Moduli

Come organizziamo il codice?
Con la **modularizzazione**: dividere per gestire la complessità.

DEF.

Il **modulo** è unità di programma che mette a disposizione risorse e servizi computazionali (dati, funzioni, …).
Il modulo è costituito da:

- –Una Interfaccia
    - definisce le risorse ed i servizi (astrazioni) messi a disposizione dei “clienti” (programma o altri moduli)
    - completamente visibile ai clienti
- Una sezione implementativa (body)
    - implementa le risorse ed i servizi esportati
    - completamente occultato
- Fondamentale nella realizzazione dei concetti di:
    - Astrazione
    - Information Hiding
- Riuso di componenti già costruite e verificate
    - – Ad esempio: una volta definite delle funzioni che consentono di risolvere sotto-problemi di utilità generale, come è possibile riusarle nella soluzione di altri problemi ?
- Un modulo può usare altri moduli
- Un modulo può essere compilato indipendentemente dal modulo (o programma) che lo usa.

In C non esiste un apposito costrutto per realizzare un modulo; di solito un modulo coincide con un file:

- Per esportare le risorse definite in un file (modulo), il C fornisce un particolare tipo di file, chiamato header file (estensione .h)

DEF.

Un **header file** rappresenta l’interfaccia di un modulo verso gli altri moduli

- Per accedere alle risorse messe a disposizione da un modulo bisogna includere il suo header file

In C il modulo si presenta come una “libreria” di funzioni per l’information hiding

- nessun effetto collaterale
- nessuna variabile globale
- funzioni di servizio nascoste

DEF.

Progetto: **Makefile** e comando make
Tutti gli ambienti di programmazione consentono di costruire un progetto

- il comando make: compilazione e collegamento dei vari moduli che compongono il progetto

In ambiente UNIX (Linux): Makefile e comando make

- Il Makefile è costituito da specifiche del tipo:
    - target_file: dipendenze_da_file

comandi

- esecuzione della specifica: make target_file

Astrazione

DEF.

L'**astrazione** è un procedimento mentale che consente di evidenziare le caratteristiche pregnanti per determinare l'obiettivo. Definire l'entità funzionali o dati che compongono un sistema. (esempio la codifica del nuovo tipo di dato esiste, ma non è necessario conoscerla per utilizzarlo.).
Ci sono vari tipi di astrazione:

- **Astrazione funzionale**: una funzione è totalmente definita ed usabile indipendentemente dal suo algoritmo.
- **Astrazione sui dati**: un dato e le sue operazioni, sono usabili a prescindere dall'implementazione
- **Astrazione sul controllo**: un meccanismo di controllo è definito ed usabile indipendentemente dalla modalità e dalle tecniche con cui è realizzato.

DEF.

Information Hiding:

- Occultamento dell'informazione ovvero la realizzazione di alti livelli di astrazione attraverso la definizione di strutture capaci di mettere a disposizione risorse servizi occultandone i dettagli implementative.
- Il programmatore può concentrarsi sul nuovo oggetto senza preoccuparsi dei dettagli implementativi.

Realizzazione di alti livelli di astrazione: la combinazione di elementi per creare un'entità più grande.

Argomenti sulla linea di comando

Il **main** è una funzione invocata dal sistema operativo che prende in input due parametri:

- un array stringhe che contiene le stringhe inserite dall'utente da linea di comando char*argv[].
    - In argv[0] c'è il nome del programma.
- Un intero argc Che contiene il numero di stringhe inserite dall'utente a linea di comando.

DEF.

Una **variabile** può essere:

1. **Locale** ed è visibile solo all'interno della funzione in cui è dichiarata
    - È possibile dichiarare dati anche in blocchi più interni ...
    - Tali variabili sono dette automatiche, perché vengono allocate in memoria a tempo di esecuzione (dell'istruzione dichiarativa) e deallocate al termine del blocco in cui sono dichiarate
2. **Globale**, dichiarata esternamente alla funzione ed è è visibile a tutte le funzioni la cui definizione segue la dichiarazione della variabile nel file sorgente

    - tali variabili sono dette statiche, perché la loro allocazione in memoria avviene all'atto del caricamento del programma (e la loro deallocazione al termine del programma).

DEF.

Ci sono tre aspetti importanti di una **dichiarazione**:

- **Scope**: parte del programma in cui è attiva una dichiarazione (dice quando può essere usato un identificatore)
- **Visibilità**: dice quali variabili sono accessibili in una determinata parte del programma (non sempre coincide con lo scope …)
- **Durata**: periodo durante il quale una variabile è allocata in memoria
    - Le variabili globali sono allocate in un'area di memoria fissa
    - Le variabili locali sono allocate in un'area di memoria detta stack
    - Le variabili dinamiche (puntatori) sono allocate in un'area di memoria detta heap

Dichiarazione di funzioni e variabili **Static**: servono a realizzare l'information Hiding. La funzione statica è visibile solo modulo, non viene esportato, è riservata al modulo e ne modifichiamo lo scoope. Non è visibile al cliente o ad altri moduli. Vale la stessa cosa per le variabili static.

Testing di un Programma

DEF.

Il **testing** è una forma di debugging cioè esercitare il programma con dati di testing per verificare che il risultato sia quello atteso.

DEF.

Una **teoria fondamentale della computazione** dice che non esiste un algoritmo capace di dimostrare la correttezza di un qualunque programma.

Provare tutte le configurazioni di dati possibili è impraticabile quindi si utilizzeranno classi di dati di test cioè una test suite.

DEF.

Debugging: individuazione e correzione del difetto che ha causato il malfunzionamento.

Per automatizzare il teste possiamo usare:

- Un file "*test_input.txt*" per leggere i dati di **input**
- Un file "*test_output.txt*" per scrivere i dati di **output**
- Un file "test_oracle.txt" con il **risultato dell'esecuzione**

Come testare il programma sull'intera teste suite.

Usiamo due file aggiuntivi:

- Un file d'input "*test_suite.txt*" Che indica per ogni test case, il nome del teste case ed il numero per esempio degli elementi dell'array.
- Un file di output "*result.txt*" che memorizza l'esito di ogni test (PASS/FAIL).

Come scrivere il programma di test

- Aprire il file *test_suite.tx*t in lettura e leggere le varie righe del file finché non si raggiunge la fine del file (EOF)
- Per ogni riga del file *test_suite.txt*:

- Usare il primo elemento della riga (l'identificatore del caso di suite per esempio TC1) Per costruire le stringhe per i nomi dei file di input, oracolo ed output
- Eseguire il codice del programma di testa usando come numero di elementi il secondo elemento della riga del file *test_suite*.txt e come nomi dei file di input, oracolo ed output quelli costruiti in precedenza
- Scrivere una riga il file *result.txt* in base al confronto tra esempio un vettore di output e un vettore contenente l'oracolo e per prima riga ad esempio se ha funzionato TC1 PASS altrimenti TC1 FAIL

Tipi di dati astratti

DEF.

L'**astrazione di dati** permette di ampliare i tipi di dati disponibili attraverso l'introduzione sia di nuovi tipi di dati che di nuovi operatori.

DEF.

Un **tipo di dato** viene definito specificandone il dominio e le operazione su di esso applicabili, in aggiunta alle operazione elementari del C.

Si effettua:

DEF.

La **specifica**: descrive l'astrazione dati e il modo in cui può essere utilizzata attraverso i suoi operatori. Se la precondizione è vera ed il programma è eseguito allora la post condizione è vera. Essa si divide in due tipi di specifiche:

- **Specifica sintattica**
  - I nomi del tipo di dati di riferimento e degli eventuali tipi di dati usati (già definiti)
  - I nomi delle operazioni del tipo di dati di riferimento
  - I tipi di dati di input e di output per ogni operatore
- **Specifica semantica**
  - L'insieme dei valori associati al tipo di dati di riferimento
  - La funzione associata ad ogni nome di operatore, specificata dalle seguenti condizioni:
    - •i) precondizione: definita sui valori dei dati di input definisce quando l'operatore è applicabile
    - •ii) postcondizione: definita sui valori dei dati di output e di input, stabilisce la relazione tra argomenti e risultato

Nell'analisi e specifica dei dati è buona norma utilizzare un **dizionario dei dati** da arricchire durante le varie fasi del ciclo di vita del programma. Una tabella il cui schema è: identificatore, tipo, descrizione. Dove la descrizione serve specificare meglio e a descrivere il contesto in cui il dato viene usato.

La **realizzazione**: come il nuovo dato e i nuovi operatori vengono ricondotti ai dati e agli operatori già disponibili.

Il tipo astratto lista

DEF.

Una **lista** è una sequenza di elementi di un determinato tipo, in cui è possibile aggiungere o togliere elementi.

- Per aggiungere o togliere elementi occorre specificare la posizione relativa all'interno della sequenza nella quale il nuovo elemento va aggiunto o nella quale il vecchio elemento va tolto.

- A differenza dell'array, che è una struttura a dimensione fissa dove è possibile accedere direttamente ad ogni elemento specificandone l'indice, la lista è a dimensione variabile e si può accedere direttamente solo al primo elemento della lista.

- Per accedere ad un generico elemento, occorre scandire sequenzialmente gli elementi della lista.

Implementazione del tipo astratto lista: Lista concatenata

- Ogni elemento di una lista concatenata è un record con un campo puntatore che serve da collegamento per il record successivo.

- Si accede alla struttura attraverso il puntatore al primo record.

- Il campo puntatore dell'ultimo record contiene il valore NULL

Differenza tra lista ed array

Una lista concatenata è più flessibile di un array; è più facile inserire o cancellare un elemento, utilizzando solo la memoria strettamente necessaria

C'è uno svantaggio rispetto agli array: si perde la capacità di accedere in modo diretto agli elementi della lista, usando l'indice dell'array.

Inserire un elemento in una lista concatenata

Il modo più semplice per inserire un nuovo elemento in una lista concatenata L è inserire l'elemento in un nuovo nodo da aggiungere in testa alla lista

1. Per prima cosa si crea il nuovo nodo, poi si aggiunge il collegamento con il record iniziale della lista
2. Poi si aggiorna L facendolo puntare al nodo appena aggiunto

Dichiarazione del tipo di un nodo

- Per usare una lista concatenata serve una struttura che rappresenti i nodi
- La struttura conterrà i dati necessari (un intero nel seguente esempio) ed un puntatore al prossimo elemento della lista:

```
struct node {
```

```
  item value;                  /* data stored in the node */
  struct node *next;      /* pointer to the next node */
};
```

- Nodebdefinisce una struttura che contiene un campo che punta ad un altra struttura di tipo node

Il passo successivo è quello di **dichiarare il tipo lista**

- typedef struct node *list;
- una variabile di tipo lista punterà al primo nodo della lista:
  list l = NULL;
    - Assegnare a l il valore NULL indica che la lista è inizialmente vuota

Creare un nodo della lista

- Per creare un nodo ci serve un puntatore temporaneo che punti al nodo:

struct node *new_node;

- Possiamo usare malloc per allocare la memoria necessaria e salvare l'indirizzo in new_node:
  new_node = malloc(sizeof(struct node));
- new_node adesso punta ad un blocco di memoria che contiene la struttura di tipo node:

Il tipo astratto stack

DEF.

Una **pila** (spesso chiamata anche stack) è una sequenza di elementi di un determinato tipo, in cui è possibile aggiungere o togliere elementi, uno alla volta, esclusivamente da un unico lato (top dello stack).

- Questo significa che la sequenza viene gestita con la modalità detta **LIFO** (**Last-in-first-out**) cioè l'**ultimo elemento inserito** nella sequenza sarà il **primo ad essere eliminato**.
- La pila è una **struttura dati lineare**, **omogenea**, a **dimensione variabile** in cui si può **accedere direttamente solo al primo elemento** della lista.
- Non è possibile accedere ad un elemento diverso dal primo, cioè quello che è stato inserito per ultimo, se non dopo aver eliminato tutti gli elementi che lo precedono (cioè che sono stati inseriti dopo).

Implementazione semplice di stack con array

- Lo stack è implementato come un **puntatore ad una struct c_stack** che contiene due elementi:
    - Un array di MAXSTACK elementi
    - Un intero che indica la posizione del top dello stack

- Quando lo stack si riempie, non è possibile eseguire l'operazione push …

Implementazione dello stack senza array a dimensione fissa

Per evitare che lo stack abbia una capienza massima:

- Bisogna usare l'allocazione dinamica della memoria e due costanti
    - La prima **STARTSIZE** definisce la **dimensione iniziale** dello **stack**
    - La seconda **ADDSIZE** definisce di **quanto allargare** lo **stack** nel caso in cui si riempia

- Questo significa che ci occorre anche una variabile **size** che ci dica **quanti elementi può contenere** lo **stack** in **ogni momento**

Implementazione dello stack con liste collegate

- Il tipo **stack** è definito come un **puntatore ad una struct** che contiene:
    - Un **intero numelem** che indica il **numero di elementi** dello **stack**
    - Un **puntatore top ad una struct nodo** (come per la lista)

Implementazione dello stack basato sul modulo lista

Il tipo stack è definito come un puntatore ad una struct che contiene un elemento **top** di **tipo list**.

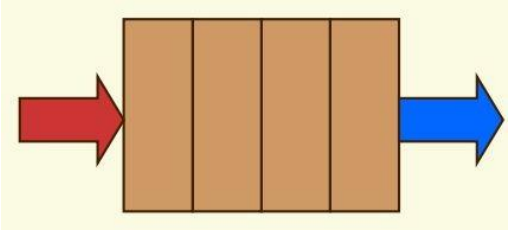
- Non serve più nemmeno l'intero numelem che indica il numero di elementi dello stack …

- Anche se abbiamo un solo elemento nella struct, continuiamo a definire il tipo stack come puntatore a struct c_stack per non cambiare la definizione nell'header file …

Il tipo astratto coda

DEF.

Una **coda** (spesso chiamata anche queue) è una sequenza di elementi di un determinato tipo, in cui gli elementi si aggiungono da un lato (**tail - primo elemento a sinistra**) e si tolgono dall'altro lato (**head - primo elemento a destra**).



- Questo significa che la sequenza viene gestita con la modalità detta **FIFO** (**First-in-first-out**) cioè il **primo elemento inserito** nella sequenza sarà il **primo ad essere eliminato**.

- La coda è una struttura dati **lineare** a **dimensione variabile** in cui si può **accedere direttamente solo alla testa** (head) della lista.
- Non è possibile accedere ad un elemento diverso da head, se non dopo aver eliminato tutti gli elementi che lo precedono (cioè quelli inseriti prima).

Il tipo astratto BTree

DEF.

Grafo

- Il GRAFO è **una struttura** dati alla quale si possono ricondurre strutture più semplici: LISTE ed ALBERI
- Un grafo orientato G e' una coppia <N,A> dove N è un insieme finito non vuoto (insieme di nodi) e A ⊆ NxN è un insieme finito di coppie ordinate di nodi, detti archi (o spigoli o linee).
    - •Se < ui, uj > ∈ A (Con u nodo) nel grafo vi è un arco da ui ad uj

DEF.

Un **albero è una collezione di nodi e di archi**, per cui

- ogni nodo, eccetto uno, detto radice, ha un solo predecessore e 0 o più successori, la radice non ha predecessori
- esiste un unico cammino dalla radice ad ogni altro nodo
- ogni nodo contiene un valore di un qualche tipo, detto etichetta

Alcuni concetti

- **Grado di un nodo**: numero di figli del nodo
- **Cammino**: sequenza di nodi <n0, n1, ..., nk> dove il nodo ni è padre del nodo ni+1, per i compreso tra 0 e k con 0 compreso e k non compreso (0=<i<k)
    - – La lunghezza del cammino è k
    - – Dato un nodo, esiste un unico cammino dalla radice dell'albero al nodo Livello di un nodo: lunghezza del cammino dalla radice al nodo
- ***DEF.*** **ricorsiva**: il livello della radice è 0, il livello di un nodo non radice è 1 + il livello del padre
- **Altezza dell'albero**: la lunghezza del più lungo cammino nell'albero – Parte dalla radice e termina in una foglia

Proprietà

- Un albero e' un **grafo aciclico**, in cui per ogni nodo esiste un **solo arco entrante** (tranne che per la radice che non ne ha nessuno)
- Se esiste un cammino che va da un nodo u ad un altro nodo v, **tale cammino è unico**
- In un albero **esiste un solo cammino** che va dalla radice a qualunque altro nodo

- Tutti nodi di un albero t (tranne la radice) possono essere ripartiti in insiemi ciascuno dei quali individua un albero: dato un nodo u, i suoi discendenti costituiscono un albero detto **sottoalbero di radice u**

DEF.

Alberi binari

Sono particolari **alberi n-ari** con caratteristiche molto importanti

- Ogni nodo può avere al più due figli
  - sottoalbero sinistro e sottoalbero destro
- Definizione ricorsiva:
  - **PASSO BASE**: albero binario è vuoto
  - **PASSO RICORSIVO**: è una terna (s, r, d), dove r è un nodo (la radice), s e d sono alberi binari
- Alberi binari semplificati
  - Costruttori bottom-up
  - Operatori di selezione
  - Operatori di visita

Implementazione dell'albero binario (BTree)

Realizzazione più diffusa: **struttura a puntatori con nodi doppiamente concatenati**

- Ogni nodo è una struttura con 3 componenti:
  - Puntatore alla radice del sottoalbero sinistro
  - Puntatore alla radice del sottoalbero destro
  - Etichetta (useremo il tipo generico ITEM per questo campo
- Un albero binario è definito come puntatore ad un nodo:
  - Se l'albero binario è vuoto, puntatore nullo
  - Se l'albero binario non è vuoto, puntatore al nodo radice

Dichiarazione del tipo nodo

Per usare una lista concatenata serve una struttura che rappresenti i nodi.
La struttura conterrà i dati necessari (un intero nel seguente esempio) ed un puntatore al prossimo elemento della lista:

```
  struct node {
/* etichetta del nodo*/
    item value;
 /* puntatore al sottoalbero sinistro*/
    struct node *left;
/* puntatore al sottoalbero destro*/
     struct node *right;
  };
```

Dichiarazione del tipo Btree

- Il passo successivo è quello di dichiarare il tipo Btree
    - *typedef struct node *Btree;*
- Una variabile di tipo Btree punterà nodo radice dell'albero
- Assegnare a T il valore NULL indica che l'albero è inizialmente vuoto
    - *Btree T = NULL;*

Costruire un albero binario

- Un albero binario viene costruito in maniera **bottom-up**
- Man mano che costruiamo l'albero, creiamo dei nuovi nodi da aggiungere come nodo radice
- I passi per creare un nodo sono:
    - *1. Allocare la memoria necessaria*
    - *2. Memorizzare i dati nel nodo*
    - *3. Collegare il sottoalbero sinistro e il sottoalbero destro, già costruiti in precedenza*

Il tipo astratto BST

- Utilizzato per la realizzazione di insiemi ordinati
- Operazioni efficienti di
    - *Ricerca*
    - *Inserimento*
    - *Cancellazione*

DEF.

Se l'albero non è vuoto:

- Ogni elemento del sottoalbero di sinistra precede (<) la radice
- Ogni elemento del sottoalbero di destra segue (>) la radice
- I sottoalberi sinistro e destro sono alberi di ricerca binaria

Alberi perfettamente bilanciati e alberi D bilanciati

Alberi perfettamente bilanciati
e alberi D bilanciati
•Le operazioni sull'albero di ricerca binaria hanno complessità logaritmica se l'albero è (perfettamente) bilanciato
–In un albero bilanciato tutti i nodi interni hanno entrambi i sottoalberi e le foglie sono a livello massimo
–Se l'albero ha n nodi l'altezza dell'albero è log2 n
•Un albero di ricerca binaria si dice D bilanciato se per ogni nodo accade che la differenza (in valore assoluto) tra le altezze dei suoi due sottoalberi è minore o uguale a D
–Si può dimostrare che l'altezza dell'albero è D + log2 n

Albero AVL

Per D = 1 si parla di alberi AVL
–Dal nome dei suoi ideatori (Adel'son, Vel'skii e Landis)
•Per prevenire il non bilanciamento ad ogni nodo bisogna aggiungere un indicatore che può assumere i seguenti valori
–1, se l'altezza del sottoalbero sinistro è maggiore (di 1) dell'altezza del sottoalbero destro
– 0, se l'altezza del sottoalbero sinistro è uguale all'altezza del sottoalbero destro
–+1, se l'altezza del sottoalbero sinistro è minore (di 1) dell'altezza del sottoalbero destro.

Ribilanciamento di un albero

Un inserimento di una foglia può provocare uno sbilanciamento dell'albero
–Per almeno uno dei nodi l'indicatore non rispetta più uno dei tre stati precedenti
•In tal caso bisogna ribilanciare l'albero con operazioni di rotazione (semplice o doppia) agendo sul nodo x a profondità massima che presenta un non bilanciamento
–Tale nodo viene detto nodo critico e si trova sul percorso che va dalla radice al nodo foglia inserito
•Considerazioni simili si possono fare anche per la rimozione di un nodo …

Heap

DEF.

Un albero quasi perfettamente bilanciato di altezza h è un albero perfettamente bilanciato fino a livello h-1
Un **heap è un albero binario** con le seguenti proprietà

- Proprietà strutturale: quasi perfettamente bilanciato e le foglie a livello h sono tutte addossate a sinistra
- Proprietà di ordinamento: ogni nodo v ha la caratteristica che l'informazione ad esso associata è la più grande tra tutte le informazioni presenti nel sottoalbero che ha v come radice

Usato per realizzare code a priorità

- Le operazioni sono inserimento di un elemento e rimozione del max

Tipo di dato astratto Code a priorità

DEF.

Struttura dati i cui elementi sono coppie (key, value) dette entry, dove key e value appartengono a due insiemi qualsiasi K e V
Le **entry vengono inserite in ordine qualsiasi**, ma **estratte in ordine di priorità secondo il valore della key**

- **Ordinamento**: sull'insieme delle chiavi è definita una relazione d'ordine ≤
- **Priorità**: per convenzione, una entry E1=(k1, v1) ha priorità su E2=(k2, v2) se e solo se k2 ≤ k1

ADT *Lista:* Specifica sintattica

- **Tipo di riferimento**: list
- **Tipi usati**: item, boolean
- **Operatori**
  - newList() → list
  - emptyList(list) → boolean
  - consList(item, list) → list
  - tailList(list) → list
  - getFirst(list) → item

ADT *Lista:* Specifica semantica

- **Tipo di riferimento list**
  - list è l'insieme delle sequenze *L=a1,a2,…,an* di tipo ***item***
  - L'insieme list contiene inoltre un elemento ***nil*** che rappresenta la lista vuota (priva di elementi)
- **Operatori**
  - newList() → l
    - Post: l = nil
  - emptyList(l) → b
    - Post: se l=nil allora b = true altrimenti b = false
  - consList(e, l) → l'
    - Post: l = <a1, a2, … an> AND l' = <e, a1, a2, …, an>
  - tailList(l) → l'
    - Pre: l = <a1, a2, …, an> n>0
    - Post: l' = <a2, …, an>
  - getFirst(l) → e
    - Pre: l = <a1, a2, …, an> n>0
    - Post: e = a1

Aggiungiamo operatori alla lista

- **Specifica sintattica**
  - sizeList(list) → integer
  - posItem(list, item) → integer
  - searchItem(list, item) → boolean
  - reverseList(list) → list
  - removeItem(list, item) → list
  - getItem(list, integer) → item
  - insertList(list, integer, item) → list
  - removeList(list, integer) → list

- **Specifica semantica**
  - sizeList(l) → n
    - Post: l = <a1, a2, …, an> AND n ≥ 0
  - searchItem(l, e) → b
    - Post: se e è contenuto in l allora b = true, se no b = false
  - posItem(l, e) → p
    - Post: se e è contenuto in l allora p è la posizione della prima occorrenza di e in l, altrimenti p = -1
  - reverseList(l) → l'
    - Post: l = <a1, a2, …, an> AND l' = <an, ..., a2, a1>
  - removeItem(l, e) → l'
    - Post: se e è contenuto il l, allora l' si ottiene da l eliminando la prima occorrenza di e in l, altrimenti l' = l
  - insertList(l, p, e) → l'
    - Pre: pos >=0 AND sizeList(l) ≥ p
      // assumiamo 0 come prima posizione
    - Post: l' si ottiene da l inserendo e in posizione p
  - removeList(l, p) → l'
    - Pre: pos >=0 AND sizeList(l) > p
    - Post: l' si ottiene da l eliminando l'elemento in posizione p
  - getItem(l, pos) → e
    - Pre: pos >= 0 AND sizeList(l) > pos

ADT *Queue:* Specifica sintattica

- **Tipo di riferimento**: queue
- **Tipi usati**: item, boolean
- **Operatori**
  - newQueue() → queue
  - emptyQueue(queue) → boolean
  - enqueue(item, queue) → queue
  - dequeue(queue) → item

ADT *Queue:* Specifica semantica

- **Tipo di riferimento queue**
  - queue è l'insieme delle sequenze *S=a1,a2,…,an* di tipo ***item***
  - L'insieme queue contiene inoltre un elemento ***nil*** che rappresenta la coda vuota (priva di elementi)

ADT *Queue:* Specifica semantica

- **Operatori**
  - newQueue() → q
    - Post: q = nil
  - emptyQueue(q) → b
    - Post: se q=nil allora b = true altrimenti b = false
  - enqueue(e, q) → q'
    - Post: se q = nil allora q' = <e> altrimenti
      se q = $<a_1, a_2, \ldots a_n>$ con n > 0 allora q' = $<a_1, a_2, \ldots a_n, e>$
  - dequeue(q) → a
    - Pre: q = $<a_1, a_2, \ldots, a_{n-1}, a_n>$ n>0 ($q \neq nil$)
    - Post: a = a_1 e l'elemento a_1 viene rimosso da q

Gli Alberi Binari

SPECIFICA SINTATTICA

TIPI: *ALBEROBIN, BOOLEAN, NODO, ITEM*

OPERATORI:

newBtree	:	( ) → ALBEROBIN
emptyBtree	:	(ALBEROBIN) → BOOLEAN
getRoot	:	(ALBEROBIN) → NODO
figlioSX	:	(ALBEROBIN) → ALBEROBIN
figlioDX	:	(ALBEROBIN) → ALBEROBIN
consBtree	:	(ITEM, ALBEROBIN, ALBEROBIN) → ALBEROBIN

SPECIFICA SEMANTICA

TIPI:

ALBEROBIN = insieme degli alberi binari, dove:
∧ ∈ ALBEROBIN (albero vuoto)
se N ∈ NODO, T1 e T2 ∈ ALBEROBIN
allora <N, T1, T2> ∈ ALBEROBIN
BOOLEAN = {vero, falso}

NODO è un qualsiasi insieme non vuoto

ITEM è un qualsiasi insieme non vuoto

SPECIFICA SEMANTICA

OPERATORI:

newBtree ( ) = T
pre:
post: T = ∧

emptyBtree (T) = v
pre:
post: se T è vuoto, allora v = vero, altrimenti v = falso

getRoot (T) = N'
pre: T = <N, Tsx, Tdx> non è l'albero vuoto
post: N = N'

figlioSX (T) = T'
pre: T = <N, Tsx, Tdx> non è l'albero vuoto
post: T' = Tsx

SPECIFICA SEMANTICA

OPERATORI:

figlioDX (T) = T'
pre: T = <N, Tsx, Tdx> non è l'albero vuoto
post: T' = Tdx

consBtree (elem, T1, T2) = T'
pre: elem ≠ NULLITEM
post: T' = <N, T1, T2>
N è un nodo con etichetta elem

Code a priorità

SPECIFICA SINTATTICA

TIPI: *PRIORITYQUEUE, BOOLEAN, ITEM*

OPERATORI:

newPQ : ( ) → PRIORITYQUEUE
emptyPQ : (PRIORITYQUEUE) → BOOLEAN
getMax : (PRIORITYQUEUE) → ITEM
deleteMax : (PRIORITYQUEUE) → PRIORITYQUEUE
insertPQ : (PRIORITYQUEUE, ITEM) → PRIORITYQUEUE

SPECIFICA SEMANTICA

TIPI:

PRIORITYQUEUE= insieme delle code a priorità, dove:
$\wedge \in$ *PRIORITYQUEUE* (coda vuota)

BOOLEAN = {vero, falso}

ITEM = (K x V) è l'insieme delle coppie (k, v) con $k \epsilon K$ *e* $v \epsilon V$

K è un insieme qualsiasi non vuoto sul quale è definita una relazione d'ordine $\leq$

V è un insieme qualsiasi non vuoto

SPECIFICA SEMANTICA

OPERATORI:

newPQ ( ) = PQ
pre:
post: PQ = $\wedge$

emptyPQ (PQ) = v
pre:
post: se PQ è vuota, allora v = vero, altrimenti v = falso

getMax (PQ) = elem
pre: PQ non è vuota
post: elem è la entry con la massima priorità fra quelle contenute in PQ

SPECIFICA SEMANTICA

OPERATORI:

deleteMax (PQ) = PQ'
pre: PQ non è vuota
post: PQ' contiene tutte le entry di PQ tranne quella con massima priorità

insertPQ (PQ, elem) = Q'
pre:
post: PQ' contiene *elem* e tutte le entry contenute in PQ