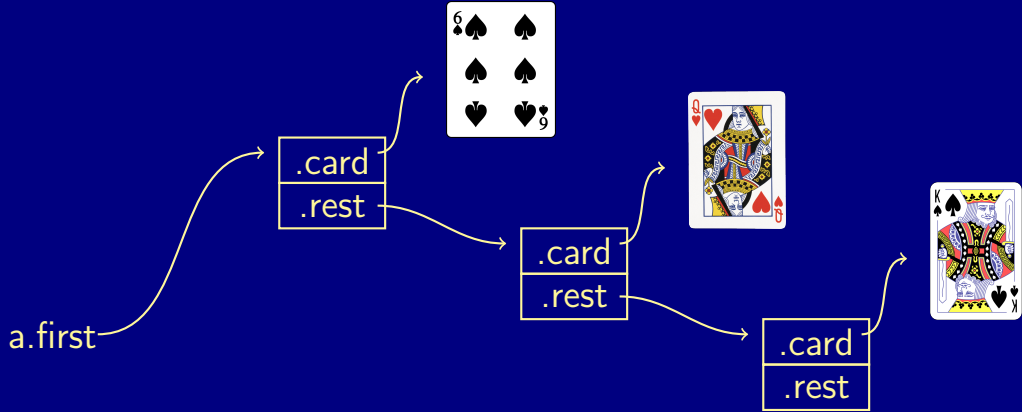
.first 4



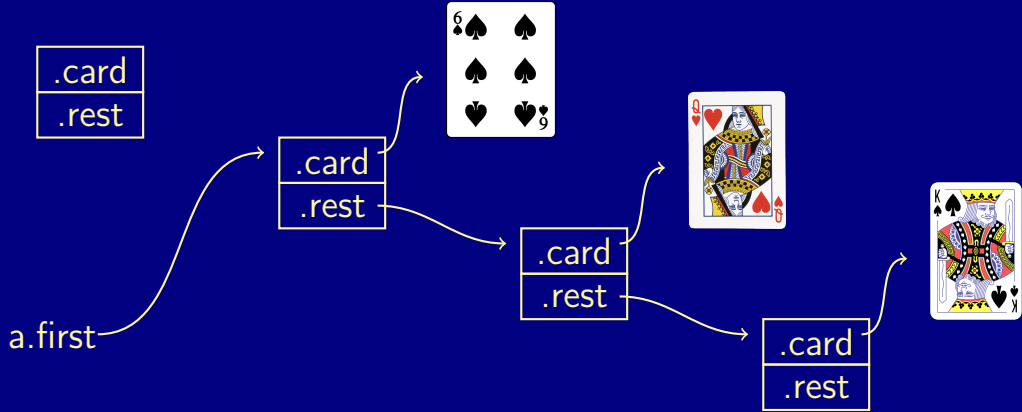
adding a card to a list of cards



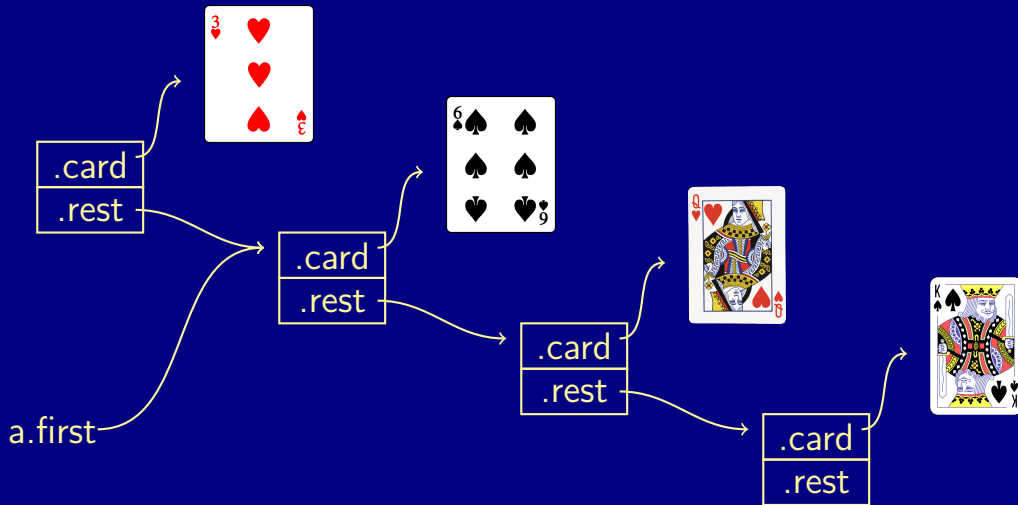
adding a card to a list of cards



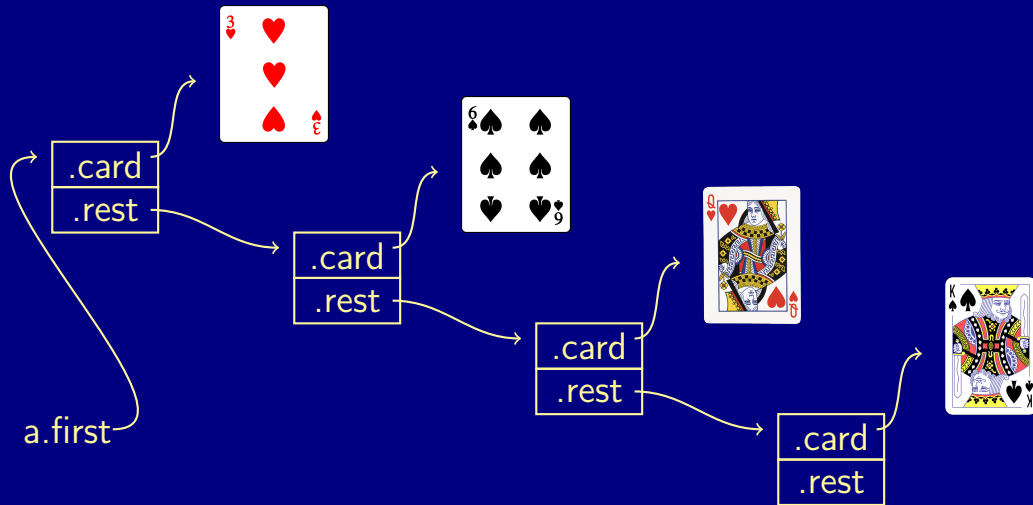
adding a card to a list of cards



adding a card to a list of cards



adding a card to a list of cards



pros and cons

pros and cons

Adding a card has a time complexity of ...

pros and cons

Adding a card has a time complexity of ...

- a list of cards: $O(1)$

pros and cons

Adding a card has a time complexity of ...

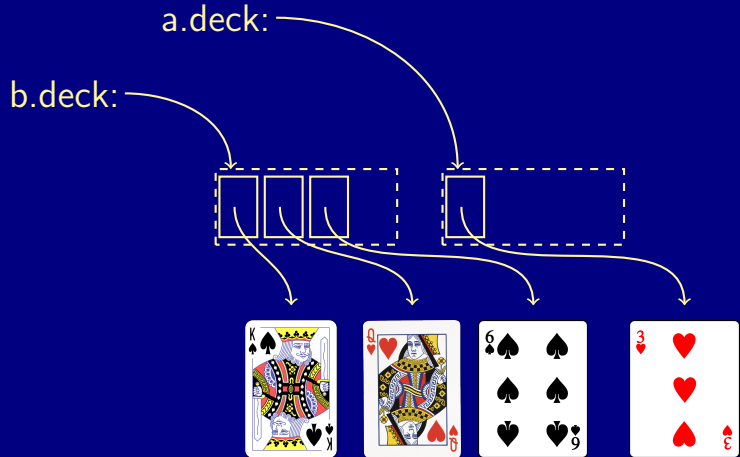
- a list of cards: $O(1)$
- a *dynamic array*: amortized cost of $O(1)$

append one deck to another

append one deck to another

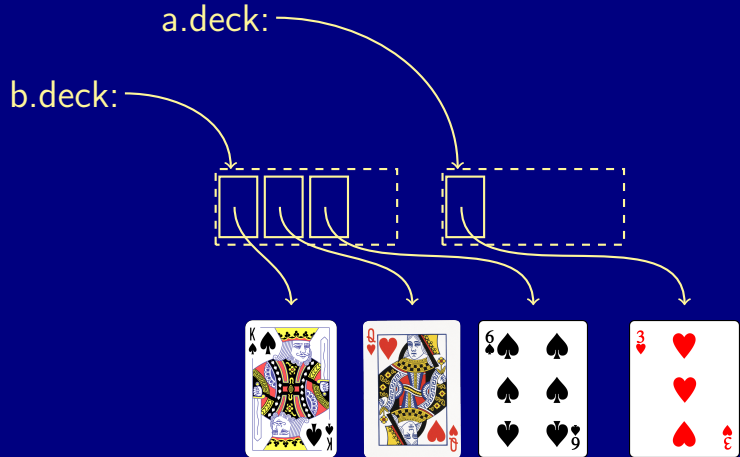
Assume we have two decks of cards, a and b, how do we *append* b to a i.e. the deck a will after the operation hold all cards and b should be empty.

append an array of cards



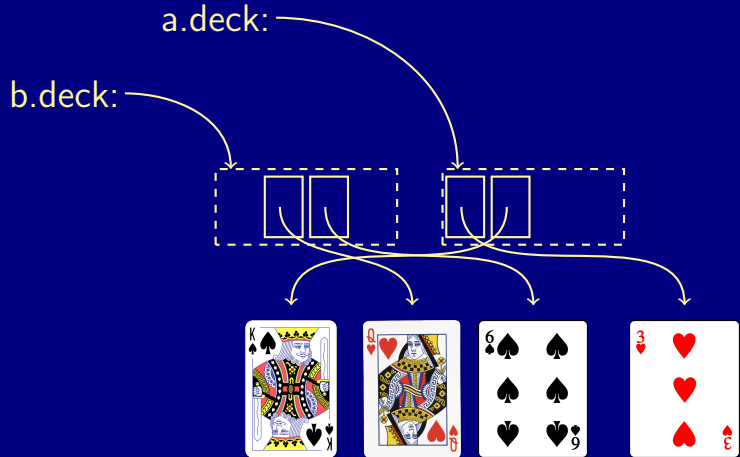
```
append(a, b);
```

append an array of cards



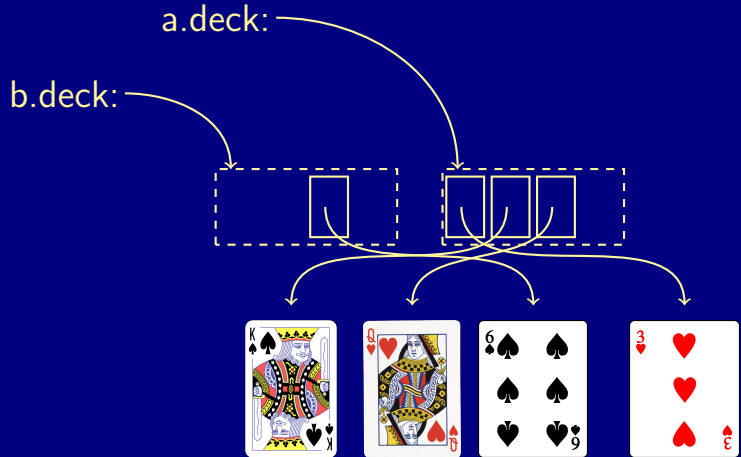
`append(a, b);`

append an array of cards



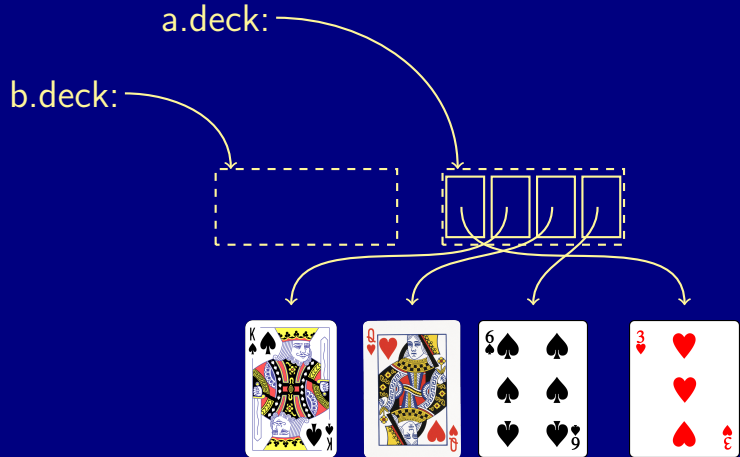
```
append(a, b);
```

append an array of cards



```
append(a, b);
```

append an array of cards

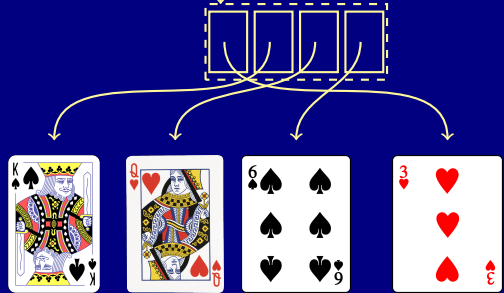


```
append(a, b);
```

append an array of cards

a.deck:

b.deck: null

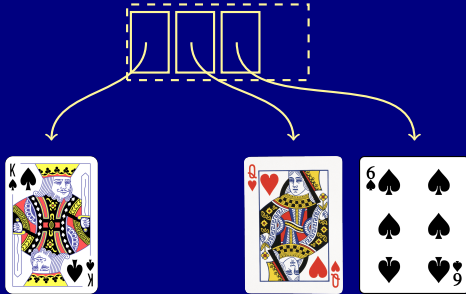


`append(a, b);`

append a list of cards

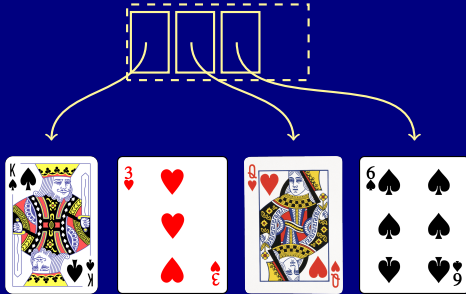
insert a card in an array

a.deck:



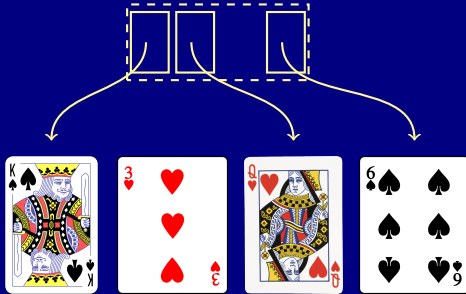
insert a card in an array

a.deck:



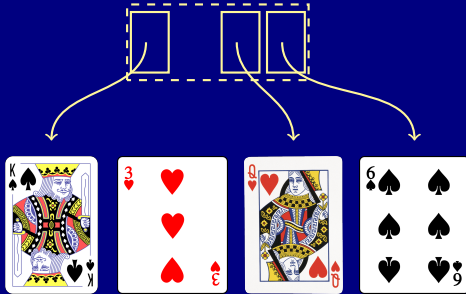
insert a card in an array

a.deck:



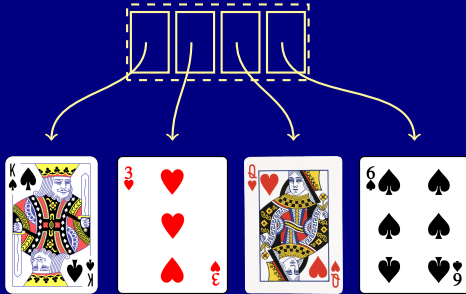
insert a card in an array

a.deck:

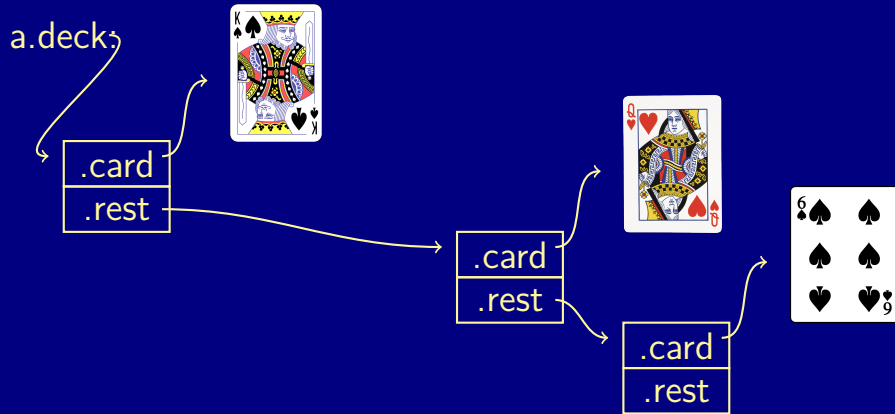


insert a card in an array

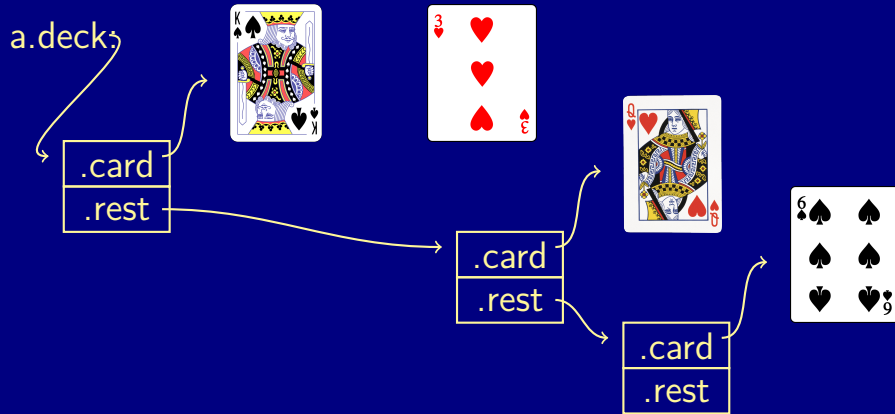
a.deck:



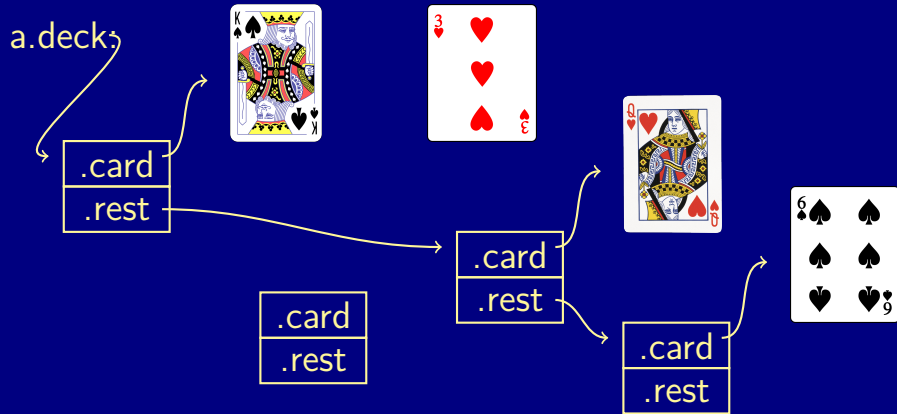
insert a card in a list



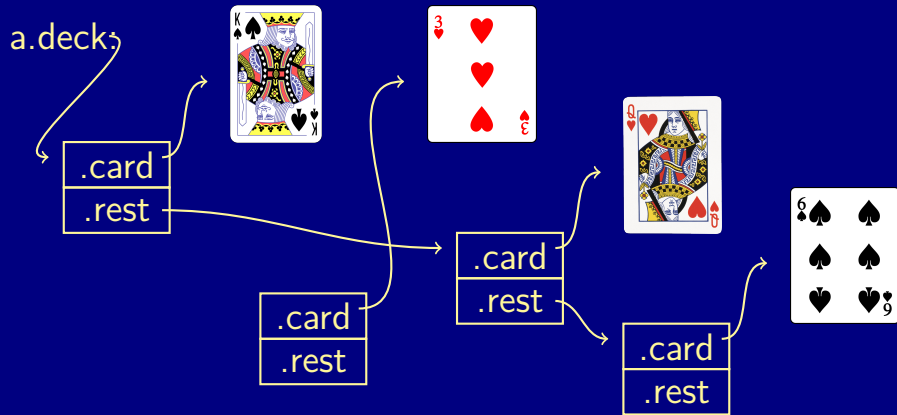
insert a card in a list



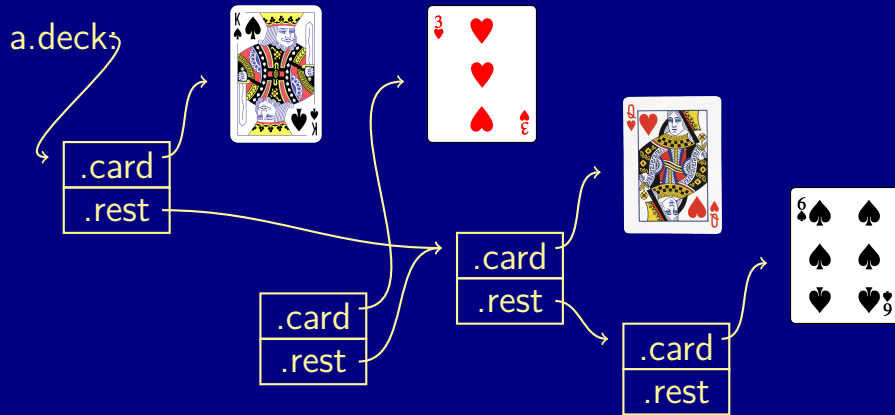
insert a card in a list



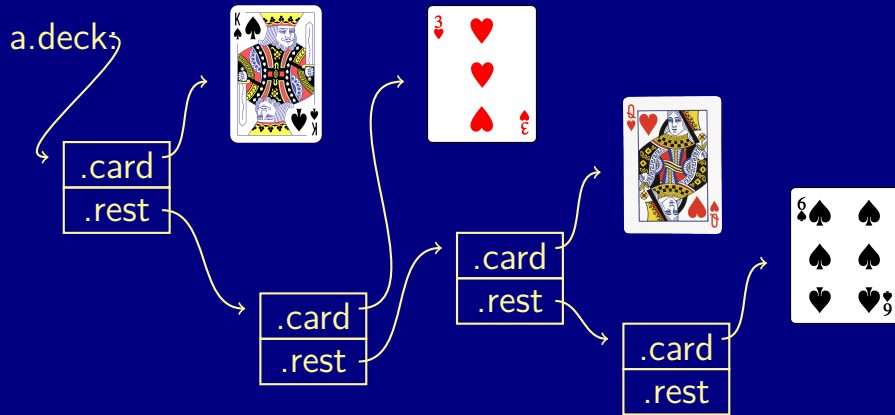
insert a card in a list



insert a card in a list



insert a card in a list



pros and cons

pros and cons

Inserting a card in an ordered sequence.

pros and cons

Inserting a card in an ordered sequence.

- The list of cards has an $O(n)$ insert operation....

pros and cons

Inserting a card in an ordered sequence.

- The list of cards has an $O(n)$ insert operation....
- ..., only $O(n)$ read operations and $O(1)$ write operations.
- The array of cards has an $O(n)$ insert operations ...

pros and cons

Inserting a card in an ordered sequence.

- The list of cards has an $O(n)$ insert operation....
- ..., only $O(n)$ read operations and $O(1)$ write operations.
- The array of cards has an $O(n)$ insert operations ...
- ..., $O(n)$ read and write operations.

a linked list

```
struct cell {  
    int head;  
    struct cell *tail;  
}
```

```
struct linked_list {  
    struct cell* first;  
}
```

a linked list

```
struct cell {  
    int head;  
    struct cell *tail;  
}
```

```
struct linked_list {  
    struct cell* first;  
}
```

The cell data structure is also referred to as a *cons cell*.

a linked list - search

```
bool search(struct linked_list *lst, int key) {  
    struct cell *nxt = lst->first;  
    while (nxt != NULL) {  
        if (nxt->head == key)  
            return true;  
        nxt = nxt->tail;  
    }  
    return false;  
}
```

a linked list - append

```
void append(linked_list *a, linked_list *b) {  
    struct cell *nxt = a->first;  
    while (nxt->tail != NULL) {  
        nxt = nxt->tail;  
    }  
    nxt->tail = b->first;  
    b->first = NULL;  
}
```

a linked list - append

```
void append(linked_list *a, linked_list *b) {  
    struct cell *nxt = a->first;  
    while (nxt->tail != NULL) {  
        nxt = nxt->tail;  
    }  
    nxt->tail = b->first;  
    b->first = NULL;  
}
```

There is an error in this code - find it.

linked list - what?

```
void what(struct linked_list *lst, int key) {  
    struct cell *nxt = lst->first;  
    struct cell *prv = NULL;  
    while (nxt != NULL) {  
        if (nxt->head == key) {  
            if (prv != NULL) {  
                prv->tail = nxt->tail;  
            } else {  
                lst->first = nxt->tail;  
            }  
            free(nxt);  
            break;  
        }  
        prv = nxt;  
        nxt = nxt->tail;  
    }  
}
```

a stack

```
struct stack {  
    struct cell *top;  
}
```

```
struct stack *create_stack() [  
    struct stack *new = malloc(sizeof(struct stack));  
    new->top = NULL;  
    return new;  
}
```

Stack - push n pop

```
void push(struct stack *stk, int item) {  
    struct cell *new = malloc(sizeof(struct cell));  
    new->head = item;  
    new->tail = stk->top;  
    stk->top = new;  
}
```

Stack - push n pop

```
void push(struct stack *stk, int item) {  
    struct cell *new = malloc(sizeof(struct cell));  
    new->head = item;  
    new->tail = stk->top;  
    stk->top = new;  
}
```

```
int pop(struct stack *stk) {  
    if (stk->top == NULL) {    exit(1);    }  
    struct cell *top = stk->top;  
    int ret =  top->head;  
    stk->top = top->tail;  
    free(top);  
    return ret;  
}
```

linked lists

linked lists

- $O(n)$ to find the right position

linked lists

- $O(n)$ to find the right position
- $O(1)$ to perform operation once position is found

linked lists

- $O(n)$ to find the right position
- $O(1)$ to perform operation once position is found
- often simple to work with

linked lists

- $O(n)$ to find the right position
- $O(1)$ to perform operation once position is found
- often simple to work with
- ideal for a dynamic stack