Progetto del corso di Strutture Dati a.a. 2013-14

Il presente documento contiene l'elenco delle classi che devono essere incluse nel progetto del corso di Strutture Dati.

STACK:

- La classe ArrayStack che implementa Stack mediante un array.
- La classe NodeStack che implementa Stack mediante una lista a puntatori singoli.

    - Oltre a contenere i metodi dell'interfaccia Stack, le classi ArrayStack e NodeStack devono sovrascrivere il metodo toString .

QUEUE:

- La classe ArrayQueue che implementa Queue mediante un array usato in modo circolare. Quando non risulta possibile effettuare ulteriori inserimenti, l'array viene sostituito con uno più grande.
- La classe NodeQueue che implementa Queue mediante una lista a puntatori singoli.

    - Oltre a contenere i metodi dell'interfaccia Queue, le classi ArrayQueue e NodeQueue devono sovrascrivere il metodo toString .

DEQUE:

- La classe NodeDeque che implementa Deque con una lista a doppi puntatori.

    - Oltre a contenere i metodi dell'interfaccia Deque, le classi NodeDeque deve sovrascrivere il metodo toString .

NODE LIST:

- La classe NodePositionList che implementa PositionList mediante una lista a doppi puntatori.
    - Oltre a contenere i metodi dell'interfaccia PositionList, la classe NodePositionList deve
        1. sovrascrivere il metodo toString
        2. contenere il metodo

           public void reverse()

           che inverte ricorsivamente la lista.

ARRAY LIST:

- La classe ArrayIndexList che implementa IndexList mediante un array.
    - Quando non risulta possibile effettuare ulteriori inserimenti, l'array viene sostituito con uno grande il **doppio**.

SEQUENCE :

- La classe NodeSequence che implementa Sequence mediante una lista a doppi puntatori;
    - La classe NodeSequence deve sovrascrivere il metodo toString
- La classe ArraySequence che implementa Sequence mediante un array
    - La classe ArraySequence deve sovrascrivere il metodo toString

    - Oltre a contenere i metodi dell'interfaccia Sequence, le classi ArraySequence e NodeSequence devono sovrascrivere il metodo toString .

ITERATOR:

- La classe IndexListIterator che implementa Iterator mediante un vettore;
- La classe ElementIterator che implementa Iterator per il tipo PositionList
    - L' implementazione di ElementIterator deve utilizzare un cursore.

TREE :

- La classe LinkedTree che implementa Tree con una struttura a puntatori in cui ciascun nodo contiene, oltre all'elemento, un riferimento al padre e un riferimento alla collezione dei figli.
    - Oltre ai metodi dell'interfaccia Tree, la classe LinkedTree deve contenere i seguenti metodi di modifica:
        - Position<E> addRoot(E **elt**): se l'albero è vuoto, inserisce nell'albero un nodo contenente **elt** restituendolo poi in output (ovviamente il nuovo nodo diventa radice dell'albero); se l'albero non è vuoto lancia l'eccezione NonEmptyTreeException.
        - Position <E> addChild(E e,Position <E> v): aggiunge all'albero un nodo foglia avente **v** come padre e contenente l'elemento **e**; se **v** non è una position valida lancia l'eccezione **InvalidPositionException**.
        - E removeRoot(): se l'albero contiene solo la radice, rimuove la radice e ne restituisce in output l'elemento; se l'albero è vuoto lancia l'eccezione **EmptyTreeException**; se l'albero contiene più di un nodo lancia l'eccezione **UndeletableNodeException**.
        - E removeExternalChild(Position<E> v): se il primo figlio di **v** è una foglia, lo cancella e ne restituisce in output l'elemento; se il primo figlio di **v** non è una foglia lancia l'eccezione **UndeletableNodeException**; se **v** è una foglia o non è una position valida lancia l'eccezione **InvalidPositionException**.
        - E remove(Position<E> v): se **v** è una foglia, rimuove **v** dall'albero; se l'albero è vuoto o **v** non è una posizione valida allora il metodo lancia **InvalidPositionException.**

BINARY TREE :

- La classe EulerTour contenente il template method EulerTour;
- Una sottoclasse (a scelta dello studente) che specializza EulerTour.
- La classe LinkedBinaryTree che implementa BinaryTree con una struttura a puntatori in cui ciascun nodo contiene oltre all'elemento, un riferimento al padre, un riferimento al figlio sinistro e un riferimento al figlio destro.
    - Oltre ai metodi dell'interfaccia BinaryTree, la classe LinkedBinaryTree deve contenere i seguenti metodi di modifica:
        - Position<E> addRoot(E **elt**): se l'albero è vuoto, inserisce nell'albero un nodo contenente **elt** restituendolo poi in output (ovviamente il nuovo nodo diventa radice dell'albero); se l'albero non è vuoto lancia l'eccezione NonEmptyTreeException;
        - Position <E> insertLeft(E **elt**,Position <E> **v**) : se **v** non ha un figlio sinistro, crea e restituisce in output una foglia contenente **elt** e fa diventare questo nuovo nodo figlio sinistro di **v**; altrimenti lancia l'eccezione InvalidPositionException;
        - Position <E> insertRight(E **elt**,Position <E> **v**) : se **v** non ha un figlio sinistro, crea e restituisce in output una foglia contenente **elt** e fa diventare questo nuovo nodo figlio destro di **v**; altrimenti lancia l'eccezione InvalidPositionException;
        - void attachLeaves(PositionList<E>L): trasforma le foglie dell'albero in nodi interni aventi come figli due foglie. Gli elementi delle nuove foglie devono essere prelevati dalla lista **L (**in base all'ordine in cui sono disposti in **L)**. Si assuma che **L** contenga un numero di elementi pari a due volte il numero di foglie iniziale dell'albero.

        - Metodi di modifica usati nell'implementazione BinarySearchTree

PRIORITY QUEUE:

- La classe UnsortedListPriorityQueue che implementa PriorityQueue mediante un'istanza di PositionList in cui le entrate compaiono in un ordine arbitrario.
- La classe SortedListPriorityQueue che implementa PriorityQueue mediante un'istanza di PositionList in cui le entrate sono ordinate in base ai valori delle chiavi.
- La classe HeapPriorityQueue che implementa PriorityQueue mediante un heap.
    - La classe HeapPriorityQueue deve contenere anche il costruttore

      HeapPriorityQueue(K **k**[] , V **v**[], Comparator<K> **C**)

      che prende in input due array **k**[] e **v**[] di uguale lunghezza e un comparatore **C**, e costruisce, **in tempo lineare nella lunghezza dei due array,** una coda a priorità contenente le entrate (**k**[0],**v**[0]), (**k**[1],**v**[1]) , ….,(**k**[n-1],**v**[n-1]), dove n indica la lunghezza dei due array.

ADAPTABLE PRIORITY QUEUE:

- SortedListAdaptablePriorityQueue che implementa AdaptablePriorityQueue estendendo la classe SortedListPriorityQueue.
- La classe HeapAdaptablePriorityQueue che implementa AdaptablePriorityQueue mediante un heap.

COMPARATOR :

- La classe DefaulComparator che usa il metodo compareTo di java.lang.Comparable per effettuare i confronti
- Un comparatore per confrontare oggetti di un tipo a vostra scelta.

COMPLETE BINARY TREE :

- La classe ArrayListCompleteBinaryTree che implementa CompleteBinaryTree con un'istanza di IndexList.

MAP :

- La classe ListMap che implementa Map mediante un'istanza di PositionList.
- La classe HashTableMap che implementa Map mediante una tabella hash in cui i conflitti sono risolti con il metodo del linear probing.
    - Il load factor deve essere mantenuto al di sotto di 0,5.

DICTIONARY :

- La classe LogFile che implementa Dictionary mediante un'istanza di PositionList in cui le entrate compaiono in un ordine arbitrario
- La classe ChainingHashTable che implementa Dictionary mediante una tabella hash in cui i conflitti sono risolti con il metodo del chaining.
    - Il load factor deve essere mantenuto al di sotto di 0,9.
- La classe LinearProbingHashTable che implementa Dictionary mediante una tabella hash in cui i conflitti sono risolti con il metodo del linear probing.
    - Il load factor deve essere mantenuto al di sotto di 0,5.
- La classe BinarySearchTree che implementa Dictionary mediante un albero di ricerca binario. La classe BinarySearchTree deve estendere la classe LinkedBinaryTree.

SET :

- La classe OrderedListSet che implementa Set mediante un'istanza di PositionList che contiene gli elementi dell'insieme ordinati secondo una certa relazione d'ordine totale.
- I metodi union, intersect e subtract devono far uso del template method merge.

PARTITION:

- La classe ListPartition che implementa Partition mediante un'istanza di PositionList che contiene gli insiemi della partizione.
    - Il metodo find deve avere tempo di esecuzione O(1) (almeno nel caso medio).
    - Il metodo union deve essere implementato mediante l'euristica dell'unione pesata.

GRAPH:

- La classe AdjacencyListGraph che implementa Graph mediante liste di adiacenza.
    - Le interfacce Vertex ed Edge devono specializzare l'interfaccia DecorablePosition
- Le classi
    - DFS: classe che contiene un template method che effettua la visita DFS di un grafo.
    - BFS: classe che contiene un template method che effettua la visita BFS di un grafo.
    - Kruskal: classe che usa l'algoritmo di Kruskal per computare il minimo albero ricoprente di un grafo (non direzionato, pesato e connesso) e restituisce la collezione iterabile degli archi che formano il minimo albero ricoprente.

- Dijkstra: classe che usa l'algoritmo di Dijkstra per computare i cammini minimi da una sorgente a tutti gli altri vertici del grafo. Al termine dell'esecuzione del metodo, ciascun vertice deve contenere una decorazione che specifica la sua "distanza minima" dalla sorgente.
- ComponentsDFS: classe che specializza DFS per determinare le componenti connesse del grafo.
- ComponentsBFS: classe che specializza BFS per determinare le componenti connesse del grafo.
- FindPathDFS: classe che specializza DFS in modo che la visita restituisca una collezione iterabile contenente la sequenza dei vertici e degli archi lungo il percorso dalla sorgente al vertice specificato come destinazione. Se il vertice non è raggiungibile dalla sorgente allora la collezione deve essere vuota.
- FindCycleDFS: classe che specializza DFS in modo che la visita restituisca una collezione iterabile contenente la sequenza dei vertici e degli archi in un ciclo del grafo (ovviamente il ciclo deve essere raggiungibile a partire dalla sorgente). Se non esiste un ciclo raggiungibile a partire dalla sorgente, la collezione iterabile deve essere vuota.

NB: Le interfacce PositionList, Sequence, Tree, BinaryTree, CompleteBinaryTree devono essere tipi Iterable e fornire il metodo positions.