Corso “Programmazione 1”
Capitolo 11: Strutture Dati Astratte

Docente:	Roberto Sebastiani - roberto.sebastiani@unitn.it
Esercitatori:	Mario Passamani - mario.passamani@unitn.it
	Alessandro Tomasi - alessandro.tomasi@unitn.it

C.D.L.:	Informatica (INF)
	Ing. Informatica, delle Comunicazioni ed Elettronica (ICE)
	Studenti con numero di matricola pari
A.A.:	2019-2020
Luogo:	DISI, Università di Trento
URL:	disi.unitn.it/rseba/DIDATTICA/prog1_2020/

Ultimo aggiornamento: 30 ottobre 2019, ore 11:29

Outline

Outline

Tipo di Dato Astratto

Un tipo di dato astratto (TDA) è un insieme di valori e di operazioni definite su di essi in modo indipendente dalla loro implementazione

- Per definire un tipo di dato astratto occorre specificare:
  - i dati immagazzinati
  - le operazioni supportate
  - le eventuali condizioni di errore associate alle operazioni
- Per lo stesso TDA si possono avere più implementazioni
  - diversa implementazione, diverse caratteristiche computazionali (efficienza, uso di memoria, ecc.)
  - stessa interfaccia (stessi header di funzioni, riportati in un file `.h`)
    $\Longrightarrow$ implementazioni interscambiabili in un programma
- È spesso desiderabile nascondere l'implementazione di un TDA (information hiding): solo i file `.h` e `.o` disponibili

N.B.: La nozione di TDA è la base della programmazione ad oggetti.

Tipo di Dato Astratto

Un tipo di dato astratto (TDA) è un insieme di valori e di operazioni definite su di essi in modo indipendente dalla loro implementazione

- Per definire un tipo di dato astratto occorre specificare:
  - i dati immagazzinati
  - le operazioni supportate
  - le eventuali condizioni di errore associate alle operazioni
- Per lo stesso TDA si possono avere più implementazioni
  - diversa implementazione, diverse caratteristiche computazionali (efficienza, uso di memoria, ecc.)
  - stessa interfaccia (stessi header di funzioni, riportati in un file `.h`)
    $\Longrightarrow$ implementazioni interscambiabili in un programma
- È spesso desiderabile nascondere l'implementazione di un TDA (information hiding): solo i file `.h` e `.o` disponibili

N.B.: La nozione di TDA è la base della programmazione ad oggetti.

Tipo di Dato Astratto

Un tipo di dato astratto (TDA) è un insieme di valori e di operazioni definite su di essi in modo indipendente dalla loro implementazione

- Per definire un tipo di dato astratto occorre specificare:
  - i dati immagazzinati
  - le operazioni supportate
  - le eventuali condizioni di errore associate alle operazioni
- Per lo stesso TDA si possono avere più implementazioni
  - diversa implementazione, diverse caratteristiche computazionali (efficienza, uso di memoria, ecc.)
  - stessa interfaccia (stessi header di funzioni, riportati in un file `.h`)
    $\Longrightarrow$ implementazioni interscambiabili in un programma
- È spesso desiderabile nascondere l'implementazione di un TDA (information hiding): solo i file `.h` e `.o` disponibili

N.B.: La nozione di TDA è la base della programmazione ad oggetti.

Tipo di Dato Astratto

Un tipo di dato astratto (TDA) è un insieme di valori e di operazioni definite su di essi in modo indipendente dalla loro implementazione

- Per definire un tipo di dato astratto occorre specificare:
  - i dati immagazzinati
  - le operazioni supportate
  - le eventuali condizioni di errore associate alle operazioni
- Per lo stesso TDA si possono avere più implementazioni
  - diversa implementazione, diverse caratteristiche computazionali (efficienza, uso di memoria, ecc.)
  - stessa interfaccia (stessi header di funzioni, riportati in un file `.h`)
    $\Longrightarrow$ implementazioni interscambiabili in un programma
- È spesso desiderabile nascondere l'implementazione di un TDA (information hiding): solo i file `.h` e `.o` disponibili

N.B.: La nozione di TDA è la base della programmazione ad oggetti.

Tipo di Dato Astratto

Un tipo di dato astratto (TDA) è un insieme di valori e di operazioni definite su di essi in modo indipendente dalla loro implementazione

- Per definire un tipo di dato astratto occorre specificare:
  - i dati immagazzinati
  - le operazioni supportate
  - le eventuali condizioni di errore associate alle operazioni
- Per lo stesso TDA si possono avere più implementazioni
  - diversa implementazione, diverse caratteristiche computazionali (efficienza, uso di memoria, ecc.)
  - stessa interfaccia (stessi header di funzioni, riportati in un file `.h`)
    $\Longrightarrow$ implementazioni interscambiabili in un programma
- È spesso desiderabile nascondere l'implementazione di un TDA (information hiding): solo i file `.h` e `.o` disponibili

N.B.: La nozione di TDA è la base della programmazione ad oggetti.

Esempi Molto Importanti di Tipi di Dato Astratto

- Le Pile (Stack)
- Le Code (Queue)
- Gli Alberi (Tree)

Outline

Outline

Le Pile (Stack)

- Una pila è una collezione di dati omogenei (e.g., puntatori a struct) in cui gli elementi sono gestiti in modo LIFO (Last In First Out)
  - Viene visualizzato/estratto l'elemento inserito più recentemente
  - Es: una scatola alta e stretta contenente documenti
- Operazioni tipiche definite su una pila di oggetti di tipo `T`:
  - `init()`/`deinit()`: inizializza/deinizializza la pila
  - `push(T)`: inserisce elemento sulla pila; fallisce se piena
  - `pop()`: estrae l'ultimo elemento inserito (senza visualizzarlo); fallisce se vuota
  - `top(T &)`: ritorna l'ultimo elemento inserito (senza estrarlo); fallisce se vuota
- Varianti:
  - `pop()` e `top(T &)` fuse in un'unica operazione `pop(T &)`
  - talvolta disponibili anche `print()`
  - [`deinit()` non sempre presente]

Le Pile (Stack)

- Una pila è una collezione di dati omogenei (e.g., puntatori a struct) in cui gli elementi sono gestiti in modo LIFO (Last In First Out)
  - Viene visualizzato/estratto l'elemento inserito più recentemente
  - Es: una scatola alta e stretta contenente documenti
- Operazioni tipiche definite su una pila di oggetti di tipo `T`:
  - `init()`/`deinit()`: inizializza/deinizializza la pila
  - `push(T)`: inserisce elemento sulla pila; fallisce se piena
  - `pop()`: estrae l'ultimo elemento inserito (senza visualizzarlo); fallisce se vuota
  - `top(T &)`: ritorna l'ultimo elemento inserito (senza estrarlo); fallisce se vuota
- Varianti:
  - `pop()` e `top(T &)` fuse in un'unica operazione `pop(T &)`
  - talvolta disponibili anche `print()`
  - [`deinit()` non sempre presente]

Le Pile (Stack)

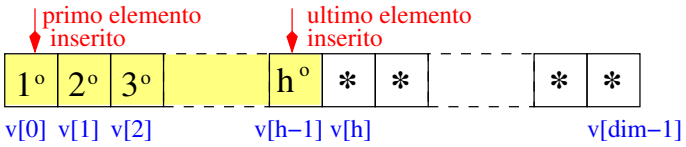
- Una pila è una collezione di dati omogenei (e.g., puntatori a struct) in cui gli elementi sono gestiti in modo LIFO (Last In First Out)
  - Viene visualizzato/estratto l'elemento inserito più recentemente
  - Es: una scatola alta e stretta contenente documenti
- Operazioni tipiche definite su una pila di oggetti di tipo `T`:
  - `init()`/`deinit()`: inizializza/deinizializza la pila
  - `push(T)`: inserisce elemento sulla pila; fallisce se piena
  - `pop()`: estrae l'ultimo elemento inserito (senza visualizzarlo); fallisce se vuota
  - `top(T &)`: ritorna l'ultimo elemento inserito (senza estrarlo); fallisce se vuota
- Varianti:
  - `pop()` e `top(T &)` fuse in un'unica operazione `pop(T &)`
  - talvolta disponibili anche `print()`
  - [`deinit()` non sempre presente]

Le Pile (Stack) II

Nota importante

In tutte le possibili implementazioni di una pila, le operazioni `push(T)`, `pop()`, `top(T &)` devono richiedere un numero costante di passi computazionali, indipendente dal numero di elementi contenuti nella pila!

Implementazione di una pila mediante array



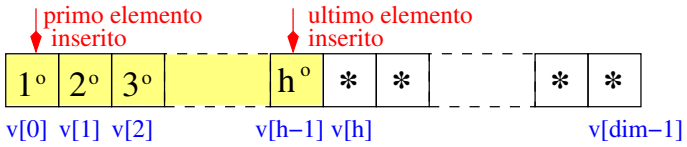
- Dati: un intero `h` e un array `v` di `dim` elementi di tipo `T`
  - `v` allocato staticamente o dinamicamente
  - `h` indice del prossimo elemento da inserire (inizialmente 0)

  $\Longrightarrow$ numero di elementi contenuti nella pila: `h`
  - pila vuota: `h==0`
  - pila piena: `h==dim`

  $\Longrightarrow$ massimo numero di elementi contenuti nella pila: `dim`

N.B.: `dim` elementi sempre allocati.

Implementazione di una pila mediante array



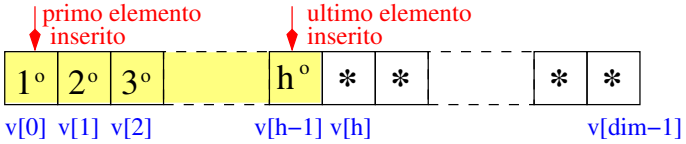
- Dati: un intero `h` e un array `v` di `dim` elementi di tipo `T`
  - `v` allocato staticamente o dinamicamente
  - `h` indice del prossimo elemento da inserire (inizialmente 0)

  $\Longrightarrow$ numero di elementi contenuti nella pila: `h`
  - pila vuota: `h==0`
  - pila piena: `h==dim`

  $\Longrightarrow$ massimo numero di elementi contenuti nella pila: `dim`

N.B.: `dim` elementi sempre allocati.

Implementazione di una pila mediante array II



- Funzionalità:
  - `init()`: pone `h=0` (alloca `v` se allocazione dinamica)
  - `push(T)`: inserisce l'elemento in `v[h]`, incrementa `h`
  - `pop()`: decrementa `h`
  - `top(T &)`: restituisce `v[h-1]`
  - `deinit()`: dealloca `v` se allocazione dinamica

Esempi su pile di interi

- semplice stack di interi come struct:
  - `STACK_QUEUE_ARRAY/struct_stack.h`
  - `STACK_QUEUE_ARRAY/struct_stack.cc`
  - `STACK_QUEUE_ARRAY/struct_stack_main.cc`
- uso di stack per invertire l'ordine:
  - `STACK_QUEUE_ARRAY/struct_stack.h`
  - `STACK_QUEUE_ARRAY/struct_stack.cc`
  - `STACK_QUEUE_ARRAY/struct_reverse_main.cc`

(`struct_stack.h|.cc` stessi del caso precedente)

Esempi su pile di interi

- semplice stack di interi come struct:
  - `STACK_QUEUE_ARRAY/struct_stack.h`
  - `STACK_QUEUE_ARRAY/struct_stack.cc`
  - `STACK_QUEUE_ARRAY/struct_stack_main.cc`
- uso di stack per invertire l'ordine:
  - `STACK_QUEUE_ARRAY/struct_stack.h`
  - `STACK_QUEUE_ARRAY/struct_stack.cc`
  - `STACK_QUEUE_ARRAY/struct_reverse_main.cc`

  (`struct_stack.h|.cc` stessi del caso precedente)

Outline

Le Code (Queue)

- Una coda è una collezione di dati omogenei in cui gli elementi sono gestiti in modo FIFO (First In First Out)
  - Viene visualizzato/estratto l'elemento inserito meno recentemente
  - Es: una coda ad uno sportello
- Operazioni tipiche definite su una coda di oggetti di tipo `T`:
  - `init()`/`deinit()`: inizializza/deinizializza la coda
  - `enqueue(T)`: inserisce elemento sulla coda; fallisce se piena
  - `dequeue()`: estrae il primo elemento inserito (senza visualizzarlo); fallisce se vuota
  - `first(T &)`: ritorna il primo elemento inserito (senza estrarlo); fallisce se vuota
- Varianti:
  - `dequeue()` e `first(T &)` fuse in un'unica operazione `dequeue(T &)`
  - talvolta disponibili anche `print()`
  - [`deinit()` non sempre presente]

Le Code (Queue)

- Una coda è una collezione di dati omogenei in cui gli elementi sono gestiti in modo FIFO (First In First Out)
  - Viene visualizzato/estratto l'elemento inserito meno recentemente
  - Es: una coda ad uno sportello
- Operazioni tipiche definite su una coda di oggetti di tipo `T`:
  - `init()`/`deinit()`: inizializza/deinizializza la coda
  - `enqueue(T)`: inserisce elemento sulla coda; fallisce se piena
  - `dequeue()`: estrae il primo elemento inserito (senza visualizzarlo); fallisce se vuota
  - `first(T &)`: ritorna il primo elemento inserito (senza estrarlo); fallisce se vuota
- Varianti:
  - `dequeue()` e `first(T &)` fuse in un'unica operazione `dequeue(T &)`
  - talvolta disponibili anche `print()`
  - [`deinit()` non sempre presente]

Le Code (Queue)

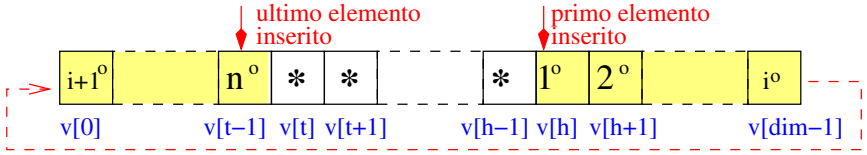
- Una coda è una collezione di dati omogenei in cui gli elementi sono gestiti in modo FIFO (First In First Out)
  - Viene visualizzato/estratto l'elemento inserito meno recentemente
  - Es: una coda ad uno sportello
- Operazioni tipiche definite su una coda di oggetti di tipo `T`:
  - `init()`/`deinit()`: inizializza/deinizializza la coda
  - `enqueue(T)`: inserisce elemento sulla coda; fallisce se piena
  - `dequeue()`: estrae il primo elemento inserito (senza visualizzarlo); fallisce se vuota
  - `first(T &)`: ritorna il primo elemento inserito (senza estrarlo); fallisce se vuota
- Varianti:
  - `dequeue()` e `first(T &)` fuse in un'unica operazione `dequeue(T &)`
  - talvolta disponibili anche `print()`
  - [`deinit()` non sempre presente]

Le Code (Queue) II

Nota importante

In tutte le possibili implementazioni di una coda, le operazioni `enqueue(T)`, `dequeue()`, `first(T &)` devono richiedere un numero costante di passi computazionali, indipendente dal numero di elementi contenuti nella coda!

Implementazione di una coda mediante array



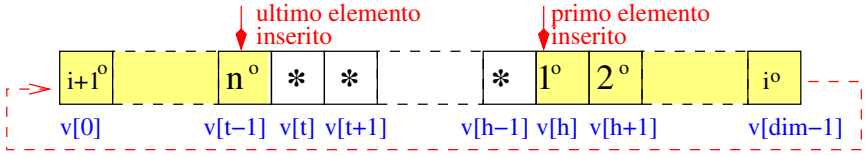
- Idea: buffer circolare: `succ(i)==(i+1)%dim`
- Dati: due interi `h,t` e un array `v` di `dim` elementi di tipo `T`
  - `v` allocato staticamente o dinamicamente
  - `h` indice del più vecchio elemento inserito (inizialmente 0)
  - `t` indice del prossimo elemento da inserire (inizialmente 0)

  $\Longrightarrow$ num. di elementi contenuti nella coda: `n=(t>=h ? t-h : t-h+dim)`
  - coda vuota: `t==h`
  - coda piena: `succ(t)==h`

  $\Longrightarrow$ massimo numero di elementi contenuti nella coda: `dim-1`

N.B.: `dim` elementi sempre allocati.

Implementazione di una coda mediante array



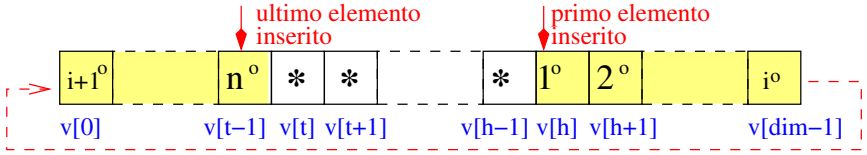
- Idea: buffer circolare: `succ(i)==(i+1)%dim`
- Dati: due interi `h,t` e un array `v` di `dim` elementi di tipo `T`
  - `v` allocato staticamente o dinamicamente
  - `h` indice del più vecchio elemento inserito (inizialmente 0)
  - `t` indice del prossimo elemento da inserire (inizialmente 0)

  $\Longrightarrow$ num. di elementi contenuti nella coda: `n=(t>=h ? t-h : t-h+dim)`
  - coda vuota: `t==h`
  - coda piena: `succ(t)==h`

  $\Longrightarrow$ massimo numero di elementi contenuti nella coda: `dim-1`

N.B.: `dim` elementi sempre allocati.

Implementazione di una coda mediante array



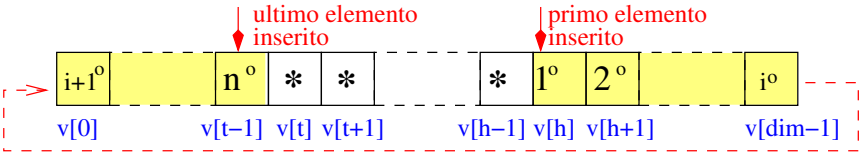
- Idea: buffer circolare: `succ(i)==(i+1)%dim`
- Dati: due interi `h,t` e un array `v` di `dim` elementi di tipo `T`
  - `v` allocato staticamente o dinamicamente
  - `h` indice del più vecchio elemento inserito (inizialmente 0)
  - `t` indice del prossimo elemento da inserire (inizialmente 0)

  $\Longrightarrow$ num. di elementi contenuti nella coda: `n=(t>=h ? t-h : t-h+dim)`
  - coda vuota: `t==h`
  - coda piena: `succ(t)==h`

  $\Longrightarrow$ massimo numero di elementi contenuti nella coda: `dim-1`

N.B.: `dim` elementi sempre allocati.

Implementazione di una coda mediante array II



- Funzionalità:
  - `init()`: pone `h=t=0` (alloca `v` se allocazione dinamica)
  - `enqueue(T)`: inserisce l'elemento in `v[t]`, "incrementa" `t` (`t=succ(t)`)
  - `dequeue()`: "incrementa" `h`
  - `first(T &)`: restituisce `v[h]`
  - `deinit()`: dealloca `v` se allocazione dinamica

Esempi su code di interi

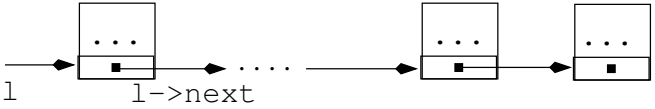
- semplice coda di interi come struct:
  - `STACK_QUEUE_ARRAY/struct_queue.h`
  - `STACK_QUEUE_ARRAY/struct_queue.cc`
  - `STACK_QUEUE_ARRAY/struct_queue_main.cc`

Esercizi proposti

Vedere file `ESERCIZI_PROPOSTI.txt`

Outline

Liste Concatenate



- (Nella sua versione più semplice) una lista concatenata `l` di oggetti di tipo `T` è definita come segue:
  - `l` è un puntatore `NULL` (lista vuota) oppure
  - `l` è un puntatore ad un nodo (struct) contenente:
    - un campo `value` di tipo `T`
    - un campo `next` di tipo lista concatenata

```
struct node;
typedef node * lista;
struct node { T value;  lista next; };
```

- Opzionalmente, possono esserci puntatori ad altri elementi

Una lista concatenata è una struttura dati dinamica, la cui struttura si evolve con l'immissione e estrazione di elementi.

Liste Concatenate

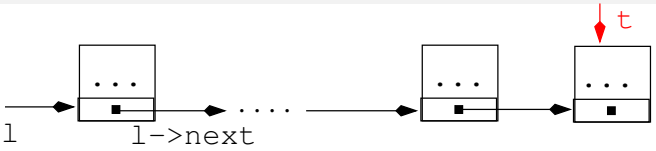
- (Nella sua versione più semplice) una lista concatenata `l` di oggetti di tipo `T` è definita come segue:
  - `l` è un puntatore `NULL` (lista vuota) oppure
  - `l` è un puntatore ad un nodo (struct) contenente:
    - un campo `value` di tipo `T`
    - un campo `next` di tipo lista concatenata

```
struct node;
typedef node * lista;
struct node { T value;  lista next; };
```

- Opzionalmente, possono esserci puntatori ad altri elementi

Una lista concatenata è una struttura dati dinamica, la cui struttura si evolve con l'immissione e estrazione di elementi.

Liste Concatenate



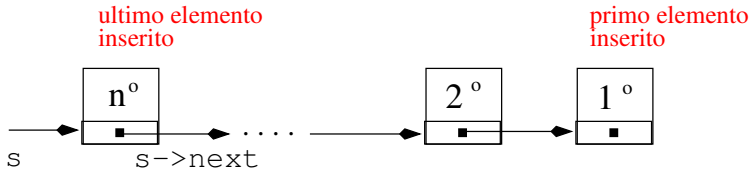
- (Nella sua versione più semplice) una lista concatenata `l` di oggetti di tipo `T` è definita come segue:
  - `l` è un puntatore `NULL` (lista vuota) oppure
  - `l` è un puntatore ad un nodo (struct) contenente:
    - un campo `value` di tipo `T`
    - un campo `next` di tipo lista concatenata

```
struct node;
typedef node * lista;
struct node { T value;  lista next; };
```

- Opzionalmente, possono esserci puntatori ad altri elementi

Una lista concatenata è una struttura dati dinamica, la cui struttura si evolve con l'immissione e estrazione di elementi.

Implementazione di una pila come lista concatenata

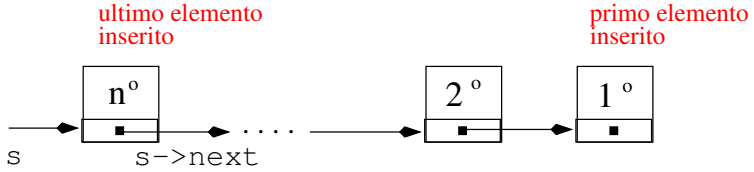


- Dati: una lista concatenata `s` di `n` elementi
  - `s` punta all'ultimo elemento inserito nella pila (inizialmente NULL)
  - [Opzionalmente un intero `n` con il numero di elementi nella lista]
  - l'ultimo elemento della lista contiene il primo elemento inserito.
  - pila vuota: `s==NULL`
  - pila piena: `out of memory`

  $\Longrightarrow$ numero di elementi contenuti nella pila limitato dalla memoria

N.B.: allocati solo gli `n` nodi necessari a contenere gli elementi

Implementazione di una pila come lista concatenata

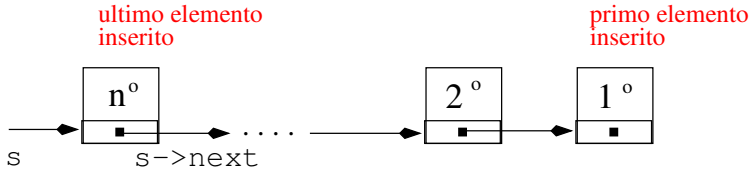


- Dati: una lista concatenata `s` di `n` elementi
  - `s` punta all'ultimo elemento inserito nella pila (inizialmente NULL)
  - [Opzionalmente un intero `n` con il numero di elementi nella lista]
  - l'ultimo elemento della lista contiene il primo elemento inserito.
  - pila vuota: `s==NULL`
  - pila piena: `out of memory`

  $\Longrightarrow$ numero di elementi contenuti nella pila limitato dalla memoria

N.B.: allocati solo gli `n` nodi necessari a contenere gli elementi

Implementazione di una pila come lista concatenata II



- Funzionalità:
  - `init()`: pone `s=NULL`
  - `push(T)`:
    1. alloca un nuovo nodo ad un puntatore `tmp`
    2. copia l'elemento in `tmp->value`
    3. assegna `tmp->next=s`, e `s=tmp`
  - `pop()`:
    1. fa puntare un nuovo puntatore `first` al primo nodo: `first=s`
    2. `s` aggira il primo nodo: `s=s->next`
    3. dealloca (l'ex) primo nodo: `delete first`
  - `top(T &)`: restituisce `s->value`
  - `deinit()`: ripete `pop()` finché la pila non è vuota

Esempi su pile di interi

- semplice stack di interi come struct:
  - `STACK_QUEUE_PUNT/struct_stack.h`
  - `STACK_QUEUE_PUNT/struct_stack.cc`
  - `STACK_QUEUE_PUNT/struct_stack_main.cc`
- uso di stack per invertire l'ordine:
  - `STACK_QUEUE_PUNT/struct_stack.h`
  - `STACK_QUEUE_PUNT/struct_stack.cc`
  - `STACK_QUEUE_PUNT/struct_reverse_main.cc`

N.B. I "main" e gli header delle funzioni identici a quelli in `STACK_QUEUE_ARRAY` $\Longrightarrow$ Tipo di Dato Astratto

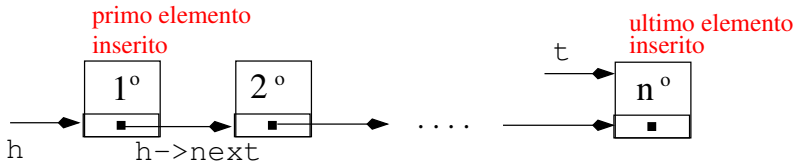
Esempi su pile di interi

- semplice stack di interi come struct:
  - `STACK_QUEUE_PUNT/struct_stack.h`
  - `STACK_QUEUE_PUNT/struct_stack.cc`
  - `STACK_QUEUE_PUNT/struct_stack_main.cc`
- uso di stack per invertire l'ordine:
  - `STACK_QUEUE_PUNT/struct_stack.h`
  - `STACK_QUEUE_PUNT/struct_stack.cc`
  - `STACK_QUEUE_PUNT/struct_reverse_main.cc`

N.B. I "main" e gli header delle funzioni identici a quelli in `STACK_QUEUE_ARRAY` $\Longrightarrow$ Tipo di Dato Astratto

Outline

Implementazione di una coda come lista concatenata

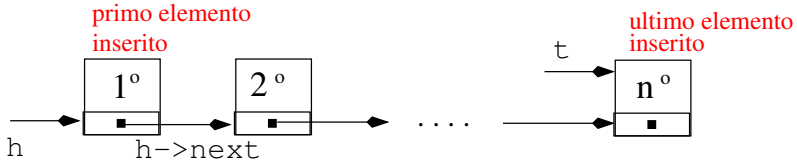


- Dati: una lista concatenata `h` di `n` elementi di tipo `T`, un puntatore `t` all'ultimo elemento
  - `h` punta al primo elemento inserito nella coda (inizialmente NULL)
  - `t` punta all'ultimo elemento inserito nella coda
  - [Opzionalmente un intero `n` con il numero di elementi nella lista]
  - coda vuota: `h==NULL`
  - coda piena: `out of memory`

$\Longrightarrow$ numero di elementi contenuti nella coda limitato dalla memoria

N.B.: allocati solo gli `n` nodi necessari a contenere gli elementi

Implementazione di una coda come lista concatenata



- Dati: una lista concatenata `h` di `n` elementi di tipo `T`, un puntatore `t` all'ultimo elemento
  - `h` punta al primo elemento inserito nella coda (inizialmente NULL)
  - `t` punta all'ultimo elemento inserito nella coda
  - [Opzionalmente un intero `n` con il numero di elementi nella lista]
  - coda vuota: `h==NULL`
  - coda piena: `out of memory`

  $\Longrightarrow$ numero di elementi contenuti nella coda limitato dalla memoria

N.B.: allocati solo gli `n` nodi necessari a contenere gli elementi

Implementazione di una coda come lista conc. II

- Funzionalità:
  - `init()`: pone `h=NULL`
  - `enqueue(T)`:
    1. alloca un nuovo nodo ad un puntatore `tmp`
    2. copia l'elemento in `tmp->value` e pone `tmp->next=NULL`
    3. (se coda non vuota) assegna `t->next=tmp`, e `t=tmp`
       (se coda vuota) assegna `h=tmp`, e `t=tmp`
  - `dequeue()`: come `pop()` della pila con il puntatore `h`
  - `first(T &)`: come `top()` della pila con il puntatore `h`
  - `deinit()`: ripete `dequeue()` finché la coda non è vuota

Esempi su code di interi

- semplice queue di interi come struct:
  - `STACK_QUEUE_PUNT/struct_queue.h`
  - `STACK_QUEUE_PUNT/struct_queue.cc`
  - `STACK_QUEUE_PUNT/struct_queue_main.cc`

N.B. I “main” e gli header delle funzioni identici a quelli in `STACK_QUEUE_ARRAY` $\Longrightarrow$ Tipo di Dato Astratto

Esercizi proposti

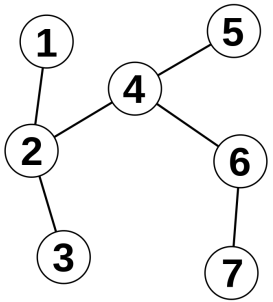
Vedere file `ESERCIZI_PROPOSTI.txt`

Outline

Alberi (Teoria dei grafi)

In teoria dei grafi un albero è un grafo non orientato nel quale due vertici qualsiasi sono connessi da uno e un solo cammino

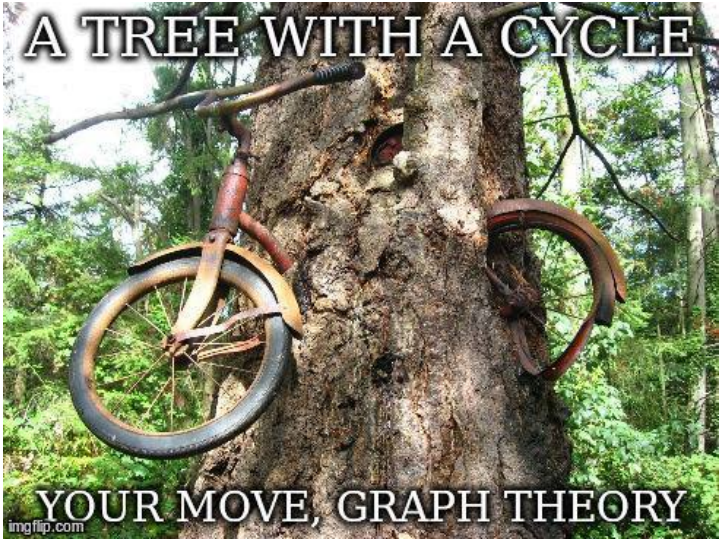
- grafo non orientato, connesso e privo di cicli.



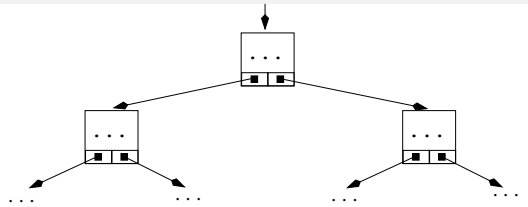
“Tree: a connected graph without cycles”

[Handbook of Graph Theory]

“Tree: a connected graph without cycles”:



Alberi binari



- (Nella sua versione più semplice) una albero binario `t` di oggetti di tipo `T` è definito come segue:
  - `t` è un puntatore `NULL` (albero vuoto) oppure
  - `t` è un puntatore ad un nodo (struct) contenente:
    - un campo `value` di tipo `T`
    - due campi `left right` di tipo albero

```
struct node;
typedef node * albero;
struct node { T value;  albero left, right; };
```

Un albero binario è una struttura dati dinamica.

Alberi binari: terminologia

- i sottoalberi (possibilmente vuoti) di un nodo `N` sono detti sottoalbero sinistro e sottoalbero destro di `N`
- Se un nodo `N` punta nell'ordine a due (eventuali) nodi `N1`, `N2`
  - `N1` e `N2` sono detti rispettivamente figlio sinistro e figlio destro di `N`
  - `N` è detto nodo padre di `N1` e `N2`
- in un albero binario ci possono essere tre tipi di nodi:
  - il nodo radice, che non ha padre
  - i nodi foglia, che non hanno figli
  - i nodi intermedi, che hanno padre e almeno un figlio
- Una catena di nodi dalla radice a una foglia è detta ramo
  - il numero di nodi in un ramo è detto lunghezza del ramo
  - la massima lunghezza di un ramo è detta altezza dell'albero
  - l'altezza di un albero binario di N elementi è $h \in [\lceil log_2(N+1) \rceil, N]$
- Un albero binario di N elementi è bilanciato se la sua altezza è $h = \lceil log_2(N+1) \rceil$
  $\Longrightarrow$ tutti i rami hanno lunghezza `h` o `h-1`
- Un albero binario di N elementi è completo se la sua altezza è tale che $N = 2^h - 1$
  $\Longrightarrow$ tutti i rami hanno lunghezza `h`

Alberi binari: terminologia

- i sottoalberi (possibilmente vuoti) di un nodo `N` sono detti sottoalbero sinistro e sottoalbero destro di `N`
- Se un nodo `N` punta nell'ordine a due (eventuali) nodi `N1`, `N2`
  - `N1` e `N2` sono detti rispettivamente figlio sinistro e figlio destro di `N`
  - `N` è detto nodo padre di `N1` e `N2`
- in un albero binario ci possono essere tre tipi di nodi:
  - il nodo radice, che non ha padre
  - i nodi foglia, che non hanno figli
  - i nodi intermedi, che hanno padre e almeno un figlio
- Una catena di nodi dalla radice a una foglia è detta ramo
  - il numero di nodi in un ramo è detto lunghezza del ramo
  - la massima lunghezza di un ramo è detta altezza dell'albero
  - l'altezza di un albero binario di N elementi è $h \in [\lceil log_2(N+1) \rceil, N]$
- Un albero binario di N elementi è bilanciato se la sua altezza è $h = \lceil log_2(N+1) \rceil$
  $\Longrightarrow$ tutti i rami hanno lunghezza `h` o `h-1`
- Un albero binario di N elementi è completo se la sua altezza è tale che $N = 2^h - 1$
  $\Longrightarrow$ tutti i rami hanno lunghezza `h`

Alberi binari: terminologia

- i sottoalberi (possibilmente vuoti) di un nodo `N` sono detti sottoalbero sinistro e sottoalbero destro di `N`
- Se un nodo `N` punta nell'ordine a due (eventuali) nodi `N1`, `N2`
  - `N1` e `N2` sono detti rispettivamente figlio sinistro e figlio destro di `N`
  - `N` è detto nodo padre di `N1` e `N2`
- in un albero binario ci possono essere tre tipi di nodi:
  - il nodo radice, che non ha padre
  - i nodi foglia, che non hanno figli
  - i nodi intermedi, che hanno padre e almeno un figlio
- Una catena di nodi dalla radice a una foglia è detta ramo
  - il numero di nodi in un ramo è detto lunghezza del ramo
  - la massima lunghezza di un ramo è detta altezza dell'albero
  - l'altezza di un albero binario di N elementi è $h \in [\lceil log_2(N+1) \rceil, N]$
- Un albero binario di N elementi è bilanciato se la sua altezza è $h = \lceil log_2(N+1) \rceil$
  $\Longrightarrow$ tutti i rami hanno lunghezza `h` o `h-1`
- Un albero binario di N elementi è completo se la sua altezza è tale che $N = 2^h - 1$
  $\Longrightarrow$ tutti i rami hanno lunghezza `h`

Alberi binari: terminologia

- i sottoalberi (possibilmente vuoti) di un nodo `N` sono detti sottoalbero sinistro e sottoalbero destro di `N`
- Se un nodo `N` punta nell'ordine a due (eventuali) nodi `N1`, `N2`
  - `N1` e `N2` sono detti rispettivamente figlio sinistro e figlio destro di `N`
  - `N` è detto nodo padre di `N1` e `N2`
- in un albero binario ci possono essere tre tipi di nodi:
  - il nodo radice, che non ha padre
  - i nodi foglia, che non hanno figli
  - i nodi intermedi, che hanno padre e almeno un figlio
- Una catena di nodi dalla radice a una foglia è detta ramo
  - il numero di nodi in un ramo è detto lunghezza del ramo
  - la massima lunghezza di un ramo è detta altezza dell'albero
  - l'altezza di un albero binario di N elementi è $h \in [\lceil log_2(N+1) \rceil, N]$
- Un albero binario di N elementi è bilanciato se la sua altezza è $h = \lceil log_2(N+1) \rceil$
  $\Longrightarrow$ tutti i rami hanno lunghezza `h` o `h-1`
- Un albero binario di N elementi è completo se la sua altezza è tale che $N = 2^h - 1$
  $\Longrightarrow$ tutti i rami hanno lunghezza `h`

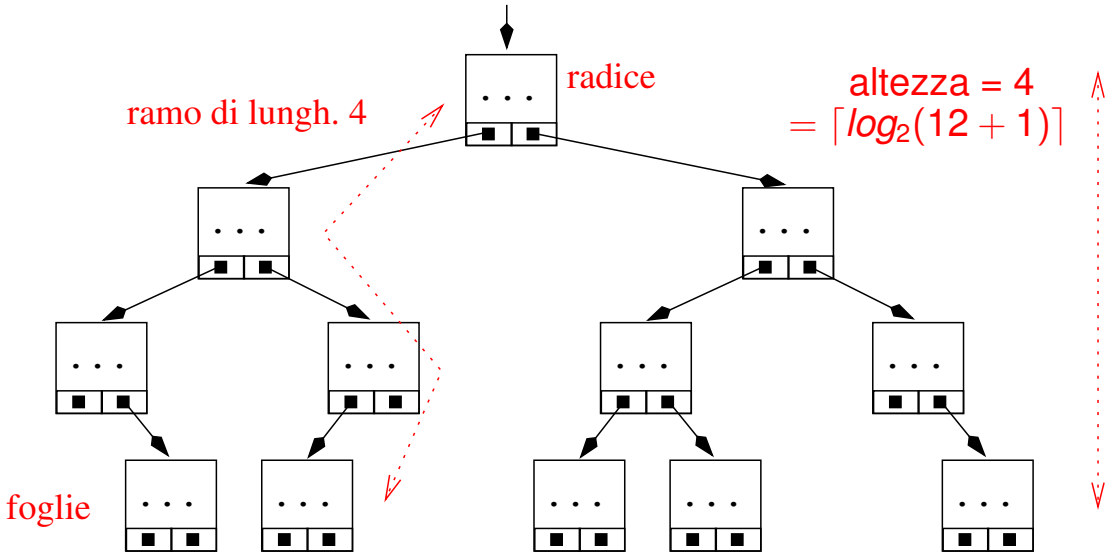
Alberi binari: terminologia

- i sottoalberi (possibilmente vuoti) di un nodo `N` sono detti sottoalbero sinistro e sottoalbero destro di `N`
- Se un nodo `N` punta nell'ordine a due (eventuali) nodi `N1`, `N2`
  - `N1` e `N2` sono detti rispettivamente figlio sinistro e figlio destro di `N`
  - `N` è detto nodo padre di `N1` e `N2`
- in un albero binario ci possono essere tre tipi di nodi:
  - il nodo radice, che non ha padre
  - i nodi foglia, che non hanno figli
  - i nodi intermedi, che hanno padre e almeno un figlio
- Una catena di nodi dalla radice a una foglia è detta ramo
  - il numero di nodi in un ramo è detto lunghezza del ramo
  - la massima lunghezza di un ramo è detta altezza dell'albero
  - l'altezza di un albero binario di N elementi è $h \in [\lceil log_2(N+1) \rceil, N]$
- Un albero binario di N elementi è bilanciato se la sua altezza è $h = \lceil log_2(N+1) \rceil$
  $\Longrightarrow$ tutti i rami hanno lunghezza `h` o `h-1`
- Un albero binario di N elementi è completo se la sua altezza è tale che $N = 2^h - 1$
  $\Longrightarrow$ tutti i rami hanno lunghezza `h`

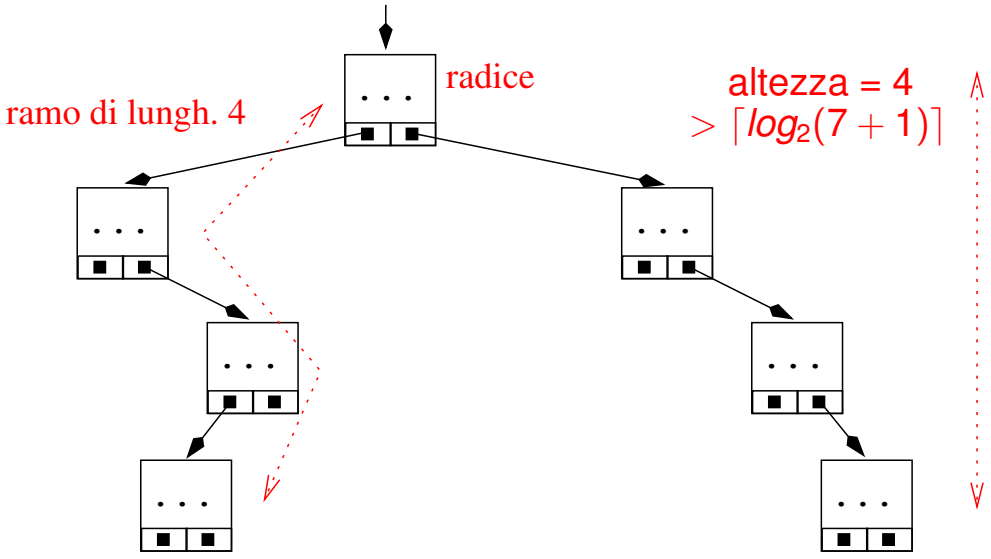
Alberi binari: terminologia

- i sottoalberi (possibilmente vuoti) di un nodo `N` sono detti sottoalbero sinistro e sottoalbero destro di `N`
- Se un nodo `N` punta nell'ordine a due (eventuali) nodi `N1`, `N2`
  - `N1` e `N2` sono detti rispettivamente figlio sinistro e figlio destro di `N`
  - `N` è detto nodo padre di `N1` e `N2`
- in un albero binario ci possono essere tre tipi di nodi:
  - il nodo radice, che non ha padre
  - i nodi foglia, che non hanno figli
  - i nodi intermedi, che hanno padre e almeno un figlio
- Una catena di nodi dalla radice a una foglia è detta ramo
  - il numero di nodi in un ramo è detto lunghezza del ramo
  - la massima lunghezza di un ramo è detta altezza dell'albero
  - l'altezza di un albero binario di N elementi è $h \in [\lceil log_2(N+1) \rceil, N]$
- Un albero binario di N elementi è bilanciato se la sua altezza è $h = \lceil log_2(N+1) \rceil$
  $\Longrightarrow$ tutti i rami hanno lunghezza `h` o `h-1`
- Un albero binario di N elementi è completo se la sua altezza è tale che $N = 2^h - 1$
  $\Longrightarrow$ tutti i rami hanno lunghezza `h`

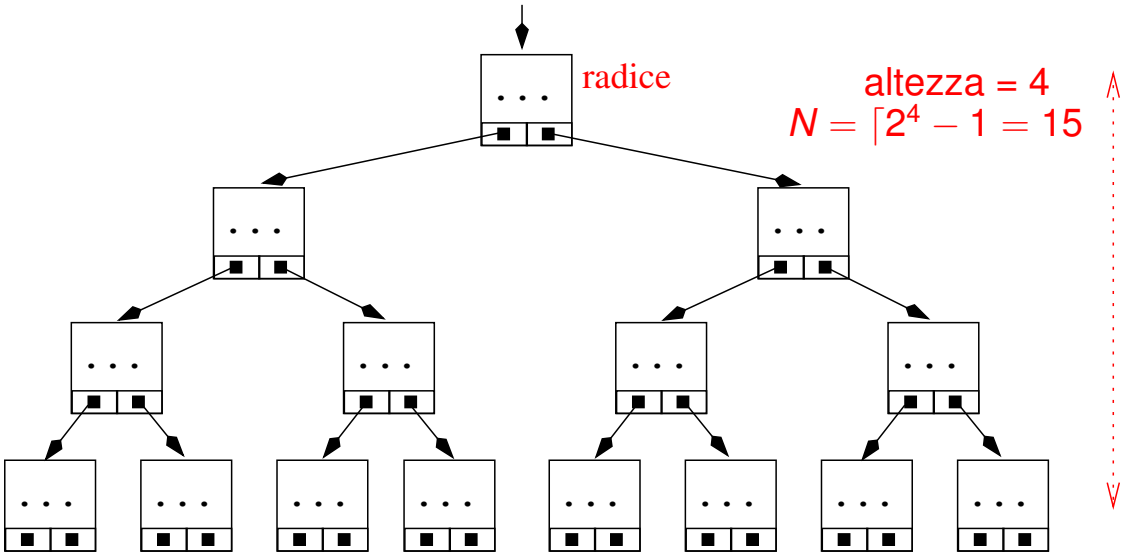
Esempio: albero binario bilanciato



Esempio: albero binario non bilanciato



Esempio: albero binario completo



Esempio: albero binario completo (2)

Albero di ricerca binaria

- Un albero di ricerca binaria è una struttura dati utile a mantenere dati ordinati.
- Assumiamo una relazione di ordine totale di precedenza "$\preceq$" tra gli elementi `T`
  - Es: ordine numerico, ordine alfabetico del campo "cognome", ecc.
- Un albero binario è un albero di ricerca binaria se ogni nodo `N` dell'albero verifica la seguente proprietà:
  - tutti i nodi del sottoalbero di sinistra precedono strettamente `N`
  - tutti i nodi del sottoalbero di destra sono preceduti da `N`

  (è possibile invertire lo "strettamente" tra sinistra e destra)

Nota: in alcuni casi non è previsto che ci possano essere due valori uguali nel valore valutato dalla relazione di precedenza (valore chiave)

Albero di ricerca binaria

- Un albero di ricerca binaria è una struttura dati utile a mantenere dati ordinati.
- Assumiamo una relazione di ordine totale di precedenza "$\preceq$" tra gli elementi `T`
  - Es: ordine numerico, ordine alfabetico del campo "cognome", ecc.
- Un albero binario è un albero di ricerca binaria se ogni nodo `N` dell'albero verifica la seguente proprietà:
  - tutti i nodi del sottoalbero di sinistra precedono strettamente `N`
  - tutti i nodi del sottoalbero di destra sono preceduti da `N`

  (è possibile invertire lo "strettamente" tra sinistra e destra)

Nota: in alcuni casi non è previsto che ci possano essere due valori uguali nel valore valutato dalla relazione di precedenza (valore chiave)

Albero di ricerca binaria

- Un albero di ricerca binaria è una struttura dati utile a mantenere dati ordinati.
- Assumiamo una relazione di ordine totale di precedenza "$\preceq$" tra gli elementi `T`
  - Es: ordine numerico, ordine alfabetico del campo "cognome", ecc.
- Un albero binario è un albero di ricerca binaria se ogni nodo `N` dell'albero verifica la seguente proprietà:
  - tutti i nodi del sottoalbero di sinistra precedono strettamente `N`
  - tutti i nodi del sottoalbero di destra sono preceduti da `N`

  (è possibile invertire lo "strettamente" tra sinistra e destra)

Nota: in alcuni casi non è previsto che ci possano essere due valori uguali nel valore valutato dalla relazione di precedenza (valore chiave)

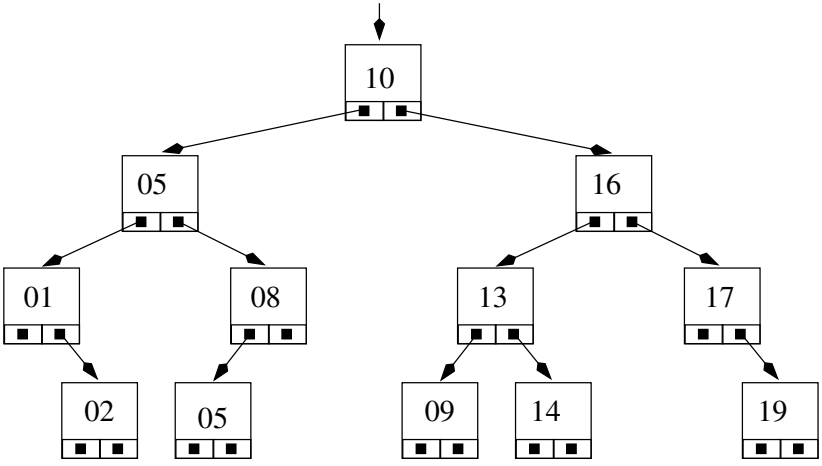
Albero di ricerca binaria

- Un albero di ricerca binaria è una struttura dati utile a mantenere dati ordinati.
- Assumiamo una relazione di ordine totale di precedenza "$\preceq$" tra gli elementi `T`
  - Es: ordine numerico, ordine alfabetico del campo "cognome", ecc.
- Un albero binario è un albero di ricerca binaria se ogni nodo `N` dell'albero verifica la seguente proprietà:
  - tutti i nodi del sottoalbero di sinistra precedono strettamente `N`
  - tutti i nodi del sottoalbero di destra sono preceduti da `N`

  (è possibile invertire lo "strettamente" tra sinistra e destra)

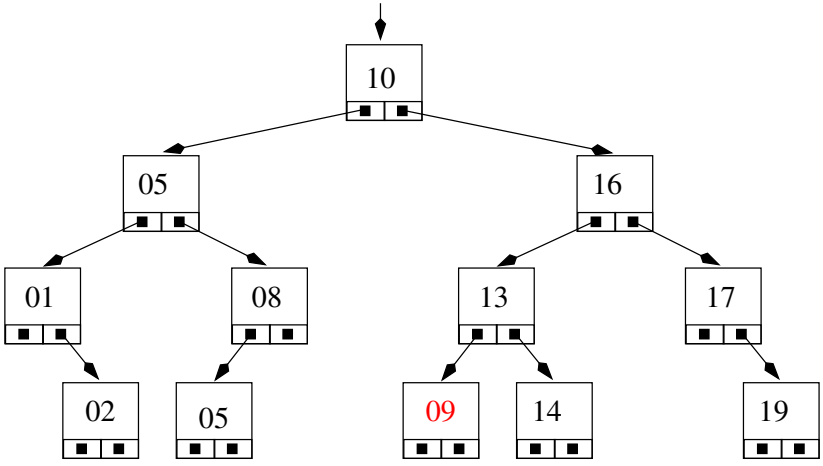
Nota: in alcuni casi non è previsto che ci possano essere due valori uguali nel valore valutato dalla relazione di precedenza (valore chiave)

Esempio: albero di ricerca binaria



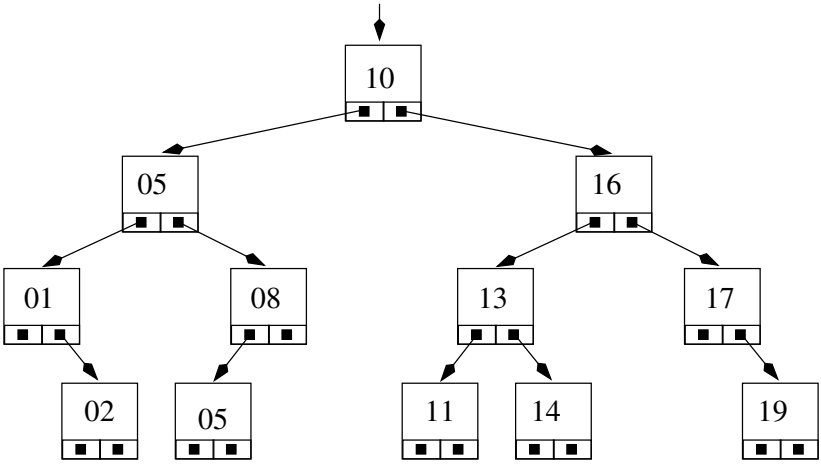
- Questo è un albero di ricerca binaria?
- No, 09 non può stare nel sottoalbero di destra di 10

Esempio: albero di ricerca binaria



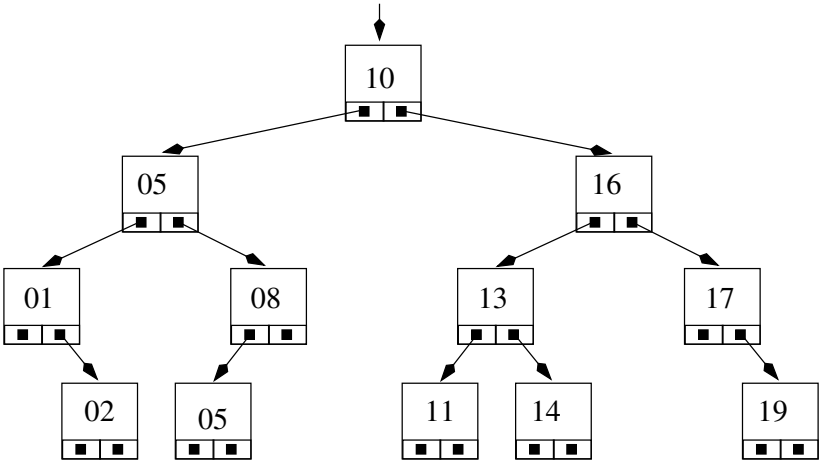
- Questo è un albero di ricerca binaria?
- No, 09 non può stare nel sottoalbero di destra di 10

Esempio: albero di ricerca binaria



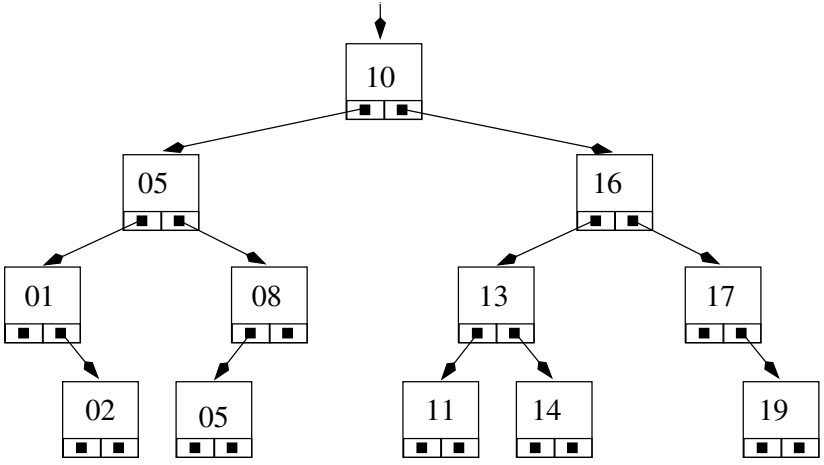
- Questo è un albero di ricerca binaria?
- Sì

Esempio: albero di ricerca binaria



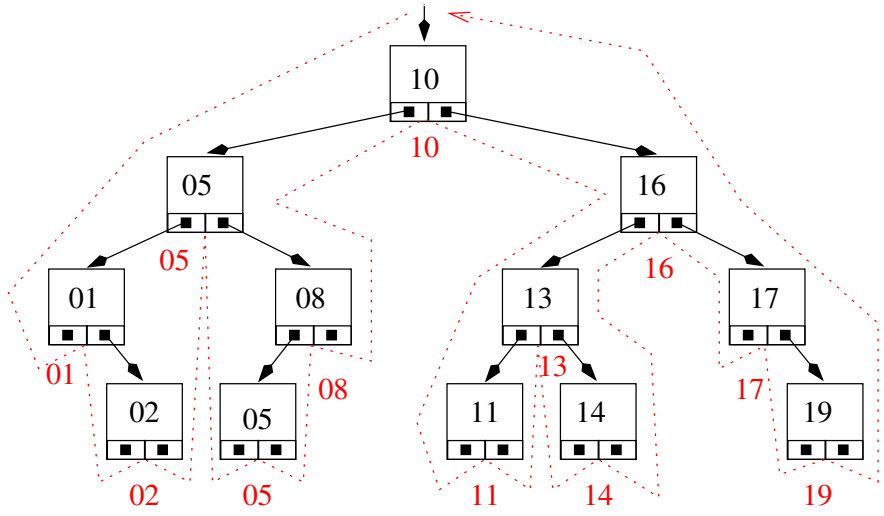
- Questo è un albero di ricerca binaria?
- Sì

Esempio: Visita ordinata di un albero di ricerca binaria



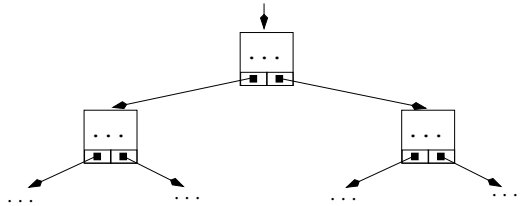
- Visita: $01, 02, 05, 05, 08, 10, 11, 13.14, 16, 17, 19 \Longrightarrow$ ordinati!

Esempio: Visita ordinata di un albero di ricerca binaria



- Visita: $01, 02, 05, 05, 08, 10, 11, 13.14, 16, 17, 19 \Longrightarrow$ ordinati!

Implementazione di un albero di ricerca binaria

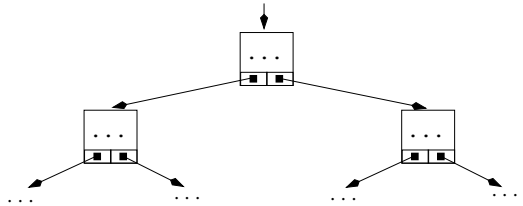


- Dati: un albero di ricerca binaria `t`
  - `t` punta al primo elemento inserito nell'albero (inizialmente NULL)
  - albero vuoto: `t==NULL`
  - albero pieno: `out of memory`

  $\Longrightarrow$ numero di elementi contenuti nell'albero limitato solo dalla memoria

N.B.: allocati solo gli `n` nodi necessari a contenere gli elementi

Implementazione di un albero di ricerca binaria

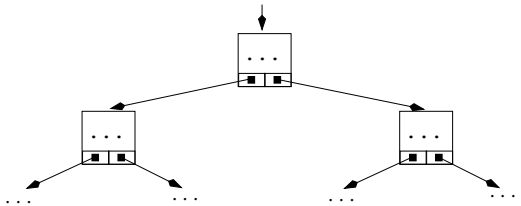


- Dati: un albero di ricerca binaria `t`
  - `t` punta al primo elemento inserito nell'albero (inizialmente NULL)
  - albero vuoto: `t==NULL`
  - albero pieno: `out of memory`

  $\Longrightarrow$ numero di elementi contenuti nell'albero limitato solo dalla memoria

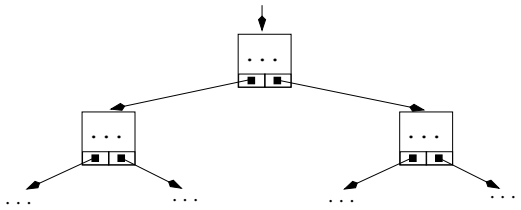
N.B.: allocati solo gli `n` nodi necessari a contenere gli elementi

Implementazione di un albero di ricerca binaria II



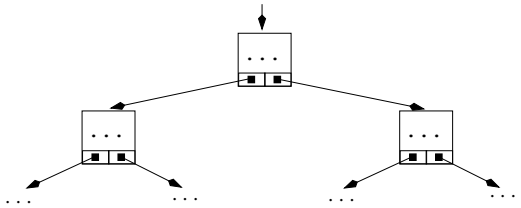
- Funzionalità:
  - `init`: pone `t=NULL`
  - `search` (cerca un elemento `val` in `t`):
    1. se `t == NULL`, restituisce `NULL`
    2. se `val == t->value`, restituisce `t`
    3. se `val < t->value`, cerca ricorsivamente in `t->left`
    4. se `val > t->value`, cerca ricorsivamente in `t->right`
  - ...

Implementazione di un albero di ricerca binaria III



- Funzionalità:
  - ...
  - `insert` (inserisce un elemento `val` in `t`):
    1. se `t` è vuoto, `t == NULL`:
       - crea un nuovo nodo per il puntatore `tmp`
       - pone `tmp->value=val`, `tmp->left=NULL`, `tmp->right=NULL`,
       - pone `t=tmp`
    2. se `val < t->value`, inserisci ricorsivamente in `t->left`
    3. se `val >=t->value`, inserisci ricorsivamente in `t->right`
  - ...

Implementazione di un albero di ricerca binaria III



- Funzionalità:
  - ...
  - `print` (stampa in modo ordinato l'albero `t`): Se l'albero non è vuoto
    - stampa ricorsivamente il sottoalbero sinistro `t->left`
    - stampa il contenuto del nodo puntato da `t`: `t->value`
    - stampa ricorsivamente il sottoalbero destro `t->right`
  - `deinit`: se l'albero non è vuoto:
    - applica ricorsivemente `deinit` ai sottoalberi sinistro `t->left` e destro `t->right`
    - applica `delete` al nodo puntato da `t`
  - [`remove` non analizzata qui]

Esempi su alberi di ricerca binaria su interi

- albero di interi:
  - `TREE/tree.h`
  - `TREE/tree.cc`
  - `TREE/tree_main.cc`
- variante della precedente:
  - `TREE/tree1.h`
  - `TREE/tree1.cc`
  - `TREE/tree1_main.cc`

Esempi su alberi di ricerca binaria su interi

- albero di interi:
  - `TREE/tree.h`
  - `TREE/tree.cc`
  - `TREE/tree_main.cc`
- variante della precedente:
  - `TREE/tree1.h`
  - `TREE/tree1.cc`
  - `TREE/tree1_main.cc`

Esempi su alberi di ricerca binaria su tipi generici

- albero di qualsiasi tipo si voglia:

  `MODULAR_TREE/tree.h`
  `MODULAR_TREE/tree.cc`
  `MODULAR_TREE/tree_main.cc`

Esercizi proposti

Vedere file `ESERCIZI_PROPOSTI.txt`

Outline

Implementazione di un albero binario tramite array

- Dati: un array `v` di `dim` elementi di tipo `t`
  - un (sotto)albero è dato da un puntatore a `v` e un indice `i`

    ```
    struct tree {  T * v; int i; };
    ```
  - `v` allocato dinamicamente
  - l'elemento radice è in `v[0]`
  - se un elemento è in posizione `v[i]`, i suoi due figli sono in posizione `v[2*i+1]` e `v[2*i+2]`
  - necessaria una nozione ausiliaria di "elemento vuoto"
- Funzionalità: come nell'implementazione precedente, cambia solo la nozione di figlio sinistro/destro

N.B.: allocati `dim` nodi $\Longrightarrow$ efficace solo se ben bilanciato

Implementazione di un albero binario tramite array

- Dati: un array `v` di `dim` elementi di tipo `t`
  - un (sotto)albero è dato da un puntatore a `v` e un indice `i`

    ```
    struct tree {  T * v; int i; };
    ```
  - `v` allocato dinamicamente
  - l'elemento radice è in `v[0]`
  - se un elemento è in posizione `v[i]`, i suoi due figli sono in posizione `v[2*i+1]` e `v[2*i+2]`
  - necessaria una nozione ausiliaria di "elemento vuoto"
- Funzionalità: come nell'implementazione precedente, cambia solo la nozione di figlio sinistro/destro

N.B.: allocati `dim` nodi $\Longrightarrow$ efficace solo se ben bilanciato

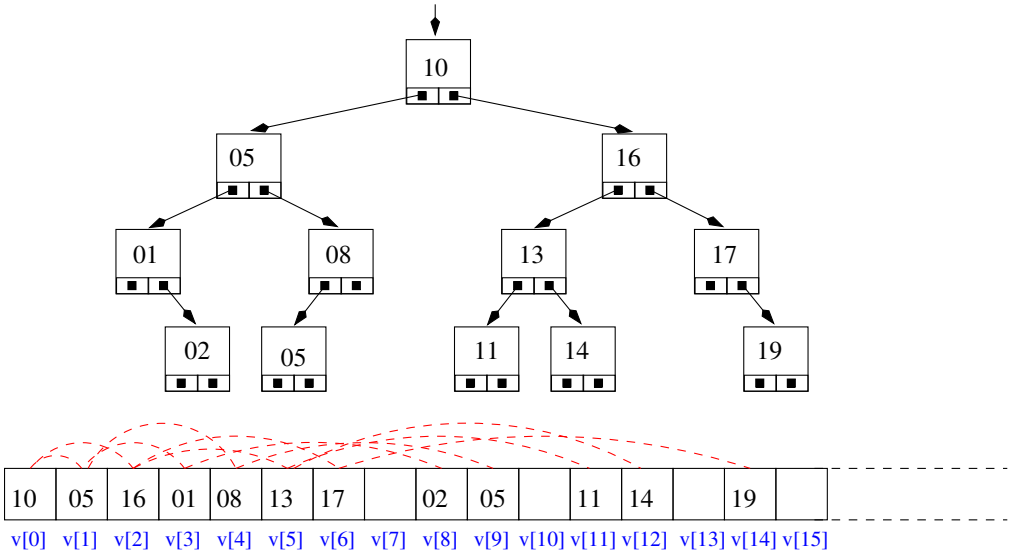
Implementazione di un albero binario tramite array

- Dati: un array `v` di `dim` elementi di tipo `t`
  - un (sotto)albero è dato da un puntatore a `v` e un indice `i`

    ```
    struct tree {  T * v; int i; };
    ```
  - `v` allocato dinamicamente
  - l'elemento radice è in `v[0]`
  - se un elemento è in posizione `v[i]`, i suoi due figli sono in posizione `v[2*i+1]` e `v[2*i+2]`
  - necessaria una nozione ausiliaria di "elemento vuoto"
- Funzionalità: come nell'implementazione precedente, cambia solo la nozione di figlio sinistro/destro

N.B.: allocati `dim` nodi $\Longrightarrow$ efficace solo se ben bilanciato

Implementazione di un albero binario tramite array II



Esempi su alberi di interi

- albero di interi:
  - `TREE_ARRAY/tree.h`
  - `TREE_ARRAY/tree.cc`
  - `TREE_ARRAY/tree_main.cc`

N.B. Il “main” e gli header delle funzioni identici a quelli in `TREE` $\Longrightarrow$ Tipo di Dato Astratto

Outline

Outline

Esempio di uso di Pile: Calcolatrice RPN

- Il metodo di calcolo Reverse Polish Notation (RPN) funziona postponendo l'operatore ai due operandi

  `34 * 3` $\Longrightarrow$ `34 3 *`

- permette di effettuare complicate concatenazioni di conti senza usare parentesi:

  `(34 * 3)/(31 - 5) + (21+3)/(24-12)`

  $\Longrightarrow$

  `34 3 * 31 5 - / 21 3 + 24 12 - / +`

- Una calcolatrice RPN funziona le modo seguente:
  - se viene immesso un operando, lo mette in uno stack di operandi
  - se viene immesso un operatore (binario) `op`:
    1. vengono prelevati dallo stack gli ultimi due operandi `op1` e `op2`
    2. viene applicato l'operatore `op` a `op2` e `op1`
    3. il risultato viene ri-immesso nello stack

Esempio di uso di Pile: Calcolatrice RPN

- Il metodo di calcolo Reverse Polish Notation (RPN) funziona postponendo l'operatore ai due operandi

  `34 * 3` $\Longrightarrow$ `34 3 *`

- permette di effettuare complicate concatenazioni di conti senza usare parentesi:

  `(34 * 3)/(31 - 5) + (21+3)/(24-12)`

  $\Longrightarrow$

  `34 3 * 31 5 - / 21 3 + 24 12 - / +`

- Una calcolatrice RPN funziona le modo seguente:
  - se viene immesso un operando, lo mette in uno stack di operandi
  - se viene immesso un operatore (binario) `op`:
    1. vengono prelevati dallo stack gli ultimi due operandi `op1` e `op2`
    2. viene applicato l'operatore `op` a `op2` e `op1`
    3. il risultato viene ri-immesso nello stack

Esempio di uso di Pile: Calcolatrice RPN

- Il metodo di calcolo Reverse Polish Notation (RPN) funziona postponendo l'operatore ai due operandi

  `34 * 3` $\Longrightarrow$ `34 3 *`

- permette di effettuare complicate concatenazioni di conti senza usare parentesi:

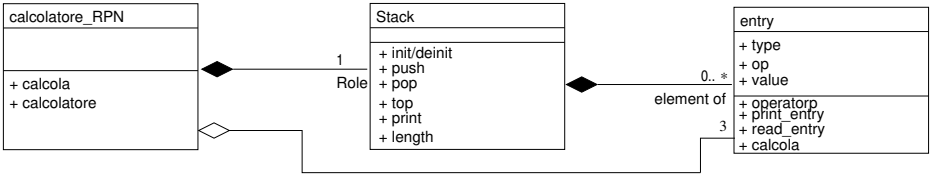
  `(34 * 3)/(31 - 5) + (21+3)/(24-12)`

  $\Longrightarrow$

  `34 3 * 31 5 - / 21 3 + 24 12 - / +`

- Una calcolatrice RPN funziona le modo seguente:
  - se viene immesso un operando, lo mette in uno stack di operandi
  - se viene immesso un operatore (binario) `op`:
    1. vengono prelevati dallo stack gli ultimi due operandi `op1` e `op2`
    2. viene applicato l'operatore `op` a `op2` e `op1`
    3. il risultato viene ri-immesso nello stack

Implementazione della Calcolatrice RPN



gestione delle entry:

{ CALC_RPN/entry.h
  CALC_RPN/entry.cc }

pila di entry:

{ CALC_RPN/stack.h
  CALC_RPN/stack.cc }

calcolatore RPN:

{ CALC_RPN/calcolatore_rpn.h
  CALC_RPN/calcolatore_rpn.cc }

main:

{ CALC_RPN/main.cc }

Outline

Esempio di uso di Code: Coda a priorità

- Una coda a priorità di messaggi è una struttura dati in cui
  - ogni messaggio arrivato ha una priorità in [0..10]
  - i messaggi vengono estratti in ordine di priorità, $0 \Longrightarrow 1 \Longrightarrow 2$ ...
  - a parità di priorità vengono estratti in modo FIFO
- Realizzabile con un array di code, una per ogni livello di priorità :
  - un messaggio di priorità *i* viene inserito nella coda *i*-esima
  - l'estrazione avviene a partire dalla coda 0-sima: se vuota si passa alla successiva, ecc.
- Esempio: l'accettazione al Pronto Soccorso di un ospedale

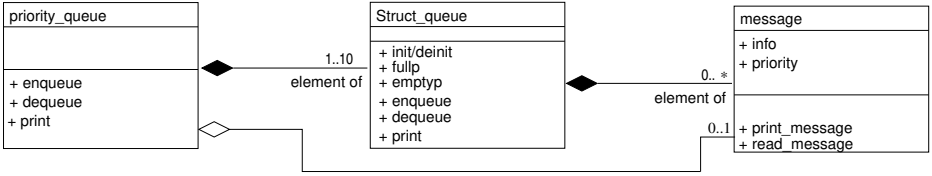
Esempio di uso di Code: Coda a priorità

- Una coda a priorità di messaggi è una struttura dati in cui
  - ogni messaggio arrivato ha una priorità in [0..10]
  - i messaggi vengono estratti in ordine di priorità, $0 \Longrightarrow 1 \Longrightarrow 2$ ...
  - a parità di priorità vengono estratti in modo FIFO
- Realizzabile con un array di code, una per ogni livello di priorità :
  - un messaggio di priorità *i* viene inserito nella coda *i*-esima
  - l'estrazione avviene a partire dalla coda 0-sima: se vuota si passa alla successiva, ecc.
- Esempio: l'accettazione al Pronto Soccorso di un ospedale

Esempio di uso di Code: Coda a priorità

- Una coda a priorità di messaggi è una struttura dati in cui
  - ogni messaggio arrivato ha una priorità in [0..10]
  - i messaggi vengono estratti in ordine di priorità, $0 \Longrightarrow 1 \Longrightarrow 2$ ...
  - a parità di priorità vengono estratti in modo FIFO
- Realizzabile con un array di code, una per ogni livello di priorità :
  - un messaggio di priorità *i* viene inserito nella coda *i*-esima
  - l'estrazione avviene a partire dalla coda 0-sima: se vuota si passa alla successiva, ecc.
- Esempio: l'accettazione al Pronto Soccorso di un ospedale

Implementazione della Coda a Priorità



gestione delle entità “message”:

{ `CODA_PRIORITA/message.h` `CODA_PRIORITA/message.cc` }

coda di (puntatori a) messaggi:

{ `CODA_PRIORITA/struct_queue.h` `CODA_PRIORITA/struct_queue.cc` }

coda a priorità di (puntatori a) messaggi:

{ `CODA_PRIORITA/prio_queue.h` `CODA_PRIORITA/prio_queue.cc` }

main:

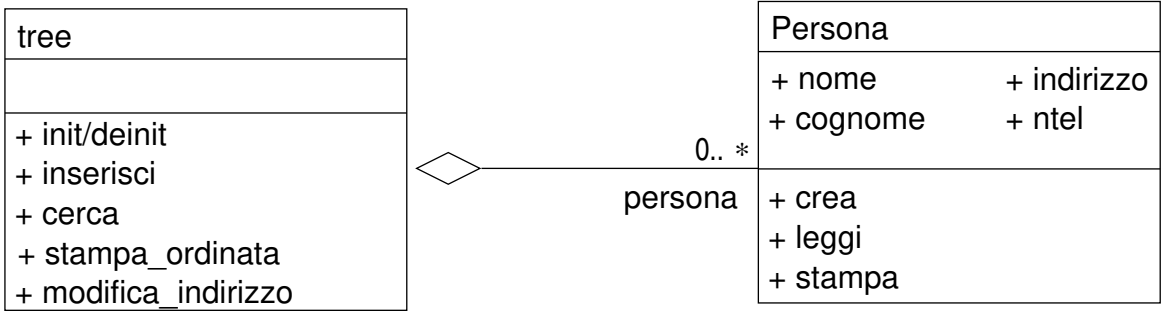
{ `CODA_PRIORITA/prio_queue_main.cc` }

Outline

Esempio di uso di Alberi: una Rubbrica

- Una Rubbrica è una lista di (dati di) persone, ordinata con qualche criterio (es per cognome)
  - realizzata come un albero di ricerca binaria di TDA “persona”
  - albero ordinato per il campo “cognome”
  - importante utilizzare puntatori a persona:
    $\Longrightarrow$ ogni copia/passaggio è il passaggio solo di un puntatore
  - per semplicità, non è possibile eliminare una persona dalla rubbrica

Implementazione della Rubbrica



gestione del TDA “persona”:

{ RUBBRICA/persona.h
RUBBRICA/persona.cc }

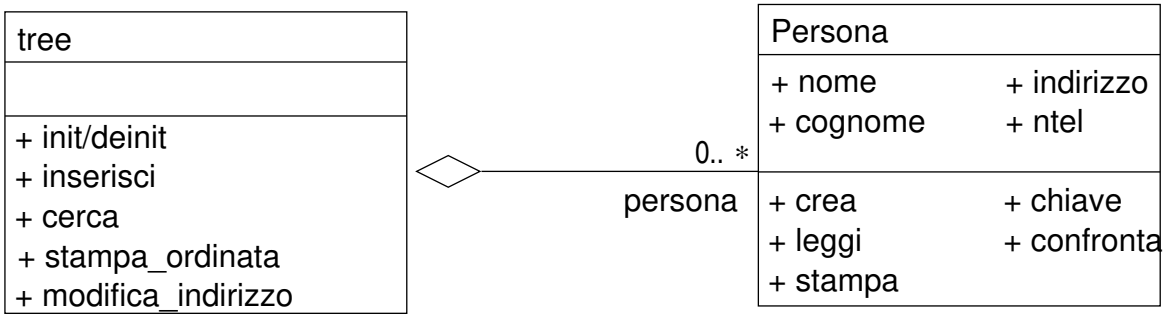
albero di puntatori a persona, ordinato per cognome:

{ RUBBRICA/tree.h
RUBBRICA/tree.cc }

main:

{ RUBBRICA/main.cc }

Implementazione della Rubbrica (“modulare”)



gestione del TDA “persona”:

{ `MODULAR_RUBBRICA/persona.h` `MODULAR_RUBBRICA/persona.cc` }

albero di puntatori a persona, ordinato per cognome:

{ `MODULAR_RUBBRICA/tree.h` `MODULAR_RUBBRICA/tree.cc` }

main:

{ `MODULAR_RUBBRICA/main.cc` }

Esercizi proposti

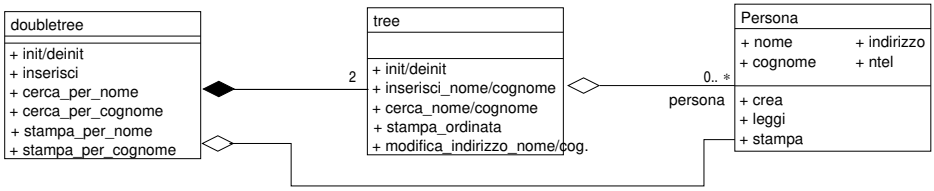
Vedere file `ESERCIZI_PROPOSTI.txt`

Outline

Esempio di uso di Alberi: una Rubbrica Doppia

- Una Rubbrica Doppia è una lista di (dati di) persone, ordinata con un doppio criterio (es per cognome e per nome)
  - realizzata come una coppia di alberi di ricerca binaria di puntatori al TDA “persona”
    - un albero ordinato per il campo “cognome”
    - un albero ordinato per il campo “nome”
  - importante utilizzare puntatori a persona:
    - $\Longrightarrow$ ogni copia/passaggio è il passaggio solo di un puntatore
    - ogni TDA persona è condiviso tra i due alberi
  - non è possibile eliminare una persona dalla rubbrica doppia

Implementazione della Rubbrica Doppia



gestione del TDA “persona”:

{ RUBBRICA_DOPPIA/persona.h
RUBBRICA_DOPPIA/persona.cc }

albero di puntatori a persona, ordinato per cognome o per nome:

{ RUBBRICA_DOPPIA/tree.h
RUBBRICA_DOPPIA/tree.cc }

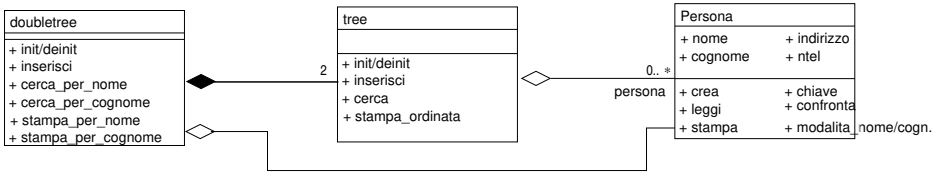
doppio albero di puntatori a persona:

{ RUBBRICA_DOPPIA/doubletree.h
RUBBRICA_DOPPIA/doubletree.cc }

main:

{ RUBBRICA_DOPPIA/main.cc }

Implementazione della Rubbrica Doppia (“modulare”)



gestione del TDA “persona”:

{ MODULAR_RUBBRICA_DOPPIA/persona.h
MODULAR_RUBBRICA_DOPPIA/persona.cc }

albero di puntatori a persona, ordinato per cognome o per nome:

{ MODULAR_RUBBRICA_DOPPIA/tree.h
MODULAR_RUBBRICA_DOPPIA/tree.cc }

doppio albero di puntatori a persona:

{ MODULAR_RUBBRICA_DOPPIA/doubletree.h
MODULAR_RUBBRICA_DOPPIA/doubletree.cc }

main:

{ MODULAR_RUBBRICA_DOPPIA/main.cc }

Outline

Esempio di uso di Albero: Calcolatrice Standard

- Un'espressione aritmetica standard utilizza operatori (binari) infissi e parentesi:
  `((34 * 3)/(31 - 5) + (21+3)/(24-12))`
- Un'espressione aritmetica standard può essere rappresentata da un albero binario di cui
  - le foglie contengano numeri
  - gli altri nodi contengano operatori
- Un'espressione viene letta e creata ricorsivamente come segue:
  - se viene letto un numero, viene creata un'espressione foglia
  - altrimenti (viene letta una parentesi aperta):
    - viene creato un nodo intermedio
    - viene letta e creata ricorsivamente la prima espressione, e assegnata al figlio sinistro
    - viene letto l'operatore, ed inserito nel nodo
    - viene letta e creata ricorsivamente la seconda espressione, e assegnata al figlio destro
    - viene letta la parentesi chiusa

Esempio di uso di Albero: Calcolatrice Standard

- Un'espressione aritmetica standard utilizza operatori (binari) infissi e parentesi:
  `((34 * 3)/(31 - 5) + (21+3)/(24-12))`
- Un'espressione aritmetica standard può essere rappresentata da un albero binario di cui
  - le foglie contengano numeri
  - gli altri nodi contengano operatori
- Un'espressione viene letta e creata ricorsivamente come segue:
  - se viene letto un numero, viene creata un'espressione foglia
  - altrimenti (viene letta una parentesi aperta):
    - viene creato un nodo intermedio
    - viene letta e creata ricorsivamente la prima espressione, e assegnata al figlio sinistro
    - viene letto l'operatore, ed inserito nel nodo
    - viene letta e creata ricorsivamente la seconda espressione, e assegnata al figlio destro
    - viene letta la parentesi chiusa

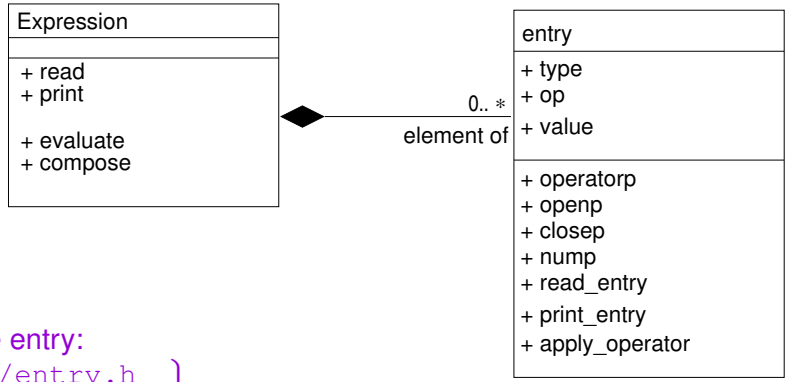
Esempio di uso di Albero: Calcolatrice Standard

- Un'espressione aritmetica standard utilizza operatori (binari) infissi e parentesi:
  `((34 * 3)/(31 - 5) + (21+3)/(24-12))`
- Un'espressione aritmetica standard può essere rappresentata da un albero binario di cui
  - le foglie contengano numeri
  - gli altri nodi contengano operatori
- Un'espressione viene letta e creata ricorsivamente come segue:
  - se viene letto un numero, viene creata un'espressione foglia
  - altrimenti (viene letta una parentesi aperta):
    - viene creato un nodo intermedio
    - viene letta e creata ricorsivamente la prima espressione, e assegnata al figlio sinistro
    - viene letto l'operatore, ed inserito nel nodo
    - viene letta e creata ricorsivamente la seconda espressione, e assegnata al figlio destro
    - viene letta la parentesi chiusa

Esempio di uso di Albero: Calcolatrice Standard II

- Un'espressione viene valutata ricorsivamente come segue:
  - se è una foglia, viene restituito il suo valore
  - altrimenti, si valutano ricorsivamente i due sottoalberi, e vi si applica l'operatore del nodo

Implementazione della Calcolatrice Standard



gestione delle entry:

{ CALC_STD/entry.h
  CALC_STD/entry.cc }

TDA espressione aritmetica:

{ CALC_STD/espressione.h
  CALC_STD/espressione.cc }

main:

{ CALC_STD/main.cc }

Esercizi proposti

Vedere file `ESERCIZI_PROPOSTI.txt`