PROGRAMMAZIONE AVANZATA

5 Liste e iteratori

- Rappresentazione di vettori
- Rappresentazione di liste
- Esempio: ricerca di un elemento nullo
- Iteratori
- Esempio: insiemi mutabili
- Ricerca di un elemento
- Rimozione di un elemento
- Inserimento di un elemento
- Stampa di insiemi
- Stampa di insiemi con funzione amica
- Esercizi

Rappresentazione di vettori

Dentro `std::vector`

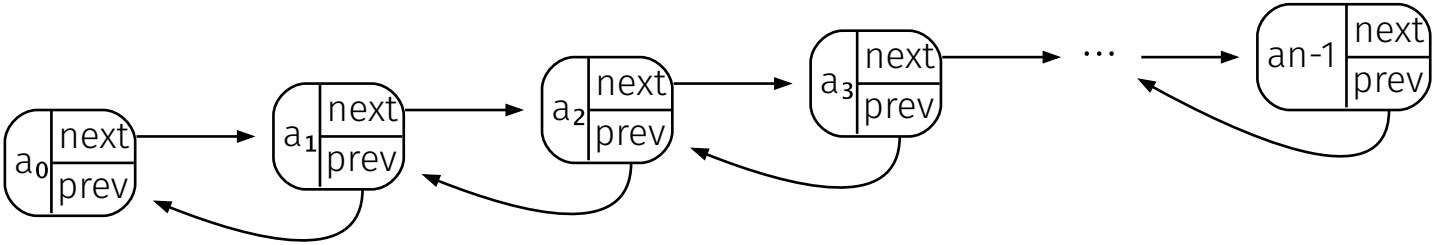
a_0	a_1	a_2	a_3	$\cdots$	a_{n-1}

Note

- Gli elementi si trovano in una regione **contigua** di memoria
- È facile accedere a ogni singolo elemento (indirizzo dell'elemento in posizione i = posizione primo elemento + $i \times$ dimensione elemento)
- È costoso inserire o rimuovere un elemento in una posizione arbitraria in quanto occorre **traslare** tutti quelli che stanno alla sua destra (il costo è proporzionale alla distanza dell'elemento dal fondo del vettore)
- È facile inserire o rimuovere un nuovo elemento in fondo (non sono necessarie traslazioni)

Rappresentazione di liste

Dentro `std::list`



Note

- Ogni elemento è memorizzato in un **nodo** la cui posizione in memoria è indipendente da quella degli altri nodi e dunque imprevedibile.
- Ogni nodo contiene un riferimento al successivo e al precedente.
- È costoso raggiungere ogni singolo nodo in quanto occorre seguire la catena di riferimenti dall'inizio (o dalla fine) (il costo è proporzionale alla posizione del nodo dall'inizio o dalla fine)
- Trovato un nodo, è facile rimuoverlo o inserirne un altro prima o dopo al nodo stesso in quanto non è necessaria alcuna traslazione, basta aggiornare opportunamente i riferimenti.
- **Attenzione**: ogni nodo **occupa più memoria** di un singolo elemento

Esempio: ricerca di un elemento nullo

Vettore

```
bool ricerca(const std::vector<int>& c) {
  for (int i = 0; i < c.size(); i++)
    if (c[i] == 0) return true;
  return false;
}
```

- L'accesso all'elemento in posizione i ha un costo indipendente da i
- La ricerca ha un costo proporzionale a n (caso pessimo)

Lista (questo codice è **ipotetico**, non c'è l'operatore [ ] sulle liste)

```
bool ricerca(const std::list<int>& c) {
  for (int i = 0; i < c.size(); i++)
    if (c[i] == 0) return true;
  return false;
}
```

- L'accesso all'elemento in posizione i ha un costo proporzionale a i
- La ricerca ha un costo proporzionale a n^2 (caso pessimo)

Iteratori

- Un **iteratore** è un oggetto mutabile che **indica** un elemento della lista
- `std::list<T>::iterator` è il tipo degli **iteratori** su liste di T
- `std::list<T>::const_iterator` è il tipo degli **iteratori costanti** su liste di T (questi iteratori non possono modificare la lista)
- L'operatore `++` sposta l'iteratore all'elemento **successivo**
- L'operatore `--` sposta l'iteratore all'elemento **precedente**
- L'operatore prefisso `*` **accede** all'elemento
- Il metodo `begin()` crea un iteratore che indica il **primo elemento**
- Il metodo `end()` crea un iteratore che indica l'**elemento fittizio** successivo all'**ultimo**
- Analogamente, i metodi `cbegin()` e `cend()` creano iteratori costanti

```
bool ricerca(const std::list<int>& l) {
  std::list<int>::const_iterator it = l.cbegin();
  while (it ≠ l.cend())
    if (*it == 0) return true;
    else it++;
  return false;
}
```

Esempio: insiemi mutabili

- Creiamo una classe `Set` di **insiemi mutabili**, ovvero insiemi in cui è possibile inserire e rimuovere elementi
- Usiamo un **contenitore** per mantenere gli elementi dell'insieme
- È conveniente mantenere il **contenitore ordinato**, così la ricerca di un elemento può interrompersi non appena se ne trova uno più grande
- Quando si aggiunge un elemento a un insieme occorre **inserirlo** nel punto giusto (non necessariamente in fondo) per preservare l'ordine del contenitore
- Per questo motivo, il contenitore appropriato è la **lista** (tra quelli visti)

```
class Set {
private:
  std::list<int> data; // invariante: lista ordinata
public:
  int size() const { return data.size(); }
  bool contains(int) const;
  void insert(int);
  void remove(int);
  ...
};
```

Ricerca di un elemento

```
bool Set::contains(int x) const {
  std::list<int>::const_iterator it = data.cbegin();
  while (it ≠ data.cend() && *it < x) it++;
  return it ≠ data.cend() && *it == x;
}
```

- usiamo un iteratore **costante** poiché la ricerca non modifica la lista
- cerchiamo `x` all'interno della lista, la ricerca termina quando si verifica una delle seguenti possibilità:
  - abbiamo raggiunto la **fine della lista** (`it == data.cend()`), in tal caso tutti gli elementi già presenti sono minori di `x`, **oppure**
  - abbiamo trovato un elemento (indicato da `it`) che è **maggiore o uguale** a `x`
- restituiamo `true` se `it` indica **proprio** `x`, `false` altrimenti

Rimozione di un elemento

```
void Set::remove(int x) {
  std::list<int>::iterator it = data.begin();
  while (it ≠ data.end() && *it < x) it++;
  if (it ≠ data.end() && *it == x) data.erase(it);
}
```

- usiamo un iteratore **non costante** poiché la rimozione, in generale, modifica la lista
- cerchiamo x all'interno della lista, la ricerca termina quando si verifica una delle seguenti possibilità:
  - abbiamo raggiunto la **fine della lista** (`it == data.cend()`), in tal caso tutti gli elementi già presenti sono minori di x, **oppure**
  - abbiamo trovato un elemento (indicato da `it`) che è **maggiore o uguale** a x
- se x è stato trovato, lo rimuoviamo (il metodo `data.erase(it)` rimuove l'elemento indicato da `it` dalla lista `data`)

Inserimento di un elemento

```
void Set::insert(int x) {
  std::list<int>::iterator it = data.begin();
  while (it ≠ data.end() && *it < x) it++;
  if (it == data.end() || *it ≠ x) data.insert(it, x);
}
```

- usiamo un iteratore **non costante** poiché l'inserimento, in generale, modifica la lista
- cerchiamo x all'interno della lista, con un duplice obiettivo:
  - se lo troviamo, non dobbiamo inserirlo
  - se non lo troviamo, individuiamo il punto giusto in cui inserirlo
- la ricerca termina quando si verifica una delle seguenti possibilità:
  - abbiamo raggiunto la fine della lista (`it == data.cend()`), in tal caso tutti gli elementi già presenti sono minori di x, **oppure**
  - abbiamo trovato un elemento (indicato da `it`) che è **maggiore o uguale** a x
- se x **non** è stato trovato, lo inseriamo **prima** di quello indicato da `it` (se `it == data.end()`, l'effetto è quello di inserirlo **in fondo** alla lista, come deve essere)

Stampa di insiemi

Dilemma

- Per stampare insiemi sul terminale vorremmo scrivere l'operatore << per la classe `Set`
- Tale operatore **non è** un membro della classe, pertanto non ha accesso diretto al campo `data` che è privato
- Se rendessimo **pubblico** il campo `data` esporremmo la classe `Set` a usi impropri (es. potremmo modificare direttamente la lista `data` violando l'invariante di classe).

Soluzione

Il progettista di una classe può definire funzioni **amiche** che, pur essendo esterne alla classe, hanno accesso ai campi privati della classe:

```
class Set {
  ...
  friend std::ostream& operator<<(std::ostream&,
                                  const Set&);
  ...
};
```

Stampa di insiemi con funzione amica

```
std::ostream& operator<<(std::ostream& os, const Set& s) {
  std::cout << "{";
  std::list<int>::const_iterator it = s.data.cbegin();
  while (it ≠ s.data.cend()) {
    std::cout << *it;
    it++;
    if (it ≠ s.data.cend()) std::cout << ",";
  }
  std::cout << "}";
  return os;
}
```

- l'`if` all'interno del ciclo è necessario per stampare il separatore `,` **solo** quando c'è almeno un altro elemento dell'insieme da stampare

Esercizi

1. Aggiungere alla classe `Set` un metodo `empty` che restituisca `true` se l'insieme è vuoto, `false` altrimenti.
2. Aggiungere alla classe `Set` un metodo `elements` che restituisca il vettore ordinato contenente tutti gli elementi di un insieme.
3. Aggiungere alla classe `Set` i metodi `minimum` e `maximum` che restituiscano, rispettivamente, l'elemento più piccolo e più grande di un insieme non vuoto.
4. Aggiungere alla classe `Set` i metodi `_union`, `intersection` e `difference` che calcolino rispettivamente l'unione, intersezione e la differenza di insiemi. Fare in modo che tali metodi non modifichino gli insiemi coinvolti nell'operazione. **Nota**: il metodo di unione non può chiamarsi `union` perché questa è una parola riservata del linguaggio.
5. Aggiungere alla classe `Set` un metodo `overlap(const Set&)` che restituisca `true` se due insiemi hanno un'intersezione non vuota, `false` altrimenti. Se possibile, scrivere `overlap` in modo tale che il costo dell'esecuzione del metodo sia, nel caso pessimo, proporzionale alla somma delle dimensioni dei due insiemi.