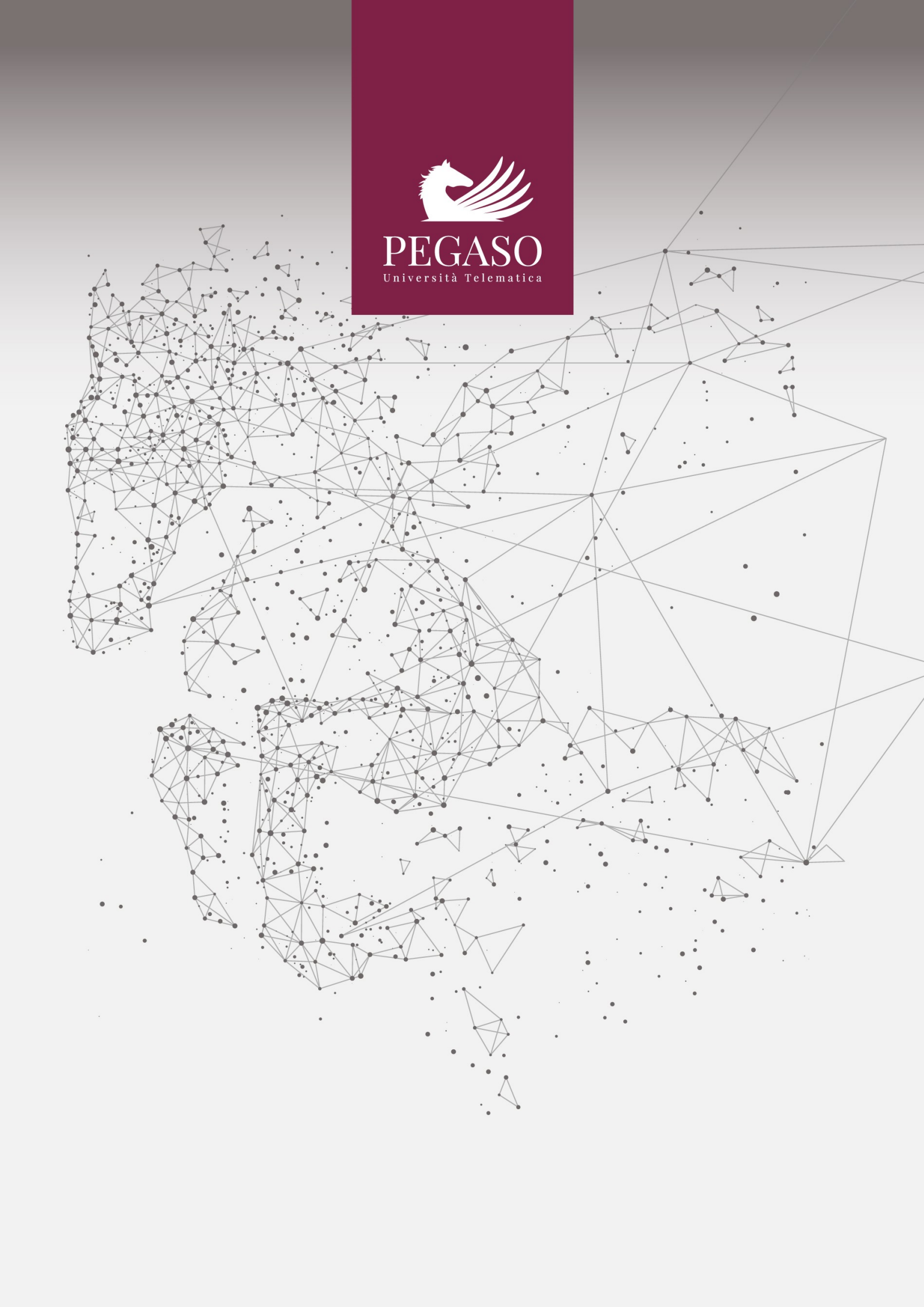

PEGASO
Università Telematica

Indice

1. Albero generico

Un albero binario è un albero radicato in cui ogni nodo ha al massimo due figli, identificati come figlio sinistro e figlio destro. In un albero generico, non abbiamo più il vincolo sul grado di un nodo, dunque possiamo avere un numero variabile di figli.

Di seguito illustriamo un generico albero:

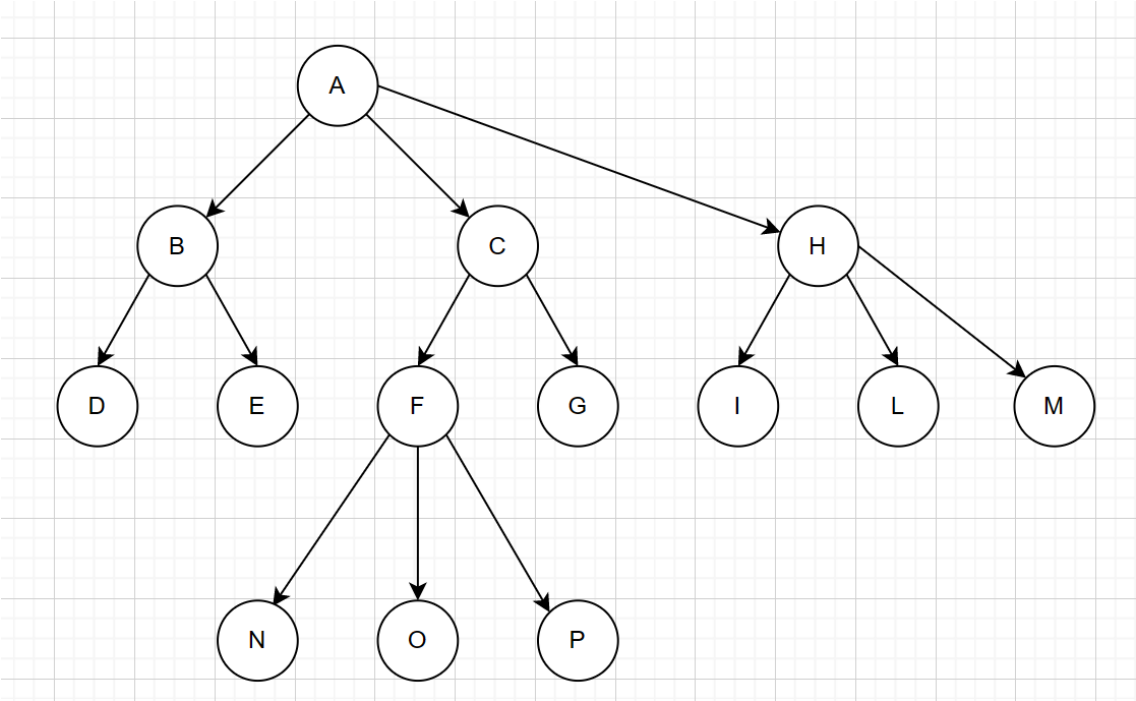


Figura 1 - Albero generico

Immaginiamo di avere una struttura dati che rappresenti il generico nodo dell'albero; tale struttura dovrà avere i seguenti campi:

- Valore memorizzato nel nodo (item)
- Riferimento al padre (parent)
- Riferimento all'array contenente i figli (children)

Nel momento in cui viene creato dunque un nuovo albero binario si dovrà creare un nodo "radice" che non avrà parent valorizzato (e continuerà a non averlo valorizzato anche successivamente essendo un nodo radice), avrà un item settato ed il puntatore all'array dei figli inizialmente nullo (successivamente si potranno "agganciare" i figli alla radice).

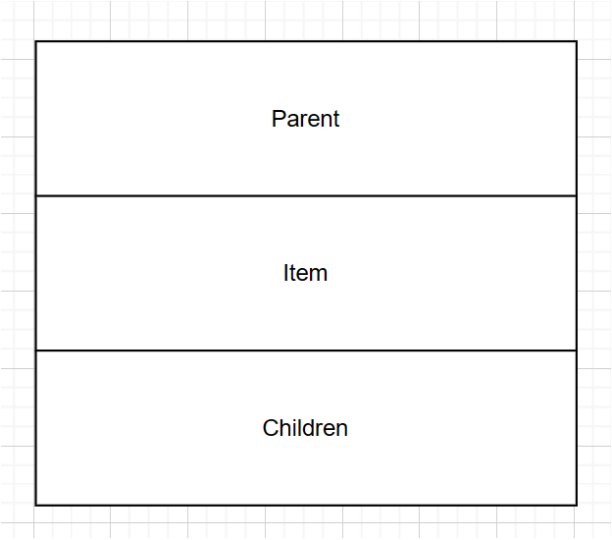


Figura 2 - Generico Nodo

Un generico nodo dell'albero dovrà avere dunque la seguente specifica:

- **`Node(Item v)`**: costruttore del nodo; inserisce l'item v, senza figli o genitori
- **`Item read()`**: legge il valore memorizzato nel nodo corrente
- **`write(Item v)`**: modifica il valore memorizzato nel nodo
- **`Tree parent()`**: restituisce il padre del nodo corrente oppure nil se questo nodo è radice
- **`Node leftmostChild()`**: restituisce il primo figlio sinistro oppure nil se questo nodo è una foglia
- **`Node rightSibling()`**: restituisce il prossimo fratello, oppure nil se assente
- **`insertChild(Node t)`**: Inserisce il sottoalbero radicato t come primo nodo di questo nodo
- **`insertSibling(Node t)`**: inserisce il sottoalbero radicato in t come prossimo fratello nodo di questo nodo
- **`deleteChild()`**: distrugge l'albero radicato identificato dal primo figlio
- **`deleteSibling()`**: distrugge l'albero radicato identificato dal prossimo fratello

2. Implementazione – albero generico

Il generico Nodo avrà dunque la seguente struttura:

```
struct node {
  int value;
  struct node *parent;
  struct node **children;
  int children_count;
};
```

La dichiarazione del nostro albero sarà:

```
struct node *root = NULL;
```

Relativamente all'inserimento, consideriamo il seguente codice:

```
void insert_child(struct node *parent, struct node *child) {
  child->parent = parent;
  parent->children_count++;
  parent->children = realloc(parent->children, sizeof(struct node*) *
parent->children_count);
  parent->children[parent->children_count - 1] = child;
}

struct node *insert_node(struct node *root, int value) {
  struct node *new_node = (struct node*) malloc(sizeof(struct node));
  new_node->value = value;
  new_node->children_count = 0;
  new_node->children = NULL;
  new_node->parent = NULL;

  if (root == NULL) {
    root = new_node;
  } else {
    insert_child(root, new_node);
  }

  return root;
}
```

Un popolamento dell'albero può essere il seguente:

```
struct node *root = NULL;
root = insert_node(root, 1);
root = insert_node(root, 2);
root = insert_node(root, 3);
insert_node(root->children[0], 4);
insert_node(root->children[0], 5);
insert_node(root->children[1], 6);
```

Utilizzando la seguente procedura di print:

```
void print_tree_graphical(struct node *root, int level) {
    if (root == NULL)
        return;
    printf("%*s%d\n", level * 2, "", root->value);
    for (int i = 0; i < root->children_count; i++) {
        for (int j = 0; j <= level * 2; j++) {
            printf(" ");
        }
        printf("|--");
        print_tree_graphical(root->children[i], level + 1);
    }
}
```

si determina la rappresentazione grafica di seguito:

```
1
|-- 2
  |--  4
  |--  5
|-- 3
  |--  6
```

Il seguente codice determinerebbe il risultato:

```
struct node *root = NULL;
root = insert_node(root, 1);
root = insert_node(root, 2);
root = insert_node(root, 3);
root = insert_node(root, 4);
root = insert_node(root, 5);
insert_node(root->children[0], 10);
insert_node(root->children[0], 11);
insert_node(root->children[0], 12);
insert_node(root->children[0], 13);
insert_node(root->children[0], 14);
print_tree_graphical(root,0);
```

```
1
 |--  2
   |--    10
   |--    11
   |--    12
   |--    13
   |--    14
 |--  3
 |--  4
 |--  5
```

3. Visita in ampiezza

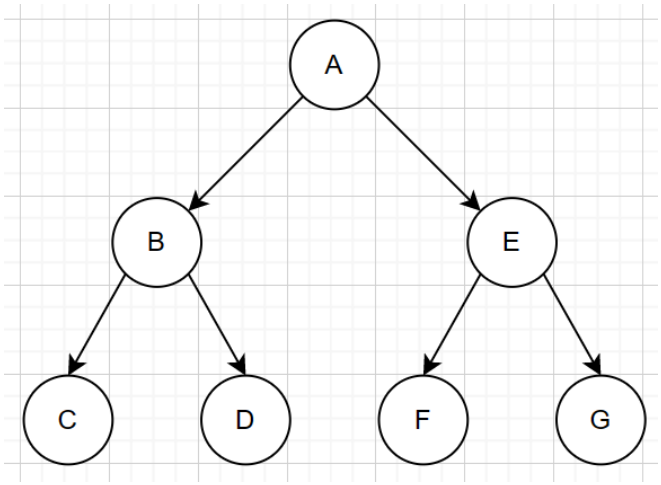
Di seguito l'implementazione in C della visita in BFS:

```
void BFS(struct node *root) {
  if (root == NULL) return;

  struct node *queue[100];
  int front = 0;
  int rear = 0;
  queue[rear++] = root;

  while (front != rear) {
    struct node *temp = queue[front++];
    printf("%d ", temp->data);
    if (temp->left != NULL) queue[rear++] = temp->left;
    if (temp->right != NULL) queue[rear++] = temp->right;
  }
}
```

Di seguito l'implementazione in Python della visita in BFS:

```
from collections import deque

def bfs(root):
    if not root:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.value)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

Facciamo un esempio concreto:



Essendo la radice "non nulla", viene inserita la radice nella coda; teniamo sotto controllo:

- **Sequenza**: NULL
- **Coda**: NULL

```
if not root:
     return
queue = deque([root])
```

Lo scenario diventa:

- **Sequenza**: NULL
- **Coda**: A

Finchè la coda non è vuota, viene eseguito il pop dalla coda e stampato il valore presente

```
while queue:
     node = queue.popleft()
     print(node.value)
```

Lo scenario diventa:

- **Sequenza**: A
- **Coda**: NULL

A questo punto, essendo presente un nodo a sinistra, questo viene inserito in coda; essendo poi presente un nodo a destra, anche questo viene inserito in coda:

```
if node.left:
     queue.append(node.left)
if node.right:
      queue.append(node.right)
```

Lo scenario diventa:

- **Sequenza**: A
- **Coda**: B, E

All'interno della coda sono presenti 2 elementi, pertanto:

```
while queue:
     node = queue.popleft()
     print(node.value)
```

Viene estratto il nodo B dalla coda e viene stampato il valore del nodo; lo scenario diventa:

- **Sequenza**: A, B
- **Coda**: E

Il nodo B che è stato estratto, ha un nodo a sinistra ed uno a destra; pertanto, si devono inserire tali nodi in coda:

```
if node.left:
     queue.append(node.left)
if node.right:
      queue.append(node.right)
```

Lo scenario diventa:

- **Sequenza**: A, B
- **Coda**: E, C, D

All'interno della coda sono presenti 2 elementi, pertanto:

```
while queue:
     node = queue.popleft()
```

```
    print(node.value)
```

Viene estratto il nodo E e stampato il suo valore; lo scenario diventa:

- **Sequenza**: A, B, E
- **Coda**: C, D

Il nodo E che è stato estratto, possiede un figlio a sinistra ed uno a destra:

```
if node.left:
    queue.append(node.left)
if node.right:
     queue.append(node.right)
```

Lo scenario diventa:

- **Sequenza**: A, B, E
- **Coda**: C, D, F, G

All'interno della coda sono presenti 4 elementi, pertanto:

```
while queue:
    node = queue.popleft()
    print(node.value)
```

Viene estratto il nodo E e stampato il suo valore; lo scenario diventa:

- **Sequenza**: A, B, E, C
- **Coda**: D, F, G

Il nodo C non ha figli, pertanto si procede nuovamente:

```
while queue:
    node = queue.popleft()
    print(node.value)
```

Viene estratto il nodo D e stampato il suo valore; lo scenario diventa:

- **Sequenza**: A, B, E, C, D
- **Coda**: F, G

Il nodo D non ha figli, pertanto si procede nuovamente:

```
while queue:
      node = queue.popleft()
      print(node.value)
```

Viene estratto il nodo F e stampato il suo valore; lo scenario diventa:

- **Sequenza**: A, B, E, C, D, F
- **Coda**: G

Il nodo F non ha figli, pertanto si procede nuovamente:

```
while queue:
      node = queue.popleft()
      print(node.value)
```

Viene estratto il nodo G e stampato il suo valore; lo scenario diventa:

- **Sequenza**: A, B, E, C, D, F, G

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley
- Sedgewick: Algoritmi in C, Pearson, 2015
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione