Chapter 1

Data Structures

SEQUENCE
boolean isEmpty();
boolean finished(Pos p);
Pos head();
Pos tail();
Pos next(Pos p);
Pos prev(Pos p);
Pos insert(Pos p, Item v);
Pos remove(Pos p);
Item read(Pos p);
write(Pos p , Item v);

SET
int = `size()`
boolean contains(ITEM x)
insert(Item x)
remvoe(Item x)
Set union(Set A, Set B)
Set intersection(Set A, Set B)
Set difference(Set A, Set B)

DICTIONARY
Item lookup(Item k)
insert(Item k, Item v)
remove(Item x)

Lista bidirezionale Lista bidirezinale con sentinella

1.1 Stack

Stack Una struttura dati dinamica, lineare in cui l'emento rimosso dall'operazione di cancellazione è predeterminato: "quello che per meno tempo è rimasto nell'insieme" (LIFO)

STACK
boolean isEmpty()
push(Item x)
Item pop()
Item top()

1.2 Queue

Queue Una struttura dati dinamica, lineare in cui l'emento rimosso dall'operazione di cancellazione è predeterminato: "quello che per più tempo è rimasto nell'insieme" (FIFO)

QUEUE
boolean isEmpty()
enqueue(Item x)
Item dequeue()
Item top()

1.3 Tree

Tree Data Structure Binary Tree Data Structure

TREE
Tree(Item v)
Item read()
write(Item x)
Item top()
Tree left()
Tree right()
insertLeft(Tree x)
insertRight(Tree x)
deleteLeft()
deleteRight()

Breath First Search Breath first search. Costo Computazionale $\Theta(n)$ in quanto ogni nodo viene visitato al massimo una volta.

```
dfs(Tree t)
if t != nil then
    /* Pre Order Visit                                    */
    print t
    /* In Order Visit                                     */
    dfs(t.left())
    print t
    /* Post Order Visit                                   */
    dfs(t.right())
    print t
```

Generic Tree Albero generico, non binario

```
Tree
Tree(Item v)
Item read()
write(Item x)
Tree parent()
Tree leftmostChild()
Tree rightSibling()
insertChild(Tree x)
insertSibling(Tree x)
deleteChild()
deleteSibling()
```

Depth-First Search Depth-First Search

```
dfs(Tree t)
if t != nil then
    /* Pre Order Visit                                    */
    print t
    Tree u = t.leftmostChild() while u != nil do
        dfs(u) u = u.rightSibling()
    /* Post Order Visit                                   */
    print t
```

Breadth-First Search Breadth First Search

```
bfs(Tree t)
    Queue Q = Queue() Q.enqueue(t)
    while Q.isEmpty() do
        Tree u = Q.dequeue() /* Visita per livello nodo u        */
        print t
        u = t.leftmostChild() while u != nil do
            Q.enqueue(u) u = u.rightSibling()
```

Chapter 2

Exams

2.1 04-09-2020

Longest Single Restituisce la lunghezza del più lungo sottovettore contiguo che non contiene valori duplicati

```
int longestSingle(int[] A, int n)
    Set S = Set()
    int maxSoFar = 0
    int start = 1
    int end = 1
    while end ≤ n do
        if S.contains(A[end]) then
            while A[start] != A[end] do
                S.remove(A[start])
                start = start + 1
            start = start + 1
        else
            S.insert(A[end])
            maxSoFar = max(maxSoFar, end - start + 1)
        end = end + 1
    return maxSoFar
```

2.2 24-07-2020

closeDuplicates Trova numeri duplicati in array raggio k

Tree Mirror Find if tree is Mirror

```
boolean isMirror(TREE t_L, TREE, t_R)
if t_L == nil and t_R == nil then
    return true
else
    if t_L != nil and t_R != nil then
        return isMirror(t_L.right, t_R.left) andisMirror(t_R.right, t_L.left)
    else
        return false
```

2.3 03-07-2020

BinaryInsert inserisce un array ordinato di numeri in un albero con T.size nei nodi. Poichè questo algoritmo è sostanzialmente una vista, la sua complessità sarà pari a $\Theta(n)$

```
biRec(TREE t, int[] A, int i)
int leftSize = 0
if T.left != nil then
    leftSize = t.left.size
    biRec(t.left, A, i)
t.value = A[i+leftSize]
if T.left != nil then
    biRec(t.left, A, i + leftSize + 1)
```

hasMajority Restituisce true se il vettore contiene un valore di maggioranza, ovvero un valore che compare più di $n/2$ volte. Il costo computazionale, dato dalla ricerca dicotomica, è $O(\log n)$.