Insiemi

Roberto Amadini

roberto.amadini@unibo.it

Abstract data types

- Abbiamo visto in precedenza il tipo di dato **lista**

- Una lista è un esempio di tipo di dato **astratto**
  - **ADT**, *Abstract Data Type*

- Gli ADT sono **entità matematiche** definite da:
  - Un insieme di **elementi**
    - Es. numeri interi, complessi, stringhe o oggetti più complessi...
  - Un insieme di **operazioni** su tali elementi
    - Es. somma, cancellazione, lettura, ricerca, …

Abstract data types

- Es. di operazioni definite su **liste**:
  - Creazione lista
  - Lettura/scrittura elemento
  - Aggiunta/eliminazione elemento
  - Ricerca elemento
  - Lunghezza lista

- Un ADT definisce elementi e operazioni in modo “astratto” o “**logico**”, senza preoccuparsi della sua **implementazione** effettiva

Abstract data types

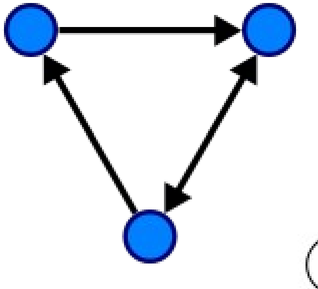
- Spesso il concetto di ADT viene confuso con quello di **struttura dati**:
  - Un ADT è un **concetto logico**, definisce in modo **astratto** elementi e operazioni per un tipo di dato
    - **Cosa** deve fare un certo tipo, quali operazioni possibili
    - Punto di vista dell'**utente** che lo utilizza
  - Una struttura dati è una **rappresentazione concreta** di un tipo di dato
    - **Come** è implementato il tipo di dato e le operazioni
    - Punto di vista dello **sviluppatore** che lo implementa

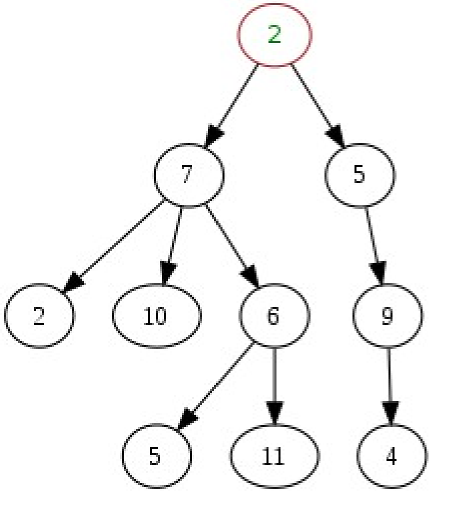
Abstract data types

- Ad es. la lista può essere **implementata** da diverse strutture dati:
  - *Singly* linked list
  - *Doubly* linked list
    - Ad es. la classe `LinkedList` di Java
  - *Circular* linked list
  - *Array*
    - Ad es. la classe `ArrayList` di Java

Abstract data types

- Esistono ovviamente molti altri ADT:
  - Pila
    - Operazioni principali `push/pop`, logica **LIFO**
  - Coda
    - Operazioni principali `push/pop`, logica **FIFO**
  - Grafi
  - Alberi
    - Casi particolari di grafi
  - **Insiemi**
  - …





Insiemi

- Un insieme è un ADT in cui:
  - **Non** esistono elementi **ripetuti**
  - Gli elementi **non** sono necessariamente **ordinati**
    - Non ha senso, in generale, parlare di *i-esimo* elemento

- Esempi:
  - {“foo”, “bar”}
  - {x}
  - {2, -4, 0}
  - { }
  - {{1,2}, {3,4}, {5}}

Operazioni su insiemi

- Le **operazioni** più comuni su un insieme sono:
  - Unione: $\mathbf{A} \cup \mathbf{B} = \{x \mid x \in A \textbf{ or } x \in B\}$
  - Intersezione: $\mathbf{A} \cap \mathbf{B} = \{x \mid x \in A \textbf{ and } x \in B\}$
  - Differenza: $\mathbf{A \setminus B} = \{x \mid x \in A \textbf{ and not } x \in B\}$
  - Sottoinsieme: $\mathbf{A} \subseteq \mathbf{B}$ sse $\forall x(\textbf{ if } x \in A \textbf{ then } x \in B)$
  - Cardinalità: $\mathbf{|A|}$
  - Controllo se insieme vuoto: $\mathbf{A = \{\ \}\ ?}$
  - Controllo appartenenza elemento: $\mathbf{x \in A\ ?}$
  - …

Tipi di insiemi

- Esistono diversi tipi di insieme, es.
  - **Statici**: una volta costruiti sono immutabili, posso solo "*interrogarli*"
    - isEmpty(S), contains(S, x), size(S), isSubset(S, T), …
  - **Dinamici**: posso *aggiungere/rimuovere* elementi in qualsiasi momento
    - add(S, x), remove(S, x), union(S, T), inters(S, T), …
  - **Ordinati**: se posso definire una *relazione d'ordine* sugli elementi dell'insieme
    - min(S), max(S), …

Implementare insiemi

- Un insieme può essere **implementato** da diverse **strutture dati**, es.
  - (Foreste di) alberi
  - Tabelle hash
  - Liste concatenate
  - Array
    - Bit-vectors
- Java offre l'interfaccia **`Set`**, implementata da diverse classi, es. **`HashSet`** o **`TreeSet`**
  - Vedremo più avanti

Insiemi di interi

- Supponiamo di dover rappresentare insiemi **dinamici** di **interi**
- Possiamo definire un ordinamento naturale, attraverso la relazione di **ordine totale ≤**
  - **≤** è relazione d'ordine **parziale** su $\mathbb{Z}$ perchè è:
    - **Riflessiva**: $(\forall x \in \mathbb{Z})\ x \leq x$
    - **Antisimmetrica:** $(\forall x,y \in \mathbb{Z})\ x \leq y \wedge y \leq x \rightarrow x = y$
    - **Transitiva:** $(\forall x,y,z \in \mathbb{Z})\ x \leq y \wedge y \leq z \rightarrow x \leq z$
  - **≤** è in particolare una relazione d'ordine **totale** su $\mathbb{Z}$ perché $\forall x,y \in \mathbb{Z}$ si ha che $x \leq y$ oppure $y \leq x$ oppure $x = y$

Insiemi di interi e arrays

- Come **implementare** un insieme di interi?

- La cosa più semplice è utilizzare un **array** di interi
  - Es. {2, 7, -3, 0} → [2, 7, -3, 0]
  - Attenzione! Devo controllare che non ci siano elementi ripetuti
    - Es. [100, 4, **3, 3**] non va bene

- Limitazioni:
  - L’array è una struttura dati **statica**
  - Anche per insiemi statici, memorizzare un **intervallo** di interi **a..b** richiede un array di **b-a+1** elementi
    - Es. 5..10000 → [5, 6, 7, …, 9998, 9999, 10000]   (10000-5+1 = 9996 int.)

Insiemi di interi e bit-vectors

- Si può far meglio sfruttando la **codifica binaria** dei valori in memoria
- Es. un **`int`** in Java occupa **32** bit: posso rappresentare $\mathbf{2^{32}}$ configurazioni diverse
  - Tante quante sono i *sottoinsiemi* di un insieme di 32 elementi $\mathbf{\{a_1, a_2, \ldots, a_{32}\}}$
    - Cioè, la cardinalità del suo *insieme delle parti*
- Posso usare i bit di una variabile per codificare la **funzione caratteristica** di un insieme

Insiemi di interi e bit-vectors

- La **funzione caratteristica f_A** identifica un insieme A. **f_A** è tale che, per ogni elemento x:
  - **$f_A(x) = 1$**, se $x \in A$
  - **$f_A(x) = 0$**, altrimenti
- Es. se A = {2, -3, -1} allora $f_A(2) = 1$, $f_A(-2) = 0$

- Posso implementarla usando **sequenze di bit**:
  - Se l'*i-esimo bit* è a **1**, *i* appartiene all'insieme
  - Se l'*i-esimo bit* è a **0**, *i non* appartiene all'insieme

Insiemi di interi e bit-vectors

- Es. A = {2,6,3,1}. Invece di un array [2,6,3,1] posso usare un intero di *almeno* 6 bit:

N	...	8	7	6	5	4	3	2	1	0
0	...	0	0	**1**	0	0	**1**	**1**	**1**	0

- Con N bit posso rappresentare insiemi fino a N elementi
  - Non necessariamente l’insieme 1..N
- Con un **array** di **M** elementi, ognuno dei quali occupa **N** bit, posso rappresentare insiemi di **N*M** elementi → **Bit Vectors**
  - Es. array di 10 `int` → insieme di 10 * 32 = 320 elementi

Insiemi di interi e bit-vectors

- Un bit-vector permette operazioni molto efficienti tra insiemi utilizzando operazioni **bitwise**
  - Es. {0, 3, 6} ∩ {1, 3} = **{3}**

N	...	8	7	6	5	4	3	2	1	0	
0	…	0	0	**1**	0	0	**1**	0	0	**1**	{0,3,6}
0	…	0	0	0	0	0	**1**	0	**1**	0	{1,3}
0	...	0	0	0	0	0	**1**	0	0	0	**{3}**

  - Per l'intersezione si usa l'operatore di **AND bitwise** (operatore **&** in Java)
    - Da non confondere con **&&**

Insiemi di interi e bit-vectors

- Analogamente per l'unione si usa l'operatore **OR bitwise** (operatore **|** in Java)
  - Es. {0, 3, 6} ∪ {1, 3} = {0, 1, 3, 6}

N	...	8	7	6	5	4	3	2	1	0	
0	…	0	0	**1**	0	0	**1**	0	0	**1**	{0,3,6}
0	…	0	0	0	0	0	**1**	0	**1**	0	{1,3}
0	...	0	0	**1**	0	0	**1**	0	**1**	**1**	{0,1,3,6}

  - Esistono altri operatori bitwise (~, <<, >>, …)
    - https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html

Insiemi di interi e bit-vectors

- *Vantaggi*:
  - Risparmio **memoria**
  - Operazioni efficienti con **operatori bitwise**

- *Svantaggi*:
  - Struttura dati **statica**
  - Per codificare un intervallo **a..b** mi servono **b-a+1 bits** settati a 1. Ad es. intervallo 2..100:

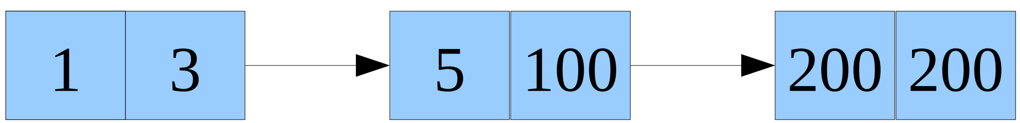
N	...	101	100	99	...	4	3	2	1	0
0	…	0	**1**	**1**	**...**	**1**	**1**	**1**	0	0

Liste di intervalli

- Un altro modo per codificare insiemi di interi è usare **liste ordinate** di **intervalli disgiunti**
- Ad es. l'insieme {<u>1, 2, 3</u>, <u>**5, 6, …, 100**</u>, 200} contiene l'intervallo **5..100**: basta memorizzare solo gli estremi **5** e **100** e non tutti i suoi elementi interni
  - La stessa cosa vale per 1, 2, 3 che di fatto corrisponde all'intervallo **1..3**
  - L'elemento 200 corrisponde all'intervallo **200..200**

Liste di intervalli

- In questo modo {1, 2, 3, 5, 7, …, 100, 200} si può rappresentare con una **lista di coppie,** dove ogni coppia codifica gli estremi (*bounds*) di un intervallo

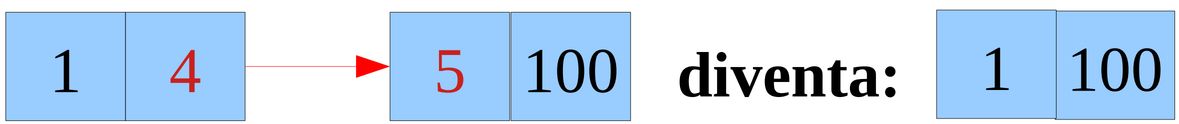


  - Anzichè 100 interi, ne memorizzo **6**

- **NOTA**: Le coppie sono **ordinate**: se **X** e **Y** sono intervalli consecutivi nella lista, allora **ub(X) < lb(Y) - 1**
  - lb = **lower** bound, ub = **upper** bound

Liste di intervalli

- Se ub(X) **=** lb(Y) – 1, allora possiamo **unire** gli intervalli X ed Y



- Esempi:
  - { } → lista vuota [ ]
  - {-2, -8, -3, 0, 1, -5} → [-8..-8, -5..-5, -3..-2, 0..1]
  - {1, 2, …, 1000000} → [1..1000000]
  - {1, 3, 5, 7, 9, 11} → [1..1, 3..3, 5..5, 7..7, 9..9]

Liste di intervalli

- *Vantaggi*:
  - Rappresentazione **compatta**
  - **Dinamici**
  - Molto utili se l'insieme è "*denso*"

- *Svantaggi*:
  - Operazioni meno efficienti di quelle **bitwise**
  - Poco utili se l'insieme è "*sparso*"

Implementazione con liste di intervalli

Roberto Amadini

roberto.amadini@unibo.it

Classe Range

- Cominciamo ad **implementare** gli insiemi di interi come liste ordinate di intervalli disgiunti
- Definiamo una classe **Range** che deve avere:
  - Due campi privati **`lower`** e **`upper`** di tipo **`int`** corrispondenti a lower e upper bound dell'intervallo
    - L'intervallo vuoto è codificato con **`lower=1`** e **`upper=0`**
- La classe Range ha 3 **costruttori**:
  - Senza parametri: crea intervallo **vuoto**
  - Con **1** parametro int **x**: crea intervallo **x..x**
  - Con **2** parametri int **x,y**: crea intervallo **x..y**.
    - Se **y < x** sollevare un eccezione **`IllegalArgumentException`**

Classe Range

- La classe **Range** deve avere anche i metodi:
  - **getLower** e **getUpper** che ritornano lower/upper bound
    - Se l'intervallo è vuoto, sollevare un **UnsupportedOperationException**
  - **setLower(x)** e **setUpper(x)** per settare lower/upper bounds
    - Se l'intervallo è vuoto, sollevare un **UnsupportedOperationException**
    - Se **setLower(x) con x > upper** oppure **setUpper(x) con x < lower**, sollevare un'eccezione di tipo **IllegalArgumentException**
  - **`size`** ritorna un valore **`long`** corrispondente dimensione dell'intervallo
    - Perchè **`long`**?
  - **`isEmpty`** ritorna `true` sse se l'intervallo è vuoto
  - **`contains(x)`** ritorna `true` sse `x` è nell'intervallo
  - **`toString()`** ritorna la stringa **lower`..`upper**
  - **`equals(r)`** ritorna true sse **this** è uguale all'intervallo **r**

Classe Range

- Implementare la classe **`Range`**, e una classe **`TestRange`** che testi le operazioni di **`Range`**
  - Cercare di raggiungere il **coverage** massimo
  - In particolare, usate blocchi **`try-catch`** per sollevare e gestire eccezioni nel main

Classe `RangeList`

- Ora definiamo una classe **`RangeList`** che implementa una lista concatenata di intervalli. Tale classe deve contenere:
  - Una classe **privata `RangeNode`**, che rappresenta un nodo della lista. **`RangeNode`** ha solo 2 campi:
    - **`range`** di tipo **`Range`** (l'elemento del nodo)
    - **`next`** di tipo **`RangeNode`** (il rif. al nodo successivo)
    - definire anche **costruttori** per RangeNode
  - Un campo privato **`head`** di tipo **`RangeNode`** che punta al **1°** elemento della lista

Classe `RangeList`

- **`RangeList`** ha 4 **costruttori**:
  - Senza parametri: crea lista vuota (cioè, **`head = null`**)
  - Con **`1`** parametro **`x`** di tipo **`int`**: crea lista di 1 elemento **`x..x`**
  - Con **`2`** parametri **`x,y`** di tipo **`int`**: crea lista di 1 elemento **`x..y`**
  - Con **`1`** parametro **`a`** di tipo **`int[]`**: crea lista contenente gli elementi dell'array a.
    - Es. se a = [2, 2, 4, 6, 1, 3, 6] la lista creata sarà [1..4, 6..6]
    - *Hint*: per **ordinare** un array si può usare **`Arrays.sort(A)`**

Classe `RangeList`

- `RangeList` ha un metodo **`toString()`** che ritorna:
  - La stringa **{ }** se l'insieme è vuoto
  - La stringa **{x}** se l'insieme ha solo l'elemento x
  - Altrimenti ritorna la stringa $\mathbf{\{S_1,S_2,\ldots,S_n\}}$ dove $\mathbf{S_i = X_i.toString()}$ e $\mathbf{X_i}$ è l'*i-esimo* intervallo della lista
    - Se Xi è un singoletto x..x stampare solo x
- Es. se la RangeList è [1..3, 5..5, 7..8] toString() ritorna la stringa **{1..3, 5, 7..8}**

Classe `RangeList`

- **`RangeList`** deve inoltre contenere i metodi:
  - **`isEmpty`** che che ritorna true sse se l’insieme è vuoto
  - **`size`** che ritorna la dimensione dell’insieme
  - **`min`** che ritorna il minimo degli elementi dell’insieme
  - **`max`** che ritorna il minimo degli elementi dell’insieme
    - Se invoco min o max sull’insieme **vuoto**, lanciare una **`UnsupportedOperationException`**
  - **`contains(x)`** che ritorna true sse x appartiene all’insieme
  - **`equals(x)`** che ritorna true sse this è uguale ad x
- Definire una classe **`TestRangeList`** che testi questi metodi, eccezioni comprese

Classe `RangeList2`

- Possiamo arricchire **`RangeList`**, definendo ad es.
  - L'insieme come lista **doppiamente concatenata**
    - **RangeNode** deve avere anche un campo **`prev`** che punta al nodo precedente
  - Un campo privato **`size`** di tipo **`int`** che tiene traccia della **cardinalità** dell'insieme
    - Deve essere **aggiornato** quando si modifica l'insieme
  - Un riferimento **`tail`** all'ultimo nodo
    - *Perchè?*
- *Esercizio:* Definire e testare una classe **`RangeList2`** che abbia queste caratteristiche e una classe **`TestRangeList2`** che la testi

Classe `RangeList2`

- Si possono poi aggiungere a **`RangeList2`** operazioni “classiche” su insiemi come:
  - Aggiunta di un elemento
  - Rimozione di un elemento
  - Unione
  - Intersezione
  - Complemento rispetto ad insieme universo U
  - Differenza insiemistica
  - Controllo se insiemi disgiunti

- Implementazione non banale
- *Esercizio*: implementare **2** metodi tra quelli sopra