CENG 3549 – Functional Programming

Formal Verification of Functional Programs with Coq

Burak Ekici

December 15, 2022

Outline

Definition (lists)

```
Inductive list (A : Type) : Type ≜
    | nil : list A
    | cons : A → list A → list A.

(** induction principle *)
list_ind =
fun (A : Type) (P : list A → Prop) (f : P [])
  (f0 : ∀ (a : A) (l : list A), P l → P (a :: l)) ⇒
fix F (l : list A) : P l ≜
  match l as l0 return (P l0) with
  | [] ⇒ f
  | y :: l0 ⇒ f0 y l0 (F l0)
  end
     : ∀ (A : Type) (P : list A → Prop),
       P [] → (∀ (a : A) (l : list A), P l → P (a :: l)) → ∀ l : list A, P l
```

Definition (list append)

```
Fixpoint append {A: Type} (l1 l2: list A): list A ≜
  match l1 with
    | []    ⇒ l2
    | x::xs ⇒ x::append xs l2
  end.
```

Definition (list append)

```
Fixpoint append {A: Type} (l1 l2: list A): list A ≜
  match l1 with
    | []    ⇒ l2
    | x::xs ⇒ x::append xs l2
  end.
```

Lemma

nil is right identity of append, that is

```
Lemma app_nil: ∀ {A: Type} (l: list A), append l [] = l.
```

Definition (list append)

```
Fixpoint append {A: Type} (l1 l2: list A): list A ≜
  match l1 with
    | []    ⇒ l2
    | x::xs ⇒ x::append xs l2
  end.
```

Lemma

nil is right identity of append, that is

```
Lemma app_nil: ∀ {A: Type} (l: list A), append l [] = l.
```

Proof.

```
Lemma app_nil: ∀ {A: Type} (l: list A), append l [] = l.
Proof. intros A l.
       induction l; intros.
       - simpl. easy.
       - simpl. rewrite IHl. easy.
Qed.
```

Lemma

append is associative, that is

```
Lemma app_assoc: ∀ {A: Type} (l1 l2 l3: list A), append (append l1 l2) l3 = append l1 (append l2 l3).
```

Lemma

append is associative, that is

```
Lemma app_assoc: ∀ {A: Type} (l1 l2 l3: list A), append (append l1 l2) l3 = append l1 (append l2 l3).
```

Proof.

```
Lemma app_assoc: ∀ {A: Type} (l1 l2 l3: list A),
  append (append l1 l2) l3 = append l1 (append l2 l3).
Proof. intros A l1.
       induction l1; intros.
       - simpl. easy.
       - simpl. rewrite IHl1. easy.
Qed.
```

Definition (list length)

```
Fixpoint length {A: Type} (l: list A): nat ≜
  match l with
    | []    ⇒ 0
    | x::xs ⇒ S (length xs)
  end.
```

Definition (list length)

```
Fixpoint length {A: Type} (l: list A): nat ≜
  match l with
    | []    ⇒ 0
    | x::xs ⇒ S (length xs)
  end.
```

Lemma

length of appended list is sum of lengths, that is,

```
Lemma app_length: ∀ {A: Type} (l1 l2: list A),  length (append l1 l2) = length l1 + length l2.
```

Definition (list length)

```
Fixpoint length {A: Type} (l: list A): nat ≜
  match l with
    | []    ⇒ 0
    | x::xs ⇒ S (length xs)
  end.
```

Lemma

length of appended list is sum of lengths, that is,

```
Lemma app_length: ∀ {A: Type} (l1 l2: list A),  length (append l1 l2) = length l1 + length l2.
```

Proof.

```
Lemma app_length: ∀ {A: Type} (l1 l2: list A), length (append l1 l2) = length l1 + length l2.
Proof. intros A l1.
       induction l1; intros.
       - simpl. easy.
       - simpl. rewrite IHl1. easy.
Qed.
```

Definition (binary trees)

```
Inductive BTree: Set → Type ≜
  | Empty A: BTree A
  | Node A : BTree A → BTree A → BTree A.

(** induction principle *)
BTree_ind =
fun (P : ∀ S : Set, BTree S → Prop) (f : ∀ A : Set, P A (Empty A))
  (f0 : ∀ (A : Set) (b : BTree A),
        P A b → ∀ b0 : BTree A, P A b0 → P A (Node A b b0)) ⇒
fix F (S : Set) (b : BTree S) {struct b} : P S b ≜
  match b as b0 in (BTree S0) return (P S0 b0) with
  | Empty A ⇒ f A
  | Node A b0 b1 ⇒ f0 A b0 (F A b0) b1 (F A b1)
  end
     : ∀ P : ∀ S : Set, BTree S → Prop,
       (∀ A : Set, P A (Empty A)) →
       (∀ (A : Set) (b : BTree A),
        P A b → ∀ b0 : BTree A, P A b0 → P A (Node A b b0)) →
       ∀ (S : Set) (b : BTree S), P S b
```

Definition (hight, perfectness and size of a binary tree)

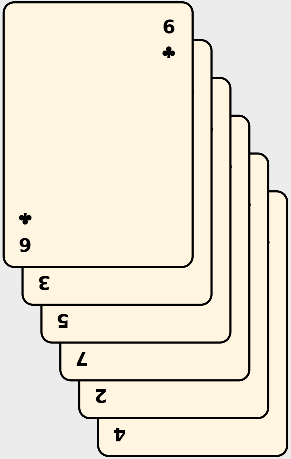
```
Fixpoint height {A: Set} (t: BTree A): nat ≜
  match t with
    | Empty A    ⇒ 0
    | Node A l r ⇒ max (@height A l) (@height A r) + 1
  end.

Fixpoint perfect {A: Set} (t: BTree A): bool ≜
  match t with
    | Empty A    ⇒ true
    | Node A l r ⇒ Nat.eqb (height l) (height r) && perfect l && perfect r
  end.

Fixpoint size {A: Set} (t: BTree A): nat ≜
  match t with
    | Empty A    ⇒ 0
    | Node A l r ⇒ size l + size r + 1
  end.
```

Lemma

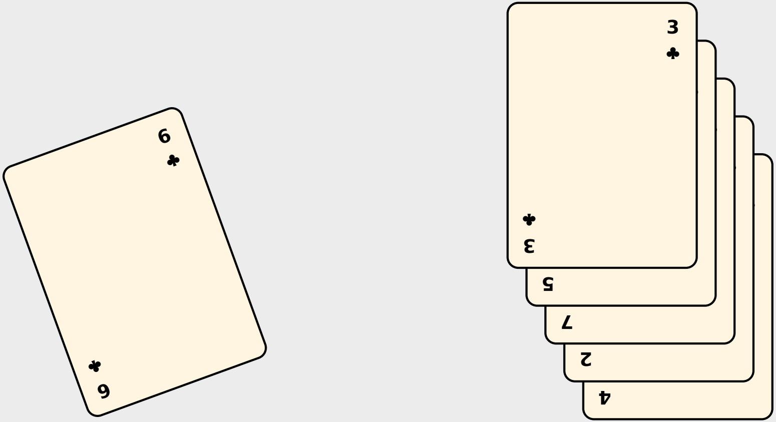
a perfect binary tree t of height n has exactly $2^n - 1$ nodes, that is,

```
Lemma perfectness: ∀ {A: Set} (t: BTree A), perfect t = true → size t = Nat.pow 2 (height t) - 1.
```

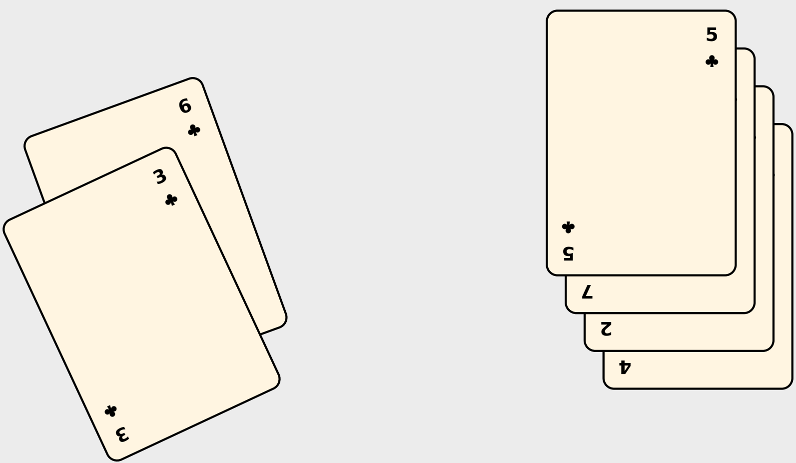
Example (Insertion Sort)



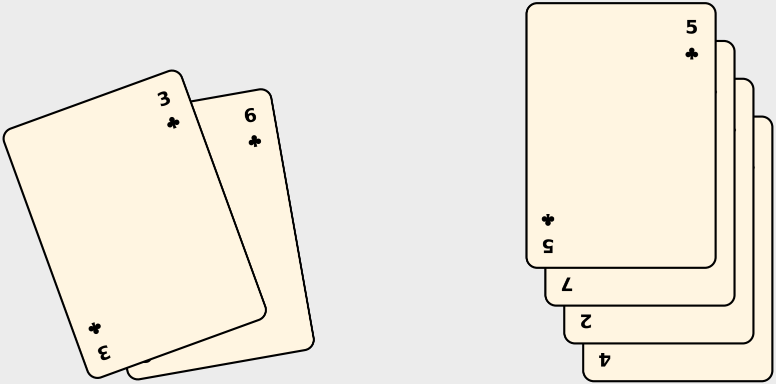
Example (Insertion Sort)



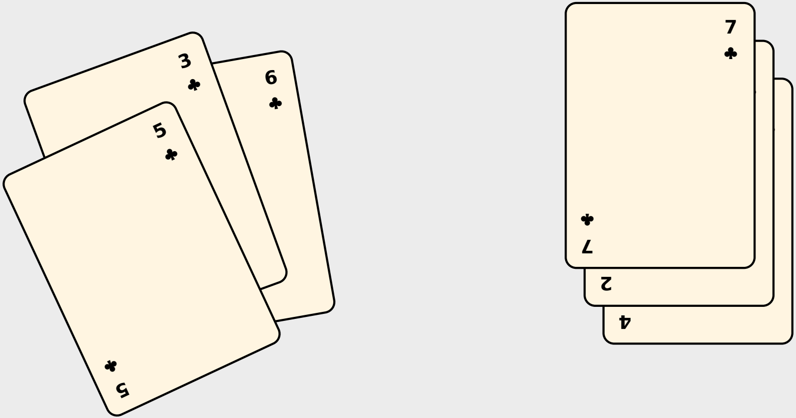
Example (Insertion Sort)



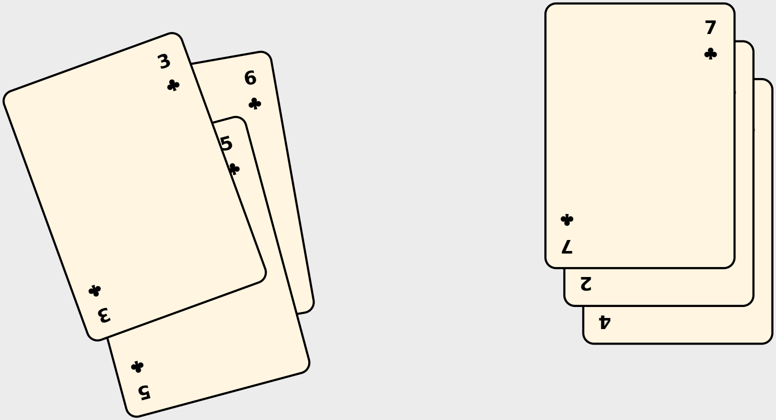
Example (Insertion Sort)



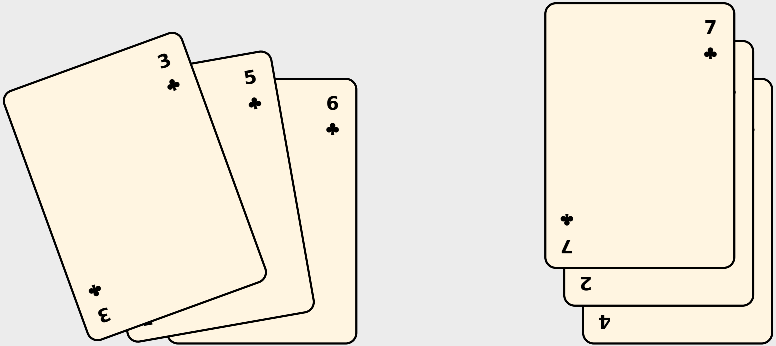
Example (Insertion Sort)



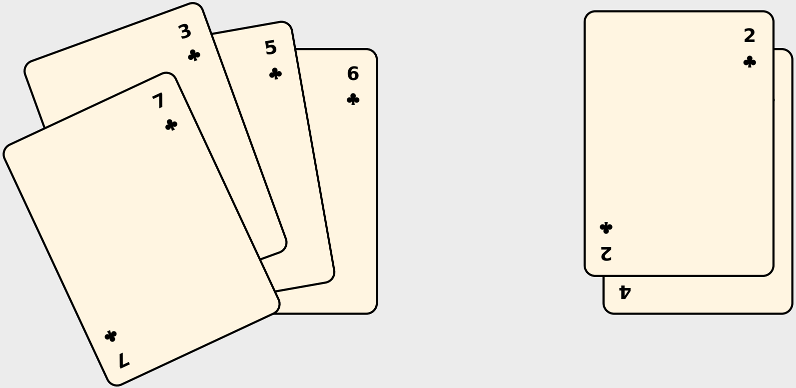
Example (Insertion Sort)



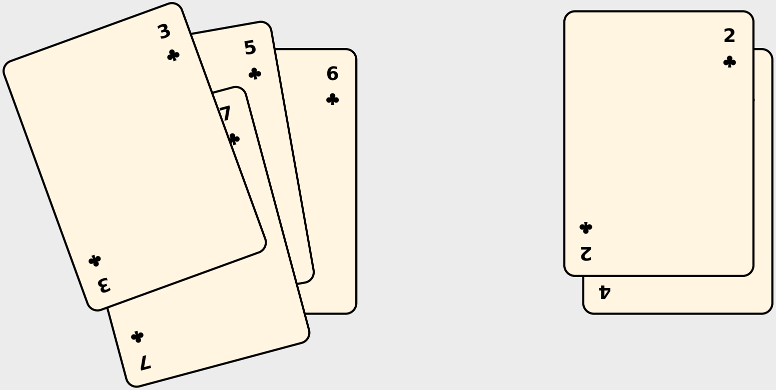
Example (Insertion Sort)



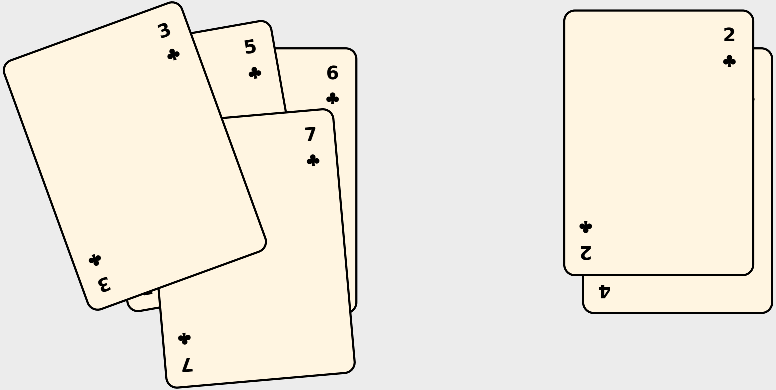
Example (Insertion Sort)



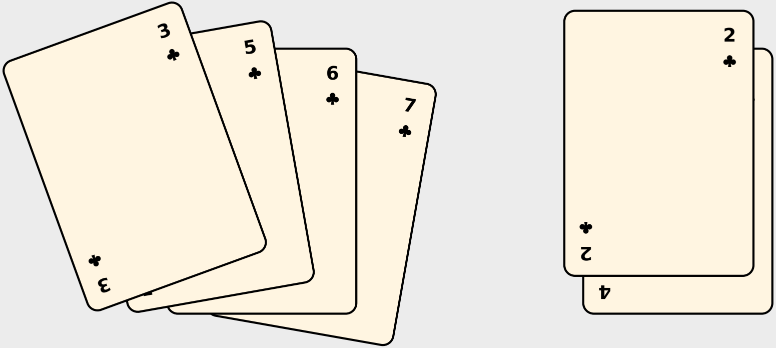
Example (Insertion Sort)



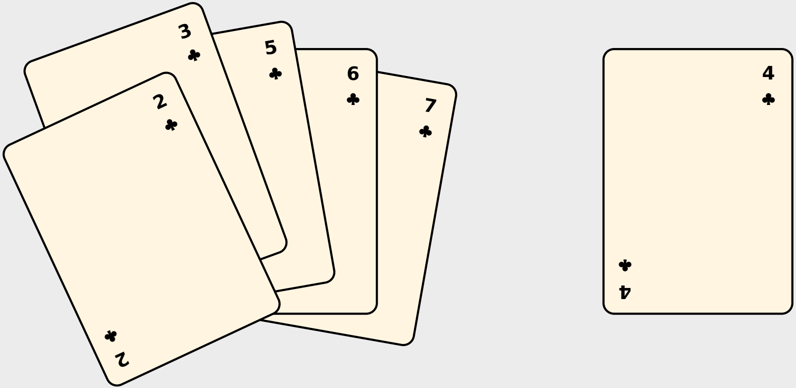
Example (Insertion Sort)



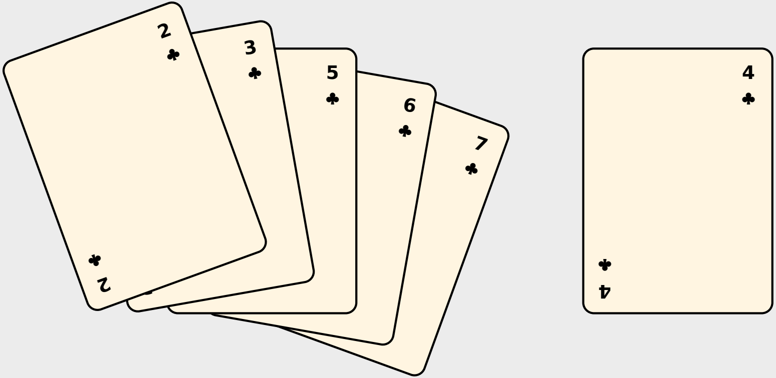
Example (Insertion Sort)



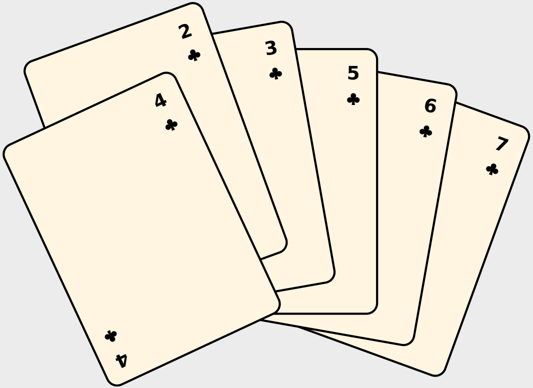
Example (Insertion Sort)



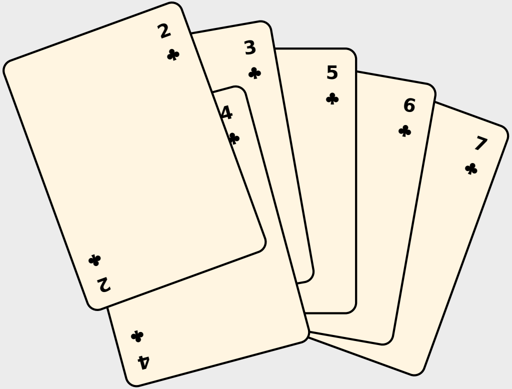
Example (Insertion Sort)



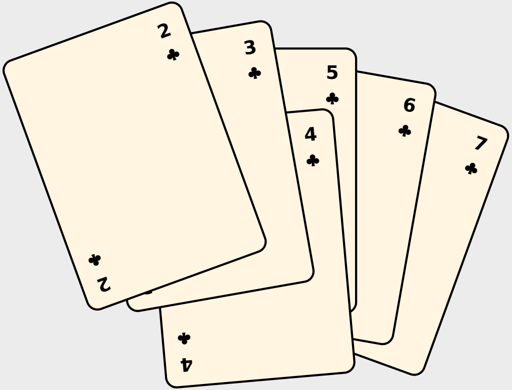
Example (Insertion Sort)



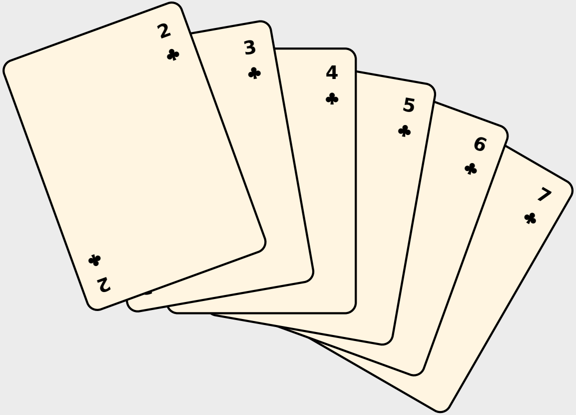
Example (Insertion Sort)



Example (Insertion Sort)



Example (Insertion Sort)



Definition (inserting an element into a sorted list)

```
Fixpoint insert (a: nat) (l: list nat): list nat ≜
  match l with
    | []    ⇒ [a]
    | x::xs ⇒ if Nat.leb a x then a::x::xs
              else x::(insert a xs)
  end.
```

Definition (inserting an element into a sorted list)

```
Fixpoint insert (a: nat) (l: list nat): list nat ≜
  match l with
    | []    ⇒ [a]
    | x::xs ⇒ if Nat.leb a x then a::x::xs
              else x::(insert a xs)
  end.
```

Definition (sorting by repeatedly inserting elements into the empty list)

```
Definition insertionsort ≜ foldr insert [].
```

Theorem

Insertion sort is a valid sorting algorithm

Theorem

Insertion sort is a valid sorting algorithm, that is, result after applying insertion sort is sorted

Theorem

Insertion sort is a valid sorting algorithm, that is, result after applying insertion sort is sorted

equivalently in Coq

Theorem

Insertion sort is a valid sorting algorithm, that is, result after applying insertion sort is sorted

equivalently in Coq

Theorem

```
Theorem insertionsort_sorts l: is_sorted (insertionsort l) = true.
```

where

Definition (is sorted predicate)

```
Fixpoint is_sorted (l:list nat) : bool ≜
  match l with
    | nil ⇒ true
    | x::xs ⇒
      match xs with
        | nil ⇒ true
        | y::_ ⇒ (Nat.leb x y) && (is_sorted xs)
      end
  end.
```

Thanks! & Questions?