Università degli Studi di Camerino
Scuola di Scienze e Tecnologie
Corso di Laurea in Informatica
Corso di Algoritmi e Strutture Dati 2020/2021
Parte di Laboratorio (6 CFU)
Docente: Luca Tesei

Istruzioni per la realizzazione del Miniprogetto 1

ASDL2021MP1

# Descrizione

Il miniprogetto ASDL2021MP1 consiste nei seguenti task:

1. Implementare la classe `ASDL2021Deque<E>` `implements` `java.util.Deque<E>` tramite una lista concatenata doppia
2. Implementare la classe `BalancedParenthesesChecker` usando la propria implementazione della classe `ASDL2021Deque<E>` per fornire uno stack di `Character`
3. Implementare la classe `TernaryHeapMinPriorityQueue` tramite un min-heap ternario

## Double ended queue (Deque)

Una coda a doppia entrata è una struttura dati lineare e sequenziale che permette di inserire un elemento di un qualsiasi tipo, diciamo `E`, in testa o in coda agli elementi attualmente presenti. Analogamente permette di estrarre dalla testa o dalla coda un elemento già presente.

Data questa sua caratteristica può essere usata sia come coda FIFO (First-in First-out), cioè una normale coda tipo quella a uno sportello in cui viene estratto sempre l'elemento inserito meno recentemente e non ancora estratto, sia come coda LIFO (Last-in First-out), cioè una pila (stack) tipo una pila di piatti in cui viene estratto sempre l'elemento inserito più recentemente e non ancora estratto.

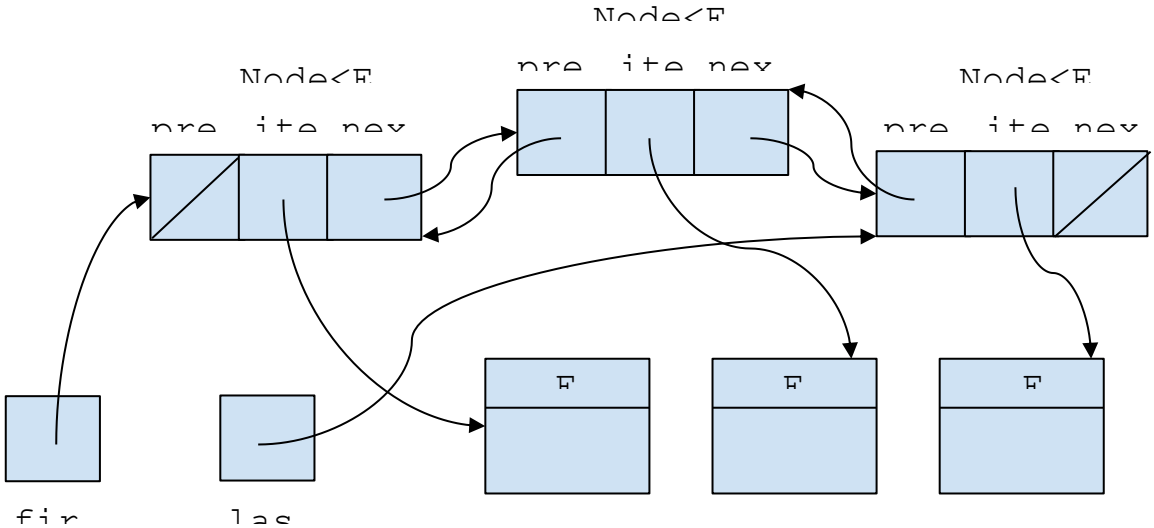
Delle API per una Deque sono già disponibili nell'interface `java.utili.Deque` che si possono consultare su

https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html

Il template del codice per la

## public class ASDL2021Deque<E> implements Deque<E>

nel file `ASDL2021Deque.java` fornito in allegato propone una implementazione tramite una lista concatenata doppia i cui elementi possono essere illustrati graficamente come segue



La classe `Node<E>` è una classe `static` interna della classe `ASDL2021Deque<E>` i cui oggetti sono i nodi della lista concatenata. Le due variabili istanza `first` e `last` sono puntatori al nodo testa e al nodo coda attuali della deque. **Tale struttura deve essere usata nell'implementazione in quanto i test JUnit a cui sarà sottoposto il codice consegnato assumeranno questa struttura** (quindi si eviti di usare altre soluzioni come liste circolari, array, sentinelle ecc.).

Nel template fornito per la classe `ASDL2021Deque<E>` sono da implementare tutti i metodi nei quali c'è il commento

```
// TODO implement
```

realizzando in maniera precisa le API specificate in https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html o nelle API delle super-interface di `Deque<E>`, cioè `Queue<E>`, `Collection<E>` e `Iterable<E>`, nel caso in cui le API ereditate non siano state specializzate in `Deque<E>`.

Si possono ignorare (cioè non verranno testate), nella realizzazione delle API, il lancio delle seguenti eccezioni, quando indicate con le motivazioni associate:

- `IllegalStateException - if the element cannot be added at this time due to capacity restrictions`
- `ClassCastException - if the class of the specified element prevents it from being added to this deque`
- `IllegalArgumentException - if some property of the specified element prevents it from being added to this deque`

in quanto l'implementazione che si deve realizzare non deve avere restrizioni al numero di elementi che si possono inserire (che possono essere anche duplicati) e accetta qualsiasi oggetto della classe `E`, tranne i riferimenti `null`.

Nel template sono presenti anche i seguenti metodi, che servono per poter implementare correttamente l'interface `Deque<E>`, ma che non devono essere implementati (lanciano l'eccezione `UnsupportedOperationException`):

- `public <T> T[] toArray(T[] a)`
- `public boolean removeAll(Collection<?> c)`
- `public boolean retainAll(Collection<?> c)`
- `public boolean removeFirstOccurrence(Object o)`
- `public boolean removeLastOccurrence(Object o)`

Nella classe `ASDL2021Deque<E>` è richiesto di implementare anche i due metodi `Iterator<E> iterator()` e `Iterator<E> descendingIterator()` che devono permettere di esaminare gli elementi della deque dalla testa alla coda e dalla coda alla testa (rispettivamente). Per implementare tali metodi sono previste nel template le due classi interne **non static** `Itr` e `DescItr`. I loro oggetti devono implementare i metodi `hasNext()` e `next()` nella maniera corretta indicata dalle API e devono comportarsi in maniera **fail-fast**. Un iteratore è **fail-fast** quando è in grado di fallire, cioè in questo caso lanciare un'eccezione `java.util.ConcurrentModificationException`, non appena si accorge che la deque che si sta scorrendo con l'iteratore è stata modificata in qualche modo dalla chiamata di uno dei metodi della classe principale. Tipicamente il lancio dell'eccezione deve accadere alla prima chiamata del metodo `next()` successiva alla modifica effettuata durante l'iterazione.

Per maggiori dettagli di come implementare una lista concatenata doppia ci si può ispirare al Capitolo 10 del libro di testo

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduzione agli algoritmi 3/ED*. McGraw- Hill, 2010.

## Parentesi bilanciate

Una stringa contenente delle parentesi tonde, quadre o graffe (nel nostro caso) si dice bilanciata quando:

1. una parentesi che si chiude è dello stesso tipo dell'ultima parentesi che si è aperta
2. una parentesi che si apre verrà sempre chiusa

3. una parentesi chiusa corrisponde sempre a una parentesi aperta precedentemente
4. non ci sono caratteri estranei alle parentesi, ma sono ammessi spazi, tabulazioni (in Java `'\t'`) e caratteri di newline (in Java `'\n'`).

Alcuni esempi:

- `" (( [( {\t (\t) [ ] } ) \n ] ) ) "` è bilanciata perché soddisfa tutte le proprietà di cui sopra
- `" "` è bilanciata perché soddisfa tutte le proprietà di cui sopra
- `"(([ )] )"` **non** è bilanciata perché viola la proprietà 1.
- `"( { } "` **non** è bilanciata perché viola la proprietà 2.
- `"}(([]))"` **non** è bilanciata perché viola la proprietà 3.
- `"( ( \n [(P)] \t ))"` **non** è bilanciata perché viola la proprietà 4.

Un modo efficace per controllare se una data stringa soddisfa tutte le proprietà richieste è quello di usare una pila (uno stack) in cui i simboli vengono via via inseriti ed estratti controllando che le proprietà siano rispettate. L'implementazione della classe `BalancedParenthesesChecker` richiede che venga usato un oggetto della classe `ASDL2021Deque<Character>` come stack per effettuare questi controlli. Il resto delle API sono specificate nel template fornito `BalancedParenthesesChecker.java`.

## Coda di Priorità Dinamica

Una coda di priorità è una struttura dati in cui si possono inserire elementi a cui è associata una certa priorità. L'elemento in testa alla coda di priorità, e che quindi può essere estratto, è sempre l'elemento che ha la priorità maggiore (o minore a seconda del tipo di coda). L'aggettivo "dinamica" significa che la priorità di un elemento già in coda può essere aumentata (o diminuita a seconda del tipo di coda) con un'apposita operazione. Ciò comporta che la struttura si aggiorni di conseguenza e potrebbe accadere che l'elemento a cui si aumenta (o si diminuisce) la priorità debba passare in testa alla coda mentre prima non lo era.

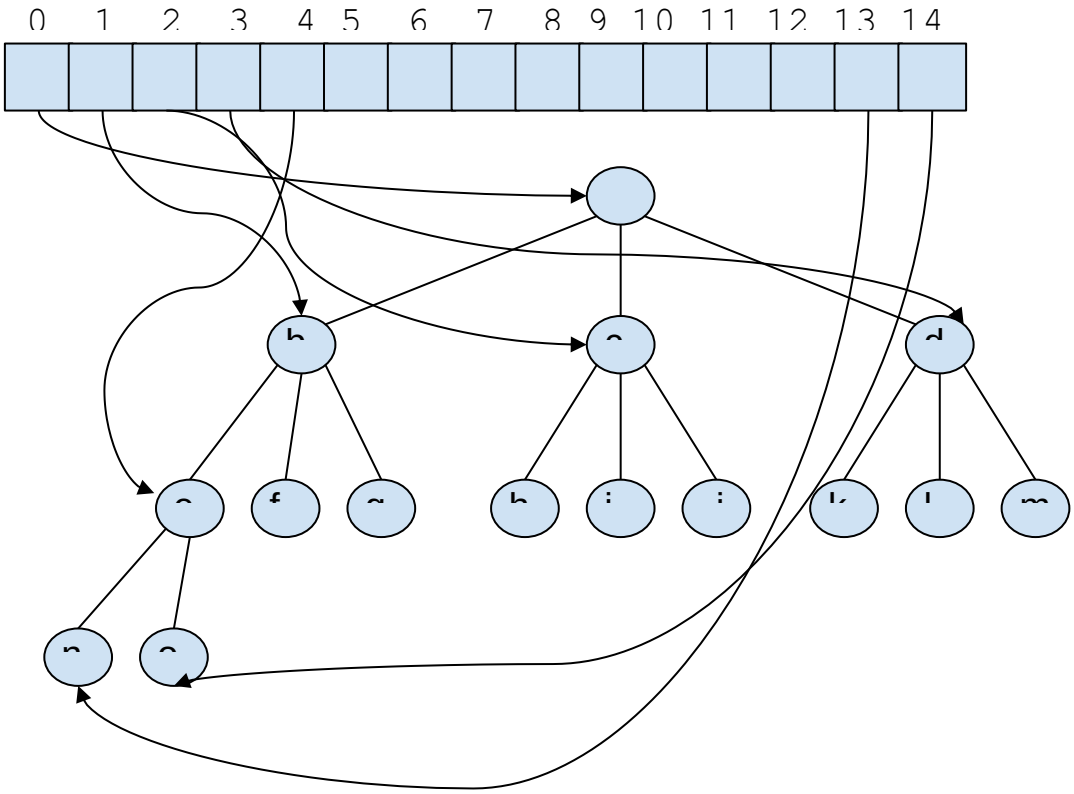
Nel nostro caso si vuole implementare nella classe `TernaryHeapMinPriorityQueue`, di cui viene dato il template, una coda min-priority, cioè una coda di priorità in cui l'elemento in testa è sempre un elemento con priorità *minima*. Viene fornita l'interface `PriorityQueueElement` come tipo generico (polimorfo) degli elementi della coda. Gli oggetti di una qualsiasi classe che implementi questa interface possono essere inseriti nella coda di min-priority. L'interface richiede che gli oggetti abbiano i due metodi set/get per gestire la priorità (un numero double).

Inoltre, è richiesto che la coda di min-priority in `TernaryHeapMinPriorityQueue` sia implementata con un min-heap ternario. Un min-heap ternario funziona come un min-heap binario, cioè:

- è un albero completo tranne l'ultimo livello
- ogni nodo ha priorità minore o uguale alla priorità di tutti i suoi discendenti

- ogni nodo non foglia ha esattamente 3 figli, tranne nel caso del penultimo livello in cui il numero di figli per un solo nodo non foglia (quello non completo più a sinistra) può essere minore di 3

La figura seguente mostra un min-heap ternario rappresentato tramite un array di nome `heap`.




Gli elementi hanno le seguenti caratteristiche:

| | | | |
|---|---|---|---|
| `heap[0] = a`<br>`a.getPriority() == 1`<br>`a.getHandle() == 0` | `heap[1] = b`<br>`b.getPriority() == 2`<br>`b.getHandle() == 1` | `heap[2] = c`<br>`c.getPriority() == 8`<br>`c.getHandle() == 2` | `heap[3] = d`<br>`d.getPriority() == 12`<br>`d.getHandle() == 3` |
| `heap[4] = e`<br>`e.getPriority() == 3`<br>`e.getHandle() == 4` | `heap[5] = f`<br>`f.getPriority() == 6`<br>`f.getHandle() == 5` | `heap[6] = g`<br>`g.getPriority() == 7`<br>`g.getHandle() == 6` | `heap[7] = h`<br>`h.getPriority() == 9`<br>`h.getHandle() == 7` |
| `heap[8] = i`<br>`i.getPriority() == 10`<br>`i.getHandle() == 8` | `heap[9] = j`<br>`j.getPriority() == 11`<br>`j.getHandle() == 9` | `heap[10] = k`<br>`k.getPriority() == 13`<br>`k.getHandle() == 10` | `heap[11] = l`<br>`l.getPriority() == 14`<br>`l.getHandle() == 11` |
| `heap[12] = m`<br>`m.getPriority() == 15`<br>`m.getHandle() == 12` | `heap[13] = n`<br>`n.getPriority() == 4`<br>`n.getHandle() == 13` | `heap[14] = o`<br>`n.getPriority() == 5`<br>`n.getHandle() == 14` | `heap[15] = null or`<br>`does not exist` |

La radice dello heap si trova in posizione 0 dell'array (**è richiesto che nell'implementazione data gli indici partano da 0**). Qui c'è un puntatore a un oggetto di una classe `X implements PriorityQueueElement` che per comodità chiamiamo `a`. Chiamando sull'oggetto puntato da `a` il metodo `getPriority()` otteniamo il valore più basso di tutto l'albero cioè 1. In posizione 1 dell'array c'è il primo figlio a sinistra della radice, `b`, che ha priorità 2. Esso ha la priorità più bassa di tutti i suoi nodi discendenti, cioè `e`, `f`, `g`, `n` e `o`. E così via.

La **handle**, che si ottiene con il metodo `getHandle()`, rappresenta la posizione corrente dell'oggetto di tipo polimorfo `PriorityQueueElement` nell'array che rappresenta lo heap ternario. **Questo indice deve essere sempre aggiornato se l'oggetto viene spostato durante il riadattamento dello heap** (ad esempio quando viene aggiunto un nuovo elemento, quando viene estratto il minimo o quando viene modificata la priorità di un certo elemento). **L'implementazione fornita deve garantire questa proprietà**.

La presenza della handle come attributo dell'oggetto di tipo polimorfo `PriorityQueueElement` è fondamentale per ottenere una prestazione efficiente del metodo

```
public void decreasePriority(PriorityQueueElement element, double newPriority)
```

infatti, in questo modo in tempo *O(1)* si può ottenere l'indice dell'elemento nell'array che rappresenta lo heap e procedere quindi all'aggiustamento della struttura (che può essere fatta in $O(\log_3 n)$). Se non avessimo l'indice disponibile dovremmo fare una ricerca tra tutti gli elementi dell'array per trovare quello di cui vogliamo modificare la priorità (oltre a dover risolvere l'ambiguità che si può creare visto che ci possono essere elementi duplicati). Ciò costerebbe, nel caso pessimo, *O(n)* vanificando lo sforzo di usare lo heap per ottenere prestazioni migliori di quella lineare.

Usando lo heap ternario e l'accorgimento della handle è possibile ottenere le seguenti prestazioni dalla coda di min-priorità:

| **Operazione** | **Heap Ternario (caso pessimo)** |
|---|---|
| Inserimento di un elemento | $O(\log_3 n)$ |
| Ricerca del Min | $\Theta(1)$ |
| Estrazione del Min | $\Theta(\log_3 n)$ |
| Decremento di Priorità | $O(\log_3 n)$ |

Nel template fornito per la classe `TernaryHeapMinPriorityQueue` l'array **deve essere implementato** come ArrayList:

```
private ArrayList<PriorityQueueElement> heap
```

In questo modo non ci sono problemi di capacità massima e, come richiesto, il numero di elementi nella coda non è limitato a priori. Si ricordi, come detto sopra, che **l'implementazione deve garantire che gli indici partano da 0** e, inoltre, che **il numero di elementi nell'ArrayList heap sia sempre esattamente il numero di elementi nella coda** (cioè a ogni estrazione del minimo si deve eliminare un elemento dall'ArrayList). I test JUnit a cui sarà sottoposto il codice implementato assumeranno questi due fatti.

Le classi `Job` e `Scheduler`, fornite con la traccia, danno un esempio di come utilizzare una coda di min-priorità per schedulare dei job su un processore in base alla scadenza più prossima. Si utilizza come priorità la scadenza, rappresentata da un numero double. Più piccolo è il numero double più prossima è la scadenza. Queste due classi sono fornite solo a scopo dimostrativo di come può essere usato un oggetto di `TernaryHeapMinPriorityQueue`.

Per maggiori dettagli di come implementare le operazioni dello heap ternario ci si può ispirare a quelle sullo heap binario che sono spiegate nel Capitolo 6 del libro di testo

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduzione agli algoritmi 3/ED*. McGraw- Hill, 2010.

# Traccia e Implementazione

La traccia del codice è fornita come .zip contenente i template delle seguenti classi/interfacce:

- `it.unicam.cs.asdl2021.mp1.ASDL2021Deque`
- `it.unicam.cs.asdl2021.mp1.BalancedParenthesesChecker`
- `it.unicam.cs.asdl2021.mp1.PriorityQueueElement`
- `it.unicam.cs.asdl2021.mp1.TernaryHeapMinPriorityQueue`
- `it.unicam.cs.asdl2021.mp1.Job`
- `it.unicam.cs.asdl2021.mp1.Scheduler`

che si possono importare in Eclipse o nel proprio IDE preferito. La versione del compilatore Java da definire nel progetto Eclipse (o altro IDE) è la 1.8 (Java 8).

Nell'implementazione:

- **non si possono usare versioni del compilatore superiori a 1.8**;
- **è vietato l'uso di lambda-espressioni** e di funzionalità avanzate di Java 8 o superiore (es. stream, ecc…);
- **è obbligatorio usare il set di caratteri utf8** per la codifica dei file del codice sorgente.

**Non sono fornite le classi di test JUnit che saranno utilizzate per testare il codice consegnato.** Quindi è **estremamente importante** leggere bene questo documento, le API scritte nel formato javadoc nei template delle classi e le API in

https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html (e sue super-interface) in modo da implementare i metodi richiesti nella maniera corretta. In caso di dubbi si utilizzi il documento google condiviso associato a questo compito denominato "`ASDL2021MP1 Q&A Traccia`" per controllare le risposte a domande già fatte e/o per fare una nuova domanda.

**La scrittura di propri test JUnit per controllare che la propria implementazione funzioni bene è fortemente consigliata.** Tuttavia tali test non dovranno essere consegnati come parte del progetto implementato, saranno solo a garanzia personale di correttezza del codice.

# Modalità di Sviluppo e Consegna

Vanno implementati tutti i metodi richiesti (segnalati con commenti della forma `// TODO implement` nel template delle classi). Non è consentito:

- aggiungere classi pubbliche;
- modificare la firma (signature) dei metodi già specificati nella traccia;
- modificare le variabili istanza già specificate nella traccia.

E' consentito:

- aggiungere classi interne private per fini di implementazione;
- aggiungere metodi privati per fini di implementazione;
- aggiungere variabili istanza private per fini di implementazione.

Nel file sorgente di ogni classe implementata, nel commento javadoc della classe, modificare il campo @author come indicato: "INSERIRE NOME E COGNOME DELLO STUDENTE - INSERIRE ANCHE L'EMAIL xxxx@studenti.unicam.it".

Creare una cartella con il seguente nome con le **lettere maiuscole e i trattini** (non underscore!) **esattamente come indicato senza spazi aggiuntivi o altri caratteri**:

**ASDL2021-COGNOME-NOME-MP1**

ad esempio **ASDL2021-ROSSI-MARIO-MP1**. Nel caso di più nomi/cognomi usare solo il primo cognome e il primo nome. Nel caso di lettere accentate nel nome/cognome usare le corrispondenti lettere non accentate (maiuscole). Nel caso di apostrofi o altri segni nel nome/cognome ometterli. Nel caso di particelle nel nome o nel cognome, ad esempio De Rossi o De' Rossi, attaccarle (DEROSSI).

All'interno di questa cartella copiare i file sorgenti java modificati con la propria implementazione:

- `ASDL2021Deque.java`
- `BalancedParenthesesChecker.java`
- `TernaryHeapMinPriorityQueue.java`

che di solito si trovano nella cartella

`cartella-del-progetto/src/it/unicam/cs/asdl2021/mp1`

all'interno del workspace. Non si modifichi il package delle classi/interfacce!

Comprimere la cartella in formato .zip (**non rar o altro, solo zip**) e chiamare l'archivio

**ASDL2021-COGNOME-NOME-MP1.zip**

**ATTENZIONE**: se i passaggi descritti per la consegna non vengono seguiti **precisamente** (ad esempio nome della cartella sbagliato, contenuto dello zip diverso da quello indicato, formato non zip ecc.) lo studente **perderà automaticamente 3 punti nel voto del miniprogetto**.
Consegnare il file **ASDL2021-COGNOME-NOME-MP1.zip** tramite Google Classroom (usare la funzione consegna associata al post di assegnazione del miniprogetto) entro la data di **scadenza**, cioè **Lunedì 23 Novembre 2020 ore 18.00**.

# Valutazione

**ATTENZIONE:** Il codice consegnato verrà sottoposto a un **software antiplagio** che lo confronterà con tutti gli altri codici consegnati dagli altri studenti. Il software segnala una percentuale di somiglianza e dei gruppi di studenti con codice probabilmente plagiato.

La valutazione si baserà sui seguenti criteri, in ordine decrescente di importanza:

1. **Codice scritto individualmente. Nel caso di conclamato "plagio" il voto del miniproject 1 di tutti i "plagi" verrà decurtato di 8 punti.**
2. **Correttezza**. Il codice consegnato verrà sottoposto a dei test JUnit che controlleranno tutte le funzionalità. Tali test verranno resi pubblici dopo la data di scadenza per la consegna in modo da poter essere eseguiti per vedere eventuali errori commessi. Sarà possibile chiedere spiegazioni e/o chiarimenti riguardo ai test. Il mancato superamento di un test comporterà il decurtamento di un certo numero di punti (a seconda del test, una griglia di valutazione verrà fornita all'atto della riconsegna con il voto).
3. **Codice chiaro, leggibile e ben commentato**.
4. Scelta di strutture dati e implementazione **efficienti** sia dal punto di vista del tempo di esecuzione che dello spazio richiesto.

Il voto assegnato al miniprogetto verrà comunicato tramite Google Classroom. Il voto sarà espresso in 30esimi e peserà per il 40% del voto finale ottenuto con le prove parziali per ASDL2021. Il miniprogetto 2 peserà per il 40% e il restante 20% sarà ricavato dalla consegna delle Esercitazioni a Casa (si veda il documento Google condiviso "`ASDL2021 Prove Parziali`" per tutti i dettagli).