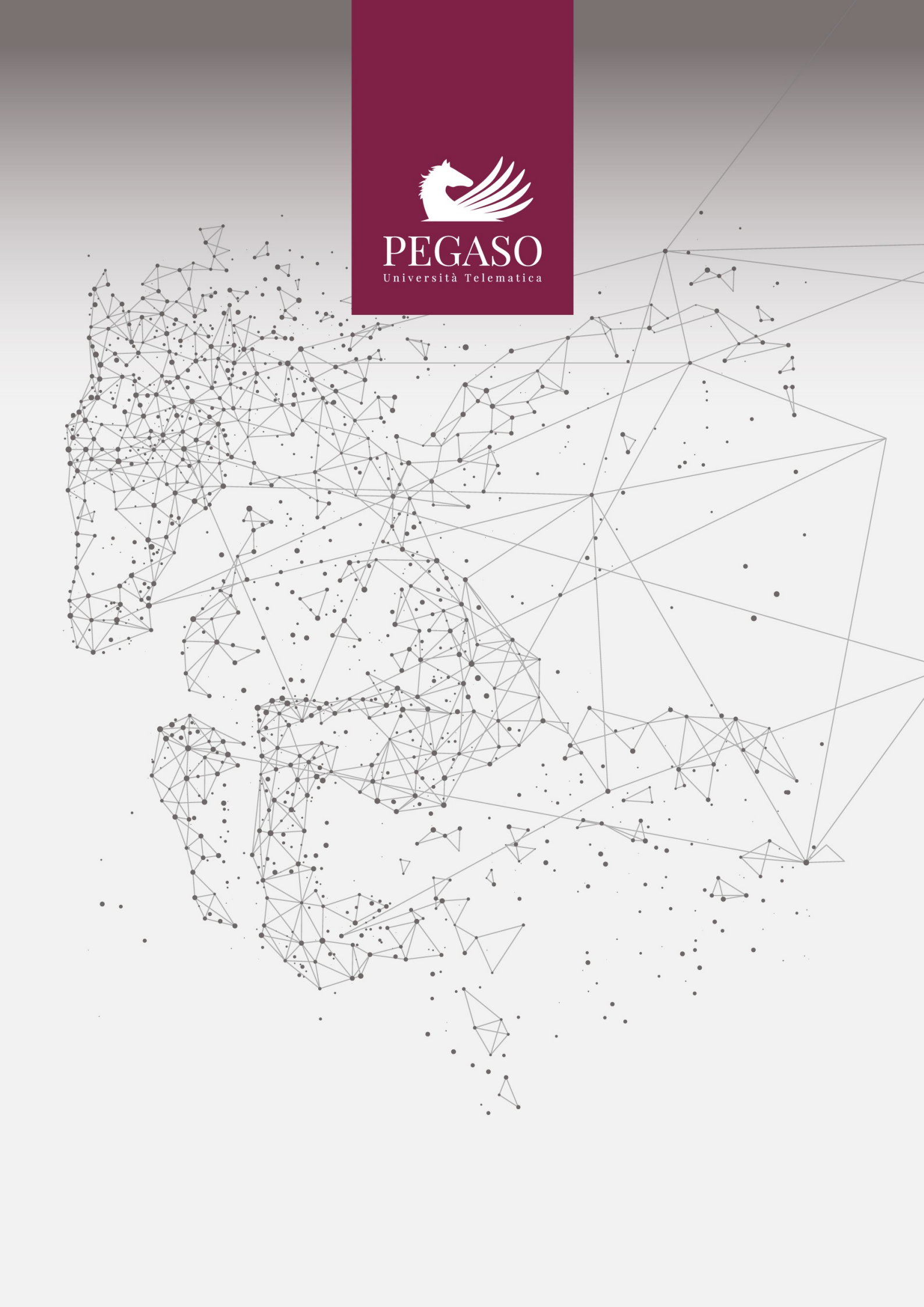
PEGASO
Università Telematica

Indice

1. Albero binario di ricerca

L'Albero Binario di Ricerca (ABR, oppure BST – Binary Search Tree in inglese) è un albero binario che soddisfa le seguenti proprietà:

- Ogni elemento ha una chiave (su cui è definito un ordinamento); due elementi non possono avere la stessa chiave (cioè le chiavi sono uniche);
- Le chiavi in un sottoalbero sinistro non vuoto devono essere più piccole della chiave nella radice dell'albero
- Le chiavi in un sottoalbero destro non vuoto devono essere più grandi della chiave nella radice dell'albero
- Anche i sottoalberi sinistro e destro sono alberi di ricerca binari

In altre parole, definito un ordinamento sull'informazione dei nodi (es: interi, stringhe, ma anche strutture complesse con un campo chiave), un albero binario è un albero binario di ricerca se per ogni nodo N dell'albero:

- L'informazione associata ad ogni nodo nel sottoalbero sinistro di N è strettamente minore dell'informazione associata ad N;
- L'informazione associata ad ogni nodo nel sottoalbero destro di N è strettamente maggiore dell'informazione associata ad N;

Di seguito alcuni esempi di ABR:

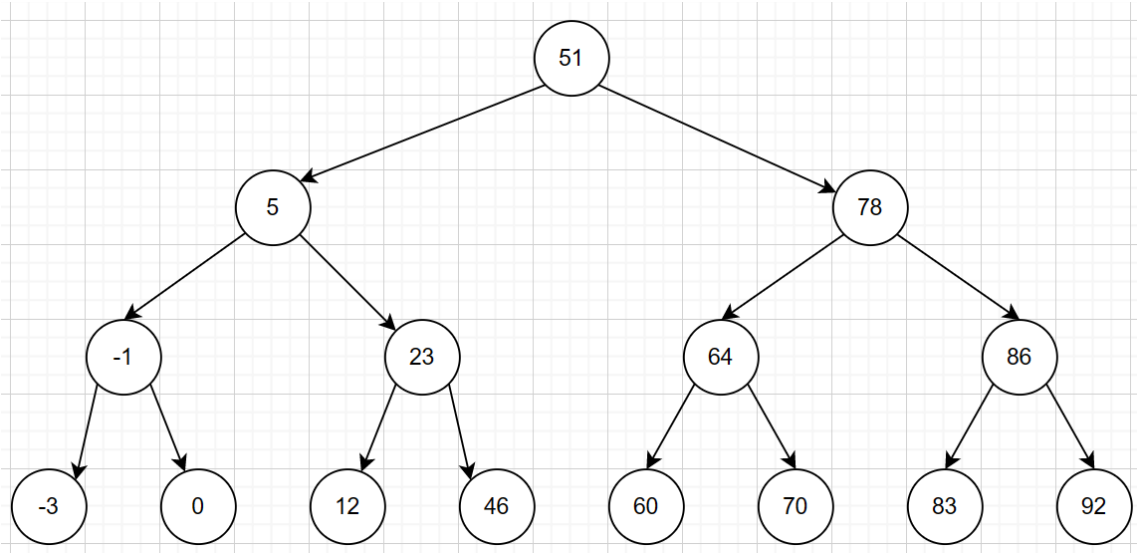

Figura 1: ABR

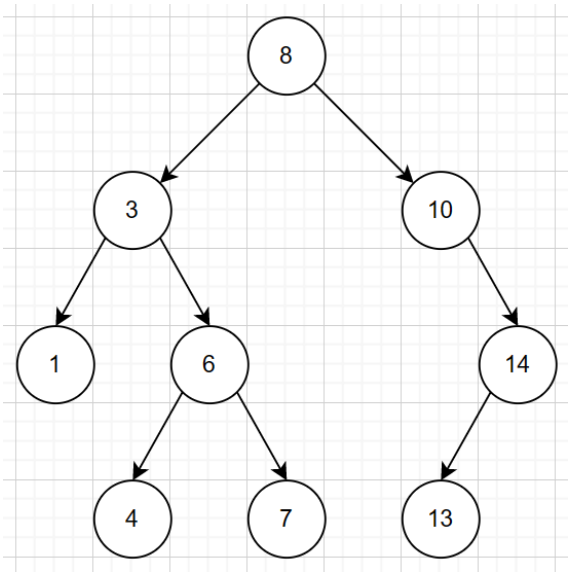

Figura 2: ABR

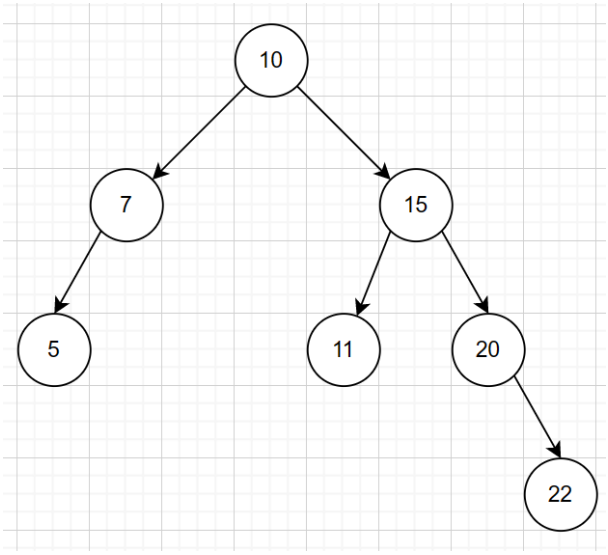

Figura 3: ABR

I seguenti sono esempio di alberi binari non di ricerca:

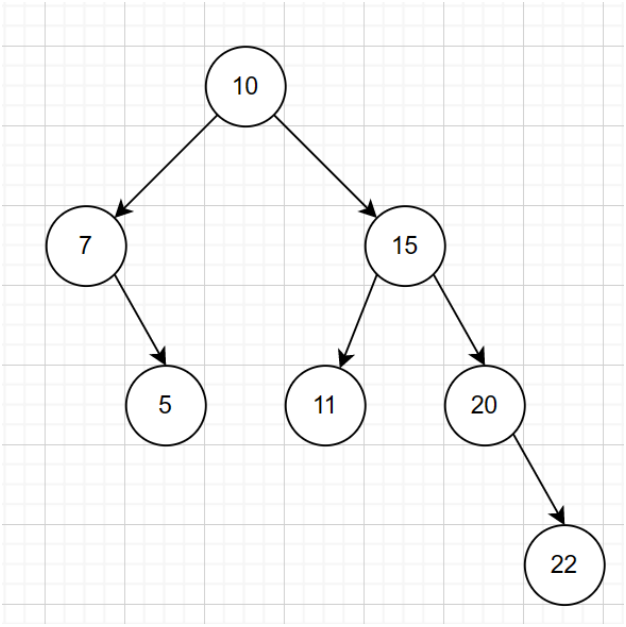

Figura 4: Albero binario non di ricerca

In questo caso il nodo 7 ha un figlio destro (5) che è minore del suo genitore (7) e quindi non è un ABR.

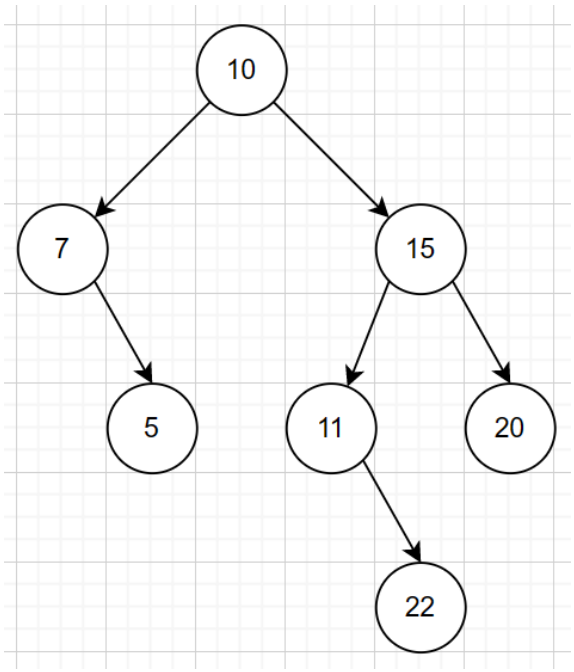

Figura 5: Albero binario non di ricerca

In tal caso il nodo 22 ha un genitore (11) più piccolo, ma il genitore del nodo 11 ha un padre (15) che è più piccolo di questo nodo (22) e questo non è possibile: il sottoalbero sinistro di 15 deve avere infatti tutti valori più piccoli di 15 (e quello destro tutti valori più grandi) ma ovviamente essendo 22>15 non viene rispettata tale condizione e quindi non è un ABR.

2. Implementazione di un ABR

Di seguito il codice in C per identificare se un albero binario è un ABR:

```
bool is_bst(struct node* node, int lower_bound, int upper_bound) {
 if (node == NULL) {
  return true;
 }
 if (node->data <= lower_bound || node->data >= upper_bound) {
  return false;
 }
 return is_bst(node->left, lower_bound, node->data) &&
      is_bst(node->right, node->data, upper_bound);
}
```

Il corrispondente codice in Python è il seguente:

```
def is_bst(node, lower_bound=-float("inf"), upper_bound=float("inf")):
   if node is None:
     return True
   if node.value <= lower_bound or node.value >= upper_bound:
     return False
   return (is_bst(node.left, lower_bound, node.value) and
         is_bst(node.right, node.value, upper_bound))
```

3. Operazioni su un ABR

Per **insieme dinamico** si intende un insieme il cui contenuto può essere modificato dinamicamente durante l'esecuzione di un programma. Questo termine viene comunemente utilizzato in programmazione per descrivere strutture dati come array, liste, set e mappe che possono essere modificate in modo efficiente durante l'elaborazione dei dati.

Un **dizionario** è un tipo di insieme dinamico che associa ad ogni elemento una chiave univoca: questa chiave è utilizzata per recuperare il valore associato a quell'elemento in modo efficiente. In altre parole, un dizionario è una struttura dati che implementa una mappa da chiavi a valori, dove entrambi possono essere di qualsiasi tipo. Poiché il contenuto di un dizionario può essere modificato dinamicamente, può essere considerato come un esempio di insieme dinamico.

In particolare, un dizionario implementa le seguenti funzionalità:

- **lookup**: ricerca di un elemento
- **insert**: inserimento di un nuovo elemento
- **remove**: cancellazione di un elemento

Lookup

Analizziamo l'operazione di **lookup**. Di seguito l'algoritmo di ricerca:

```
IF isEmpty(tree) THEN
        RETURN "elemento non presente"
ELSE IF (element == root(tree)) THEN
        RETURN "elemento trovato"
ELSE IF (element < root(tree)) THEN
RETURN lookup(tree.left)
ELSE
        RETURN lookup(tree.right)
```

Un albero (binario) si dice **bilanciato** se, detta P la profondità dell'albero, tutte le foglie stanno a livello P o P-1, e tutti i nodi interni hanno due figli, tranne al più un nodo al livello P-1 che ne ha uno solo:

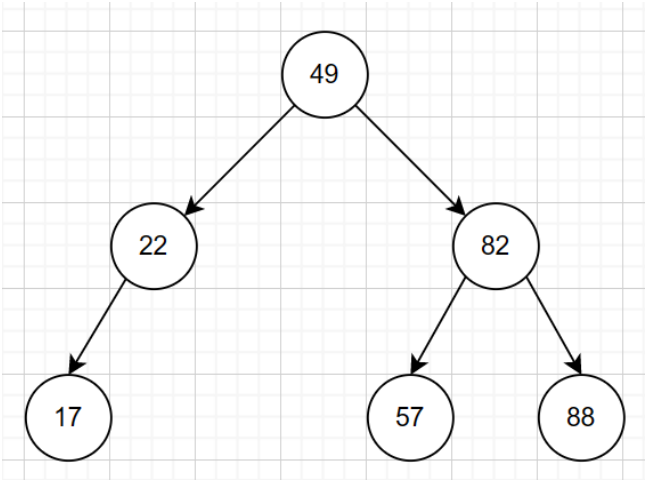

Figura 6: ABR bilanciato

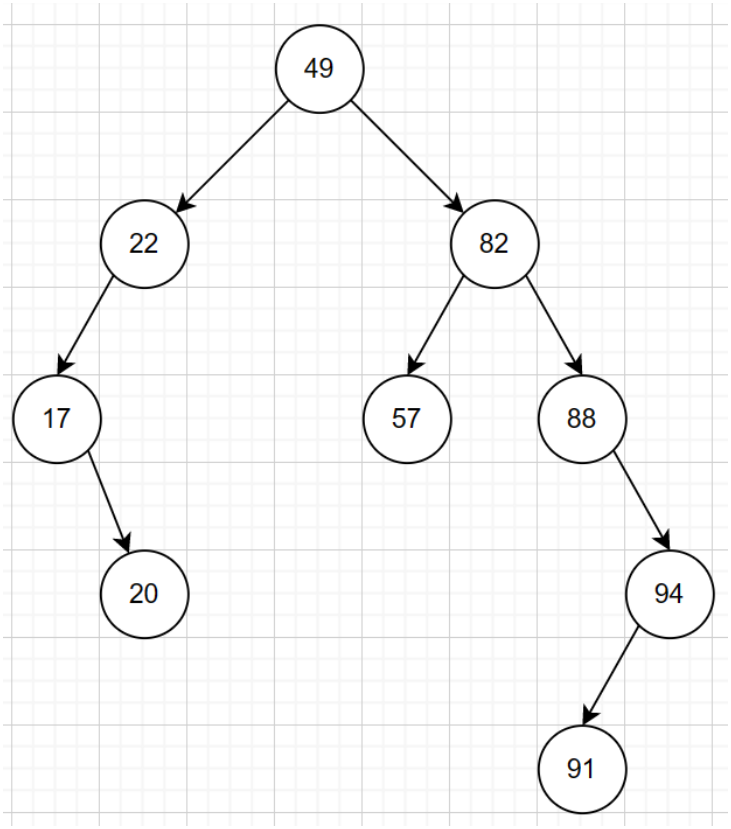

Figura 7: ABR non bilanciato

La ricerca di un elemento in un albero binario di ricerca è un'operazione molto efficiente (se l'albero è bilanciato). Se l'albero non fosse bilanciato ma "sbilanciato", cioè totalmente orientato verso un sottoalbero (sinistro o destro), la ricerca diventerebbe la ricerca di un elemento in una lista e quindi non ci sarebbero i benefici della ricerca "binaria".

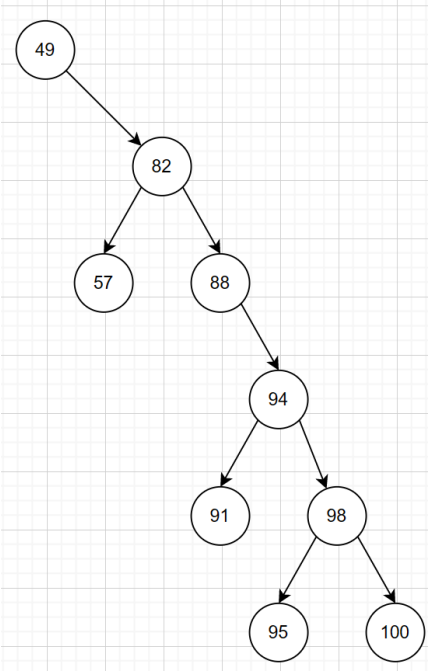

Figura 8: ABR sbilanciato

Dato dunque un ABR costituito da n nodi, si ha:

- nel caso peggiore l'albero degenera in una lista e dunque la complessità è lineare: $O(n)$
- nel caso medio dipende dalla distribuzione dei nodi e risulta ottimale se l'albero è bilanciato, ed in questo caso la complessità è logaritmica: $O\left(\log_2 n\right)$
- nel caso migliore la complessità è costante: $O(1)$

Per un ABR, la visita in-ordine risulta particolarmente interessante in quanto produce un elenco ordinato di chiavi. Ricordiamo che la visita in ordine simmetrico (in-order) consiste nella visita rispettivamente di:

- sottoalbero sinistro
- radice
- sottoalbero destro

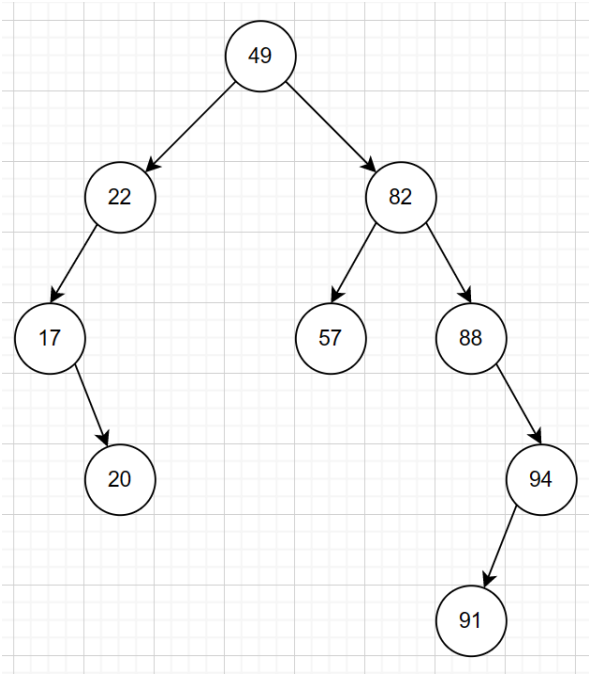

Figura 9: ABR

Visita simmetrica (in-ordine): 17, 20, 22, 49, 57, 82, 88, 91, 94

Riportiamo di seguito l'implementazione della procedura di ricerca in Python nella modalità iterativa e ricorsiva:

```
def search_iterative_bst(node, value):
  while node:
    if node.value == value:
      return node
    elif node.value < value:
      node = node.right
    else:
      node = node.left
  return None


def search_bst(node, value):
  if node is None:
    return None
  if node.value == value:
    return node
  if value < node.value:
```

```
    return search_bst(node.left, value)
  else:
    return search_bst(node.right, value)
```

Di seguito la versione in C:

```
struct node* search_bst(struct node *root, int key) {
  if (root == NULL || root->data == key) {
    return root;
  }
  if (root->data > key) {
    return search_bst(root->left, key);
  }
  return search_bst(root->right, key);
}

struct node* search_iterative_bst(struct node *root, int key) {
  while (root != NULL && root->data != key) {
    if (root->data > key) {
      root = root->left;
    } else {
      root = root->right;
    }
  }
  return root;
}
```

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.