Azzolini Riccardo 2019-03-11

Implementazione di lista, pila e coda in Java

1 Lista

```
public class List<Item> {
    private Node first;
    private int N;
    private class Node {
        Item item;
        Node next;
    }

    public boolean isEmpty() {
        return first == null;
    }

    public int length() { return N; }

    public void insert(Item item, int i) {
        Node pred = first;
        Node nuovo = new Node();
        nuovo.item = item;
        if (i == 1) {
            first = nuovo;
            first.next = pred;
        } else {
            for (int j = 1; j < i - 1; j++) {
                pred = pred.next;
            }
            nuovo.next = pred.next;
            pred.next = nuovo;
        }
        N++;
    }

    public void delete(int i) {
```

```
        if (isEmpty()) return;
        Node pred = first;
        if (i == 1) {
            first = first.next;
        } else {
            for (int j = 1; j < i - 1; j++) {
                pred = pred.next;
            }
            pred.next = pred.next.next;
        }
        N--;
    }

    public Item read(int i) {
        if (isEmpty()) return null;
        Node pred = first;
        for (int j = 1; j < i; j++) {
            pred = pred.next;
        }
        return pred.item;
    }
}
```

1.1 Complessità

- `isEmpty`: $O(1) = \Theta(1)$.
- `length`: $O(1) = \Theta(1)$ perché la lunghezza viene memorizzata in una variabile, evitando così di dover scorrere l'intera lista ogni volta.
- `insert`: $O(n)$
  - caso migliore $\Theta(1)$;
  - caso peggiore $\Theta(n)$;
  - in media $O(\frac{n}{2}) = O(n)$ (se le posizioni di inserimento sono distribuite in modo uniforme).
- `delete`: $O(n)$, come `insert` (si trascura il costo di *garbage collection*).
- `read`: $O(n)$, come `insert`.

2 Pila (Stack)

```
public class Stack<Item> {
    private Node first;
    private int N;
    private class Node {
        Item item;
        Node next;
    }

    public boolean isEmpty() {
        return first == null;
    }

    public int size() { return N; }

    public void push(Item item) {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
        N++;
    }

    public Item pop() {
        if (isEmpty()) return null;
        Item item = first.item;
        first = first.next;
        N--;
        return item;
    }

    public Item top() {
        if (isEmpty()) return null;
        return first.item;
    }
}
```

2.1 Osservazioni

- Tutte le operazioni hanno complessità $\Theta(1)$.

- L'operazione `pop` implementata è una variante che restituisce anche l'elemento rimosso (in pratica, combina Top e Pop).

3 Coda (Queue)

```
public class Queue<Item> {
    private Node first;
    private Node last;
    private int N;
    private class Node {
        Item item;
        Node next;
    }

    public boolean isEmpty() {
        return first == null;
    }

    public int size() { return N; }

    public void enqueue(Item item) {
        Node oldlast = last;
        last = new Node();
        last.item = item;
        last.next = null;
        if (isEmpty()) {
            first = last;
        } else {
            oldlast.next = last;
        }
        N++;
    }

    public Item dequeue() {
        if (isEmpty()) return null;
        Item item = first.item;
        first = first.next;
        if (isEmpty()) last = null;
        N--;
        return item;
    }
```

```
    public Item front() {
        if (isEmpty()) return null;
        return first.item;
    }
}
```

3.1 Osservazioni

- Tutte le operazioni hanno complessità $\Theta(1)$. In particolare, per `enqueue` ciò è possibile grazie al riferimento `last`.
- L'operazione `dequeue` restituisce l'elemento rimosso (combina FRONT e DEQUEUE).