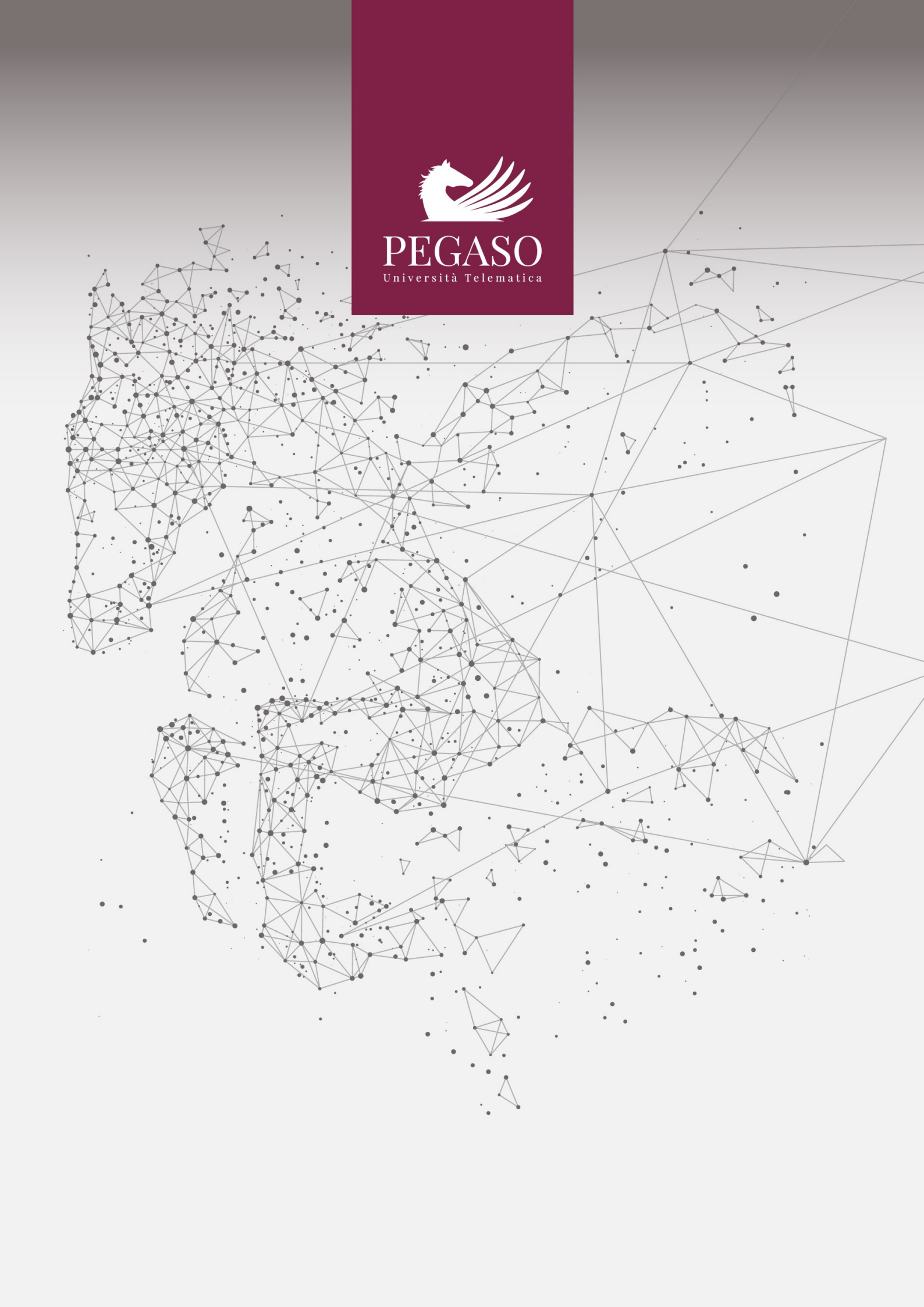
PEGASO
Università Telematica

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 un esempio di ABR:

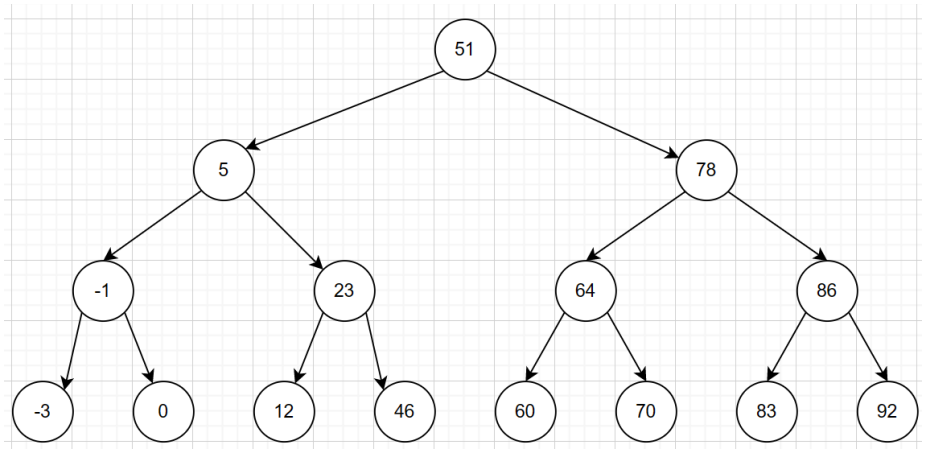

Figura 1: ABR

Per **insieme dinamico** si intende un insieme il cui contenuto può essere modificato dinamicamente durante l'esecuzione di un programma. 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

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".

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: $O(\log_2 n)$
- 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 e sottoalbero destro).

2. INSERIMENTO

Analizziamo ora la procedura di **inserimento** di un elemento X nell'ABR T:

```
IF T.isEmpty() THEN
T.createNode(x);
RETURN
ELSE IF x == T.root() THEN
        RETURN
ELSE IF x<T.root() THEN
        T.Insert(x, T.left())
ELSE
        T.Insert(x, T.right())
```

Supponiamo di avere il seguente ABR:

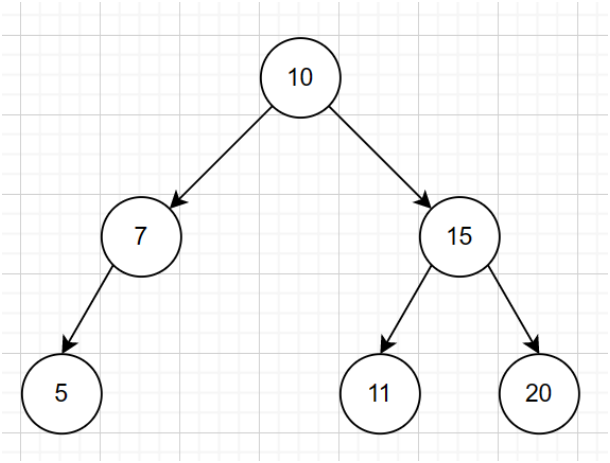

Figura 2: ABR

Se si volesse inserire il numero 6 i passi da seguire sarebbero:

- essendo l'ABR non vuoto si verifica se la radice (10) è uguale all'elemento da ricercare (6)
- essendo x<root (6<10) si procede con il sottoalbero sinistro
- essendo l'ABR non vuoto si verifica se la radice (7) è uguale all'elemento da ricercare (6)
- essendo x<root (6<7) si procede con il sottoalbero sinistro
- essendo l'ABR non vuoto si verifica se la radice (5) è uguale all'elemento da ricercare (6)
- essendo x>root (6>5) si procede con il sottoalbero destro
- essendo l'ABR vuoto, si procede con la creazione del nodo

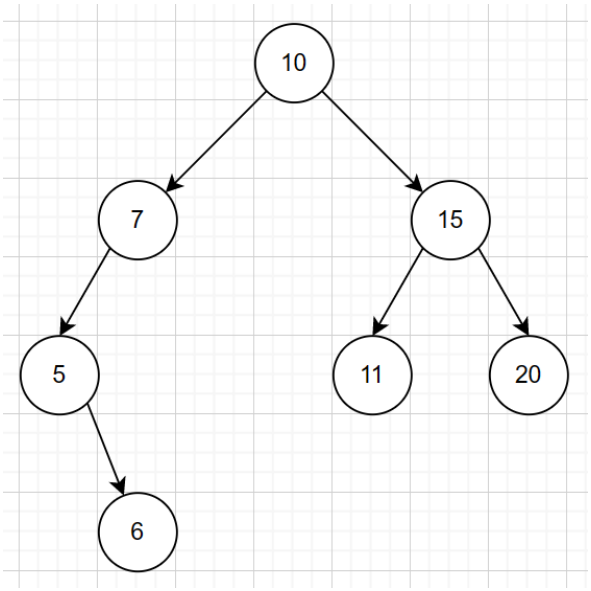

Figura 3: Inserimento in ABR

Di seguito l'implementazione in Python:

```
def insert_bst(root, value):
  if root is None:
    root = Node(value)
  else:
    if value < root.value:
      if root.left is None:
        root.left = Node(value)
      else:
        insert_bst(root.left, value)
    else:
      if root.right is None:
        root.right = Node(value)
      else:
        insert_bst(root.right, value)
```

Di seguito l'implementazione in C:

```
void insert_bst(struct node **root, int value) {
  if (*root == NULL) {
    *root = (struct node*) malloc(sizeof(struct node));
    (*root)->data = value;
```

```
    (*root)->left = NULL;
    (*root)->right = NULL;
   } else {
    if (value < (*root)->data) {
     insert_bst(&(*root)->left, value);
    } else {
     Insert_bst(&(*root)->right, value);
    }
   }
}
```

Generalizzando possiamo dire che l'inserimento di un elemento in un ABR prevede che ogni nuovo nodo venga inserito come foglia; possiamo distinguere 2 fasi:

- **fase 1**: ricerca il nodo genitore X; questa fase termina quando si raggiunge un nodo dell'ABR privo del figlio in cui avrebbe avuto senso continuare la ricerca: questo non deve necessariamente essere una foglia, è il nodo inserito che diviene foglia.
- **fase 2**: inserimento del nodo come figlio di X; tale nodo è una foglia

Nel caso peggiore:

- costo fase 1: $O(n)$
- costo fase 2: $O(1)$
- costo totale: $O(n)$

Nel caso medio (con una distribuzione uniforme):

- costo fase 1: $O \log_2 n$
- costo fase 2: $O(1)$
- costo totale: $O \log_2 n$

Ogni inserimento introduce una nuova foglia: il costo è (proporzionale a) la lunghezza del ramo radice-foglia interessato all'operazione e nel caso peggiore la complessità è dunque $O(n)$.

3. CANCELLAZIONE

Analizziamo ora la procedura di **cancellazione** di un elemento X nell'ABR T.

In questo caso occorre distinguere tra 3 scenari differenti:

1. il nodo da eliminare non ha figli (è una foglia)
2. il nodo da eliminare ha un figlio
3. il nodo da eliminare ha due figli

Consideriamo il seguente ABR:

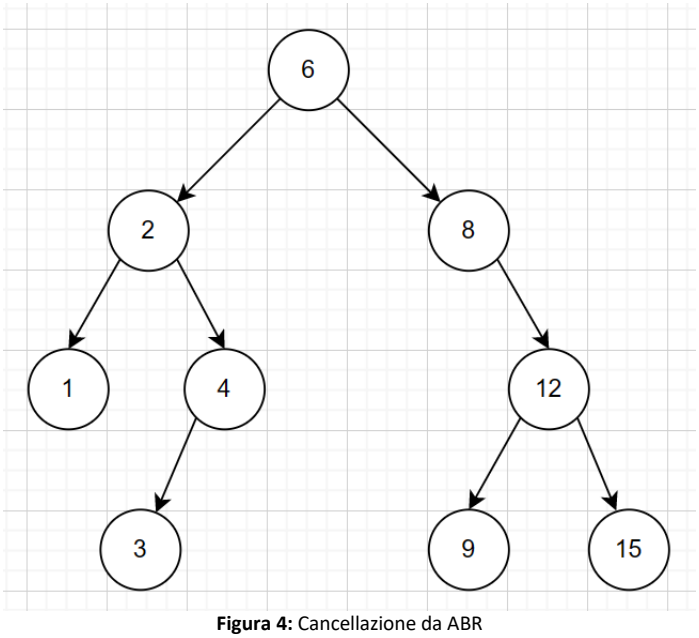

Figura 4: Cancellazione da ABR

Nel primo scenario, l'eliminazione di una foglia (1, 3, 9…) semplicemente comporterebbe l'eliminazione del nodo senza alcun problema.

Nel secondo scenario invece l'eliminazione del nodo 4 comporterebbe:

- eliminazione di 4
- creazione dell'arco dal (l'ex) padre di 4 (cioè 2) al figlio di 4 (cioè 3) → **short cut**

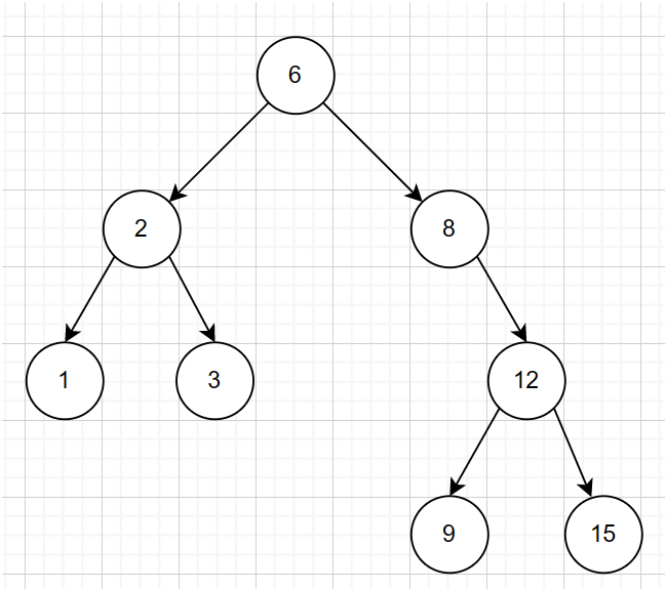

Figura 5: Cancellazione di nodo con un solo figlio

Nel terzo scenario invece, riprendendo l'ABR di fig. 4, l'eliminazione del nodo 2 comporterebbe:

- la ricerca del **successore** di 2 (in questo caso è 3)
- il distaccamento di 3 dall'ABR
- l'aggancio dell'eventuale figlio destro di 3 al padre di 3 (short cut); da precisare che può esserci solo un eventuale figlio destro e non sinistro, infatti se fosse presente, questo sarebbe il successore del nodo da eliminare
- il successore prende il posto del nodo da eliminare (2)

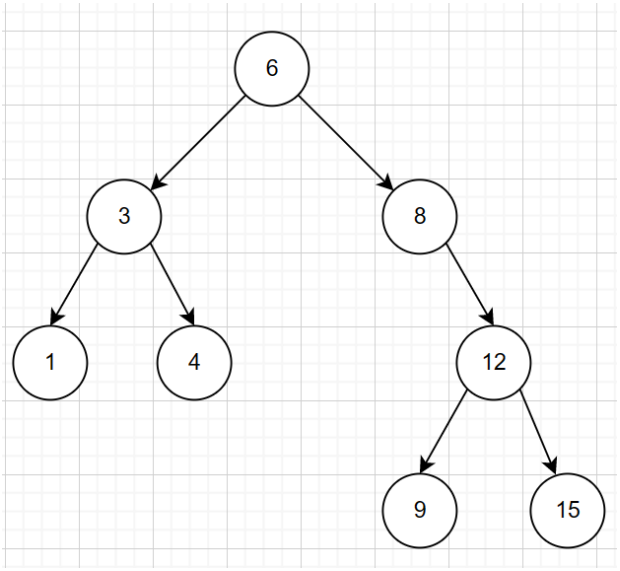

Figura 6: cancellazione da ABR

Riportiamo di seguito l'implementazione della procedura di eliminazione di un nodo da un ABR in linguaggio C tenendo conto del fatto che vi sono 3 casi da considerare a seconda della posizione del nodo e dei suoi figli:

- **nodo senza figli**: può essere cancellato semplicemente liberando la memoria con la funzione free() e impostando il puntatore del padre a NULL
- **nodo con un solo figlio**: può essere cancellato sostituendo il suo valore con quello del suo figlio unico e quindi cancellando il figlio
- **nodo con due figli**: può essere cancellato trovando il successore (il più piccolo nodo a destra) o il predecessore (il più grande nodo a sinistra) e sostituendo il suo valore con quello del successore/predecessore; quindi cancellare il successore/predecessore

```
struct node *minValueNode(struct node *node) {
 struct node *current = node;
 while (current && current->left != NULL) {
  current = current->left;
 }
 return current;
}

struct node *delete_bst(struct node *root, int data) {
 if (root == NULL) {
  return root;
 }
 if (data < root->data) {
  root->left = delete_bst(root->left, data);
 } else if (data > root->data) {
  root->right = delete_bst(root->right, data);
 } else {
  if (root->left == NULL) {
   struct node *temp = root->right;
   free(root);
   return temp;
  } else if (root->right == NULL) {
```

```
    struct node *temp = root->left;
    free(root);
    return temp;
   }
   struct node *temp = minValueNode(root->right);
   root->data = temp->data;
   root->right = delete_bst(root->right, temp->data);
  }
  return root;
 }
```

La complessità della procedura di eliminazione di un nodo in un albero di ricerca binario è $O(h)$, dove h è la altezza dell'albero. In media, l'altezza di un albero binario di ricerca equilibrato è logaritmica rispetto al numero di nodi, quindi la complessità media dell'eliminazione di un nodo è logaritmica. Tuttavia, in casi peggiori in cui l'albero non è equilibrato, l'altezza potrebbe essere lineare rispetto al numero di nodi, e di conseguenza la complessità potrebbe essere lineare anche in questi casi.

Ecco lo scenario del caso "peggiore":

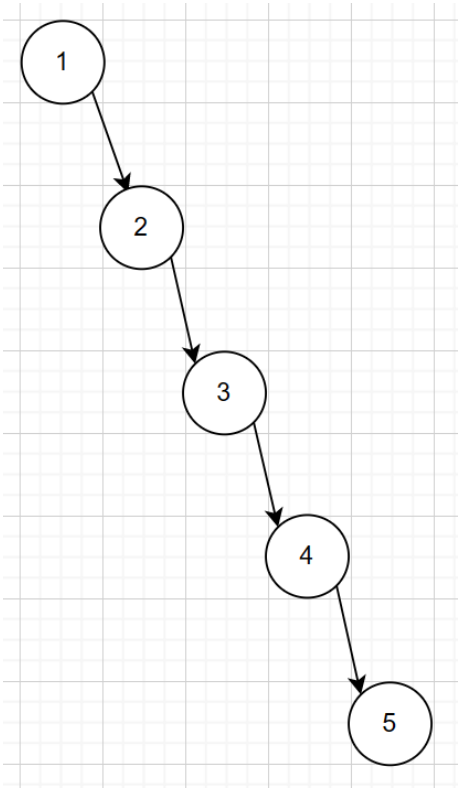

Figura 7: ABR: cancellazione nel caso peggiore

Lo scenario "migliore" si ha nel caso di albero "equilibrato", in cui la complessità diventa logaritmica: $O(\log_2 n)$:

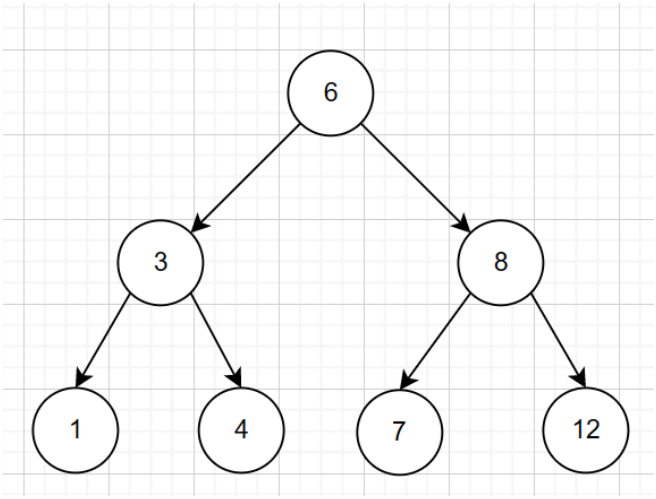

Figura 8: ABR bilanciato

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.