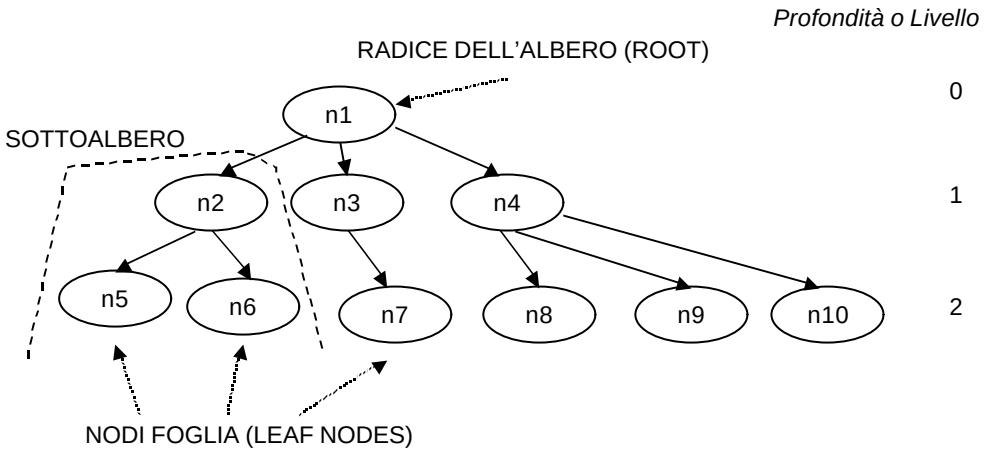
Alberi

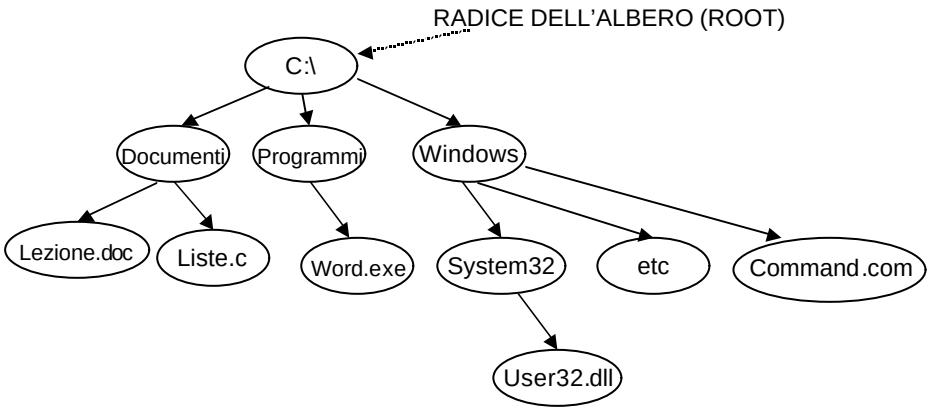
Sono strutture dati del tipo:



1

Alberi: Esempio di utilizzo

Rappresentazione di un file system:

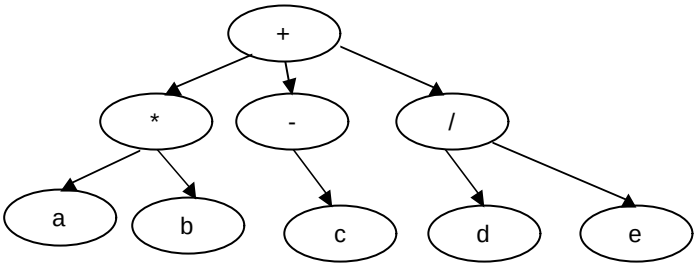


2

Alberi: Esempio di utilizzo

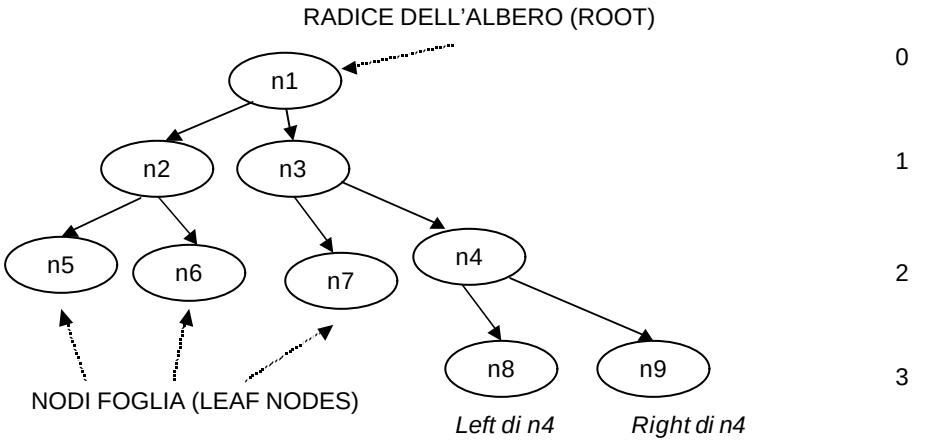
Rappresentazione (e valutazione) delle espressioni matematiche

(a * b) + (- c) + (d / e)



Alberi Binari

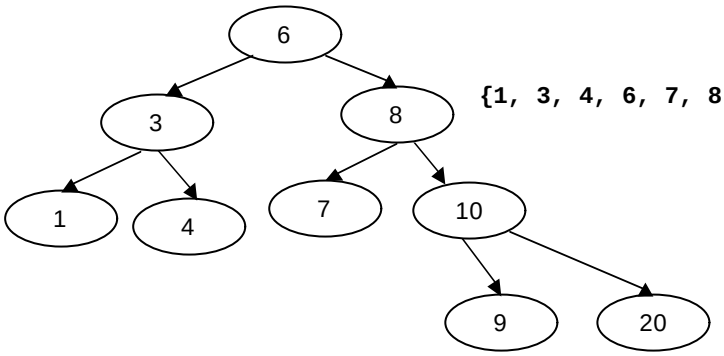
Sono alberi i cui nodi hanno 0, 1 o al più 2 figli:



Alberi Binari di Ricerca (ABR)

Sono alberi i cui nodi soddisfano alla relazione:

left (nodo) < nodo < right (nodo)

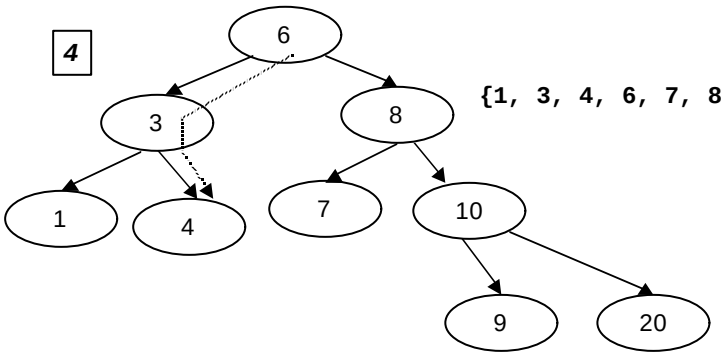


{1, 3, 4, 6, 7, 8, 9, 10, 20}

ABR: Ricerca di un elemento

1. N = radice
2. Se info_da_cercare == N allora TROVATO
3. Se info_da_cercare < N allora N = left (N), vai al passo 2
4. Se info_da_cercare > N allora N = right (N), vai al paso 2

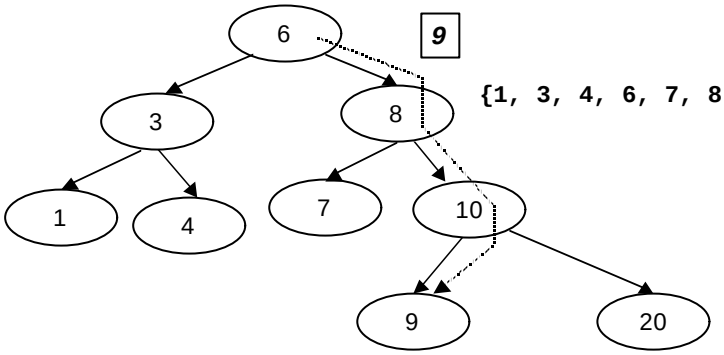
   Se N non ha left o right (nodo foglia), l'algoritmo si ferma con segnalazione di elemento non trovato



{1, 3, 4, 6, 7, 8, 9, 10, 20}

ABR: Ricerca di un elemento

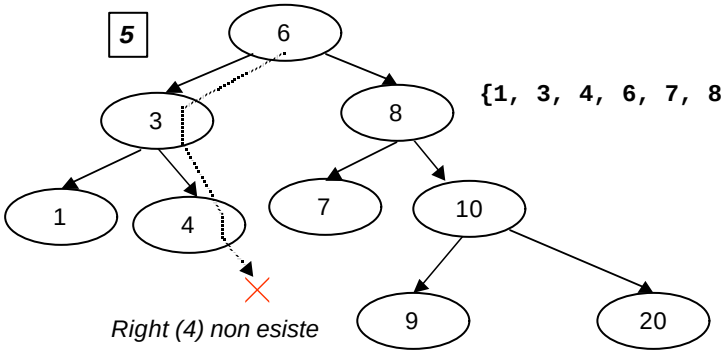
1. N = radice
2. Se info_da_cercare == N allora TROVATO
3. Se info_da_cercare < N allora N = left (N), vai al passo 2
4. Se info_da_cercare > N allora N = right (N), vai al paso 2
   Se N non ha left o right (nodo foglia), l'algoritmo si ferma con segnalazione di elemento non trovato



{1, 3, 4, 6, 7, 8, 9, 10, 20}

ABR: Ricerca di un elemento

1. N = radice
2. Se info_da_cercare == N allora TROVATO
3. Se info_da_cercare < N allora N = left (N), vai al passo 2
4. Se info_da_cercare > N allora N = right (N), vai al paso 2
   Se N non ha left o right (nodo foglia), l'algoritmo si ferma con segnalazione di elemento non trovato

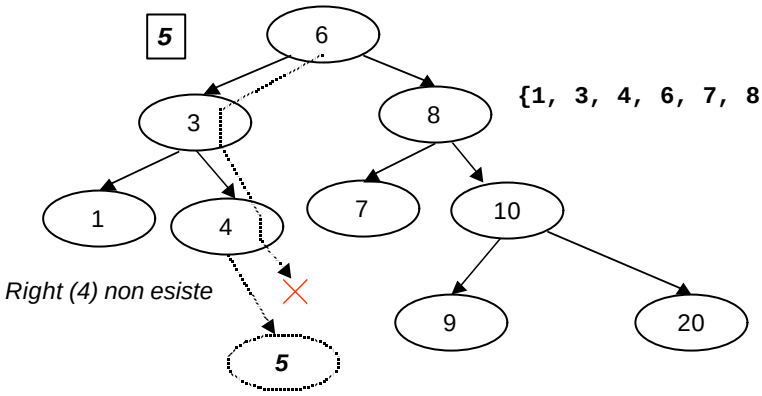


{1, 3, 4, 6, 7, 8, 9, 10, 20}

Right (4) non esiste

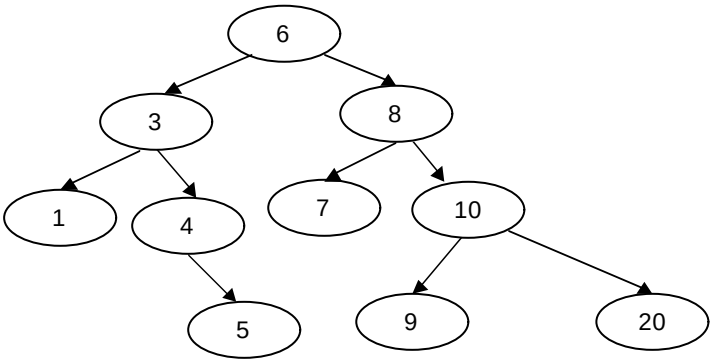
ABR: Inserimento di un elemento

1. Uso lo stesso algoritmo per la ricerca in modo da trovare il nodo foglia per l'inserimento
2. Inserisco l'elemento



ABR: Complessità

Numero di elementi del livello



2^0

2^1

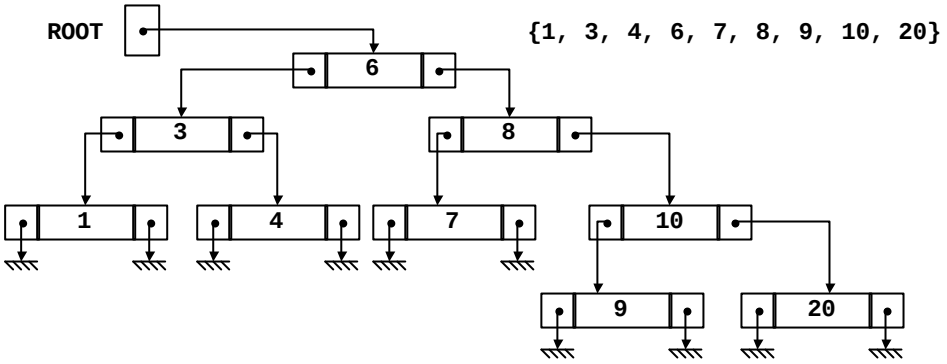
2^2

2^3

In media $N = 2^L$

$O(\log_2 N)$

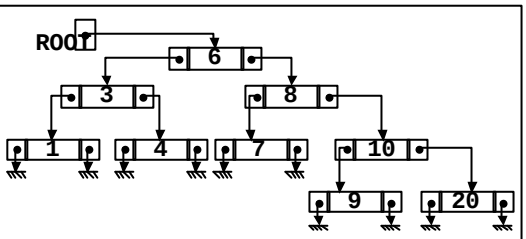
ABR: Struttura a puntatori



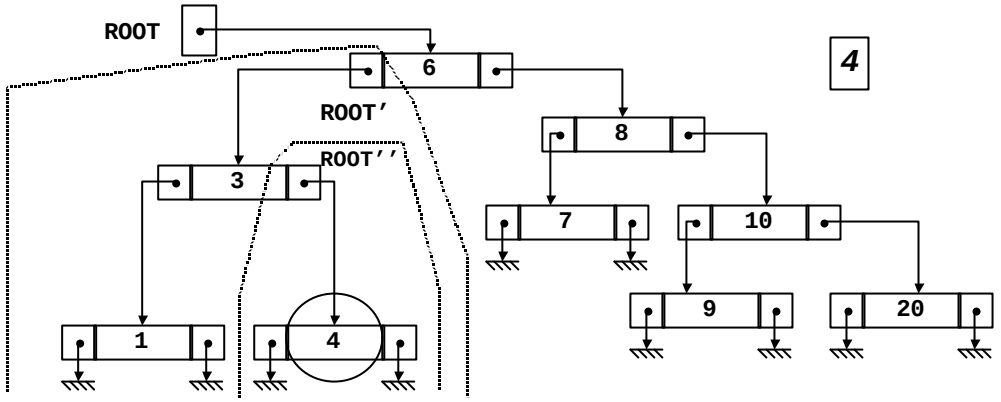

```
typedef struct t_node {
  int             data;
  struct t_node * left, * right;
} t_node;

typedef t_node *   t_tree;
```


ABR: Ricerca

```
typedef struct t_node {
  int             data;
  struct t_node * left, * right;
} t_node;

typedef t_node *   t_tree;


t_node * tree_find (t_node * root, int n)
{
  while (root != NULL) {
    if (n == root->data)
       return root;  // trovato!
    if (n < root->data)
       root = root->left;   // cerca a sinistra
    else                    // n > root->data
       root = root->right;  // cerca a destra
  }
  return NULL;
}
```

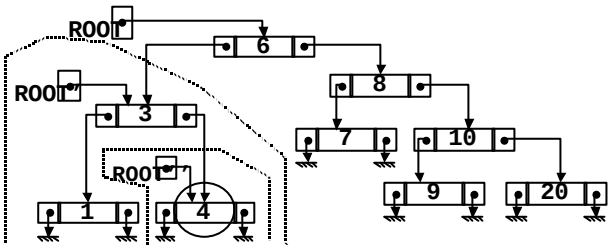


ABR: Ricerca ricorsiva



13

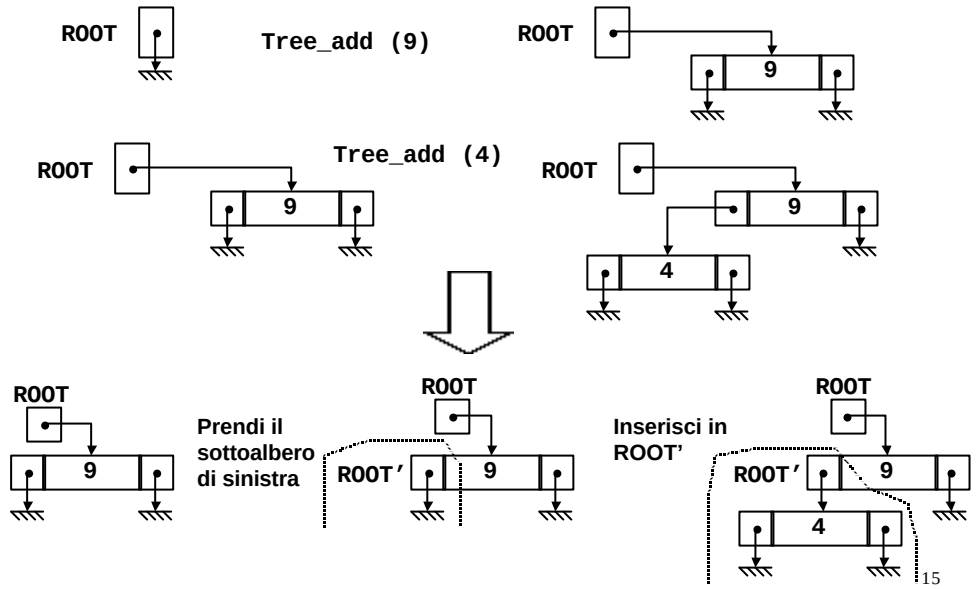
ABR: Ricerca ricorsiva



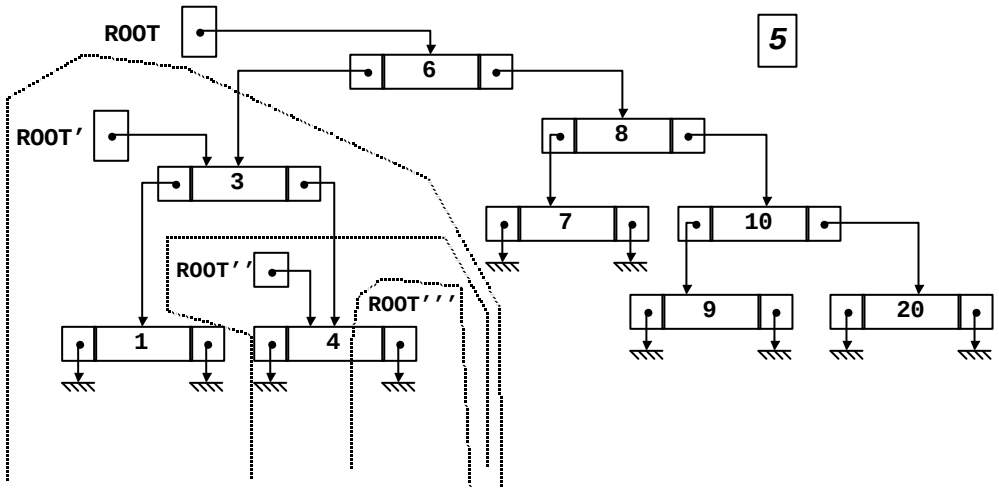

```
t_node * recursive_tree_find (t_tree root, int n)
{
  if (root == NULL)
    return NULL
  if (n == root->data)
     return root;  // trovato!
  if (n < root->data)
     return recursive_tree_find (root->left, n); // cerca a sx
  else     // n > root->data
     return recursive_tree_find (root->right, n); // cerca a dx
}
```

14

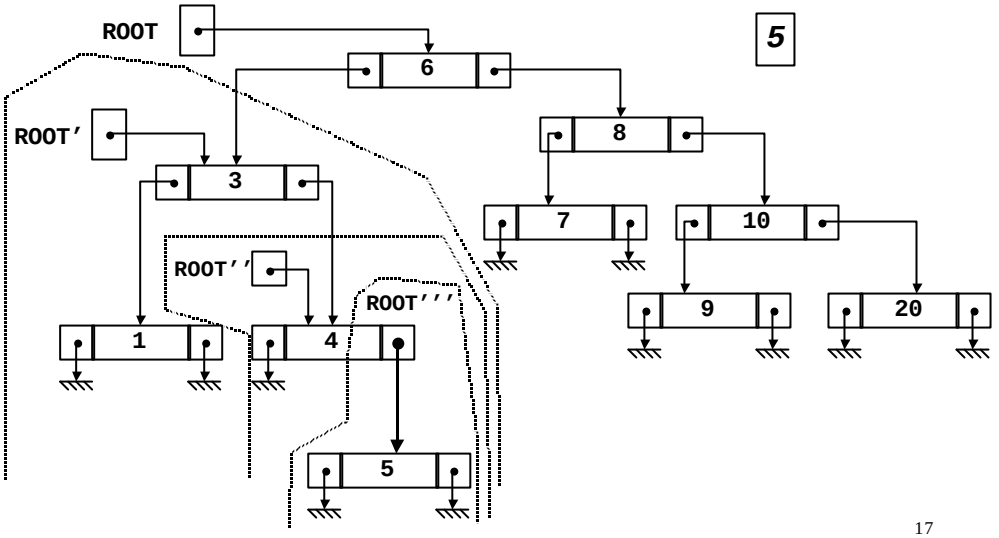
ABR: Inserimento



ABR: Inserimento



ABR: Inserimento



ABR: Inserimento (Ricorsivo)

```
void tree_add (t_tree * root, int n)
{
  if (*root == NULL) {
    t_node * newnode = (t_node *)malloc (sizeof (t_node));
    newnode->data = n;
    newnode->left = NULL;
    newnode->right = NULL;
    *root = newnode;
  }
  else {
    if (n == (*root)->data) {
      printf ("Elemento già esistente\n");
      return;
    }
    if (n < (*root)->data)
       tree_add (&((*root)->left), n);
    else                      // n > root->data
       tree_add (&((*root)->right), n);
  }
}
```

ABR con dati strutturati

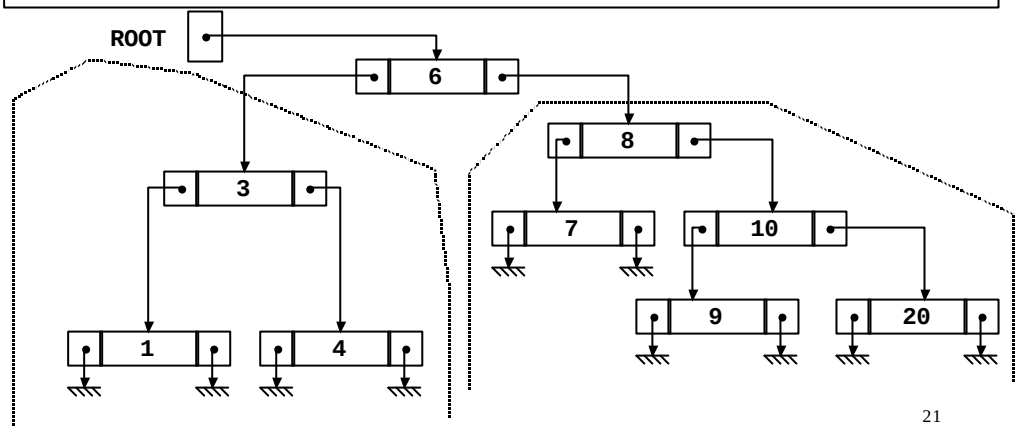
```
typedef struct {
  char key [20];
  char campo1 [40];
  ...
} t_element;

typedef struct t_node {
  t_element      data;
  struct t_node * left, * right;
} t_node;

typedef t_node *   t_tree;


t_tree init_tree (void)
{
  return NULL;
}
```


ABR: Ricerca ricorsiva

```
t_node * recursive_tree_find (t_tree root, char * chiave)
{
  if (root == NULL)
    return NULL
  if (strcmp (chiave, root->data.key) == 0)
     return root;  // trovato!
  if (strcmp (chiave, root->data.key) < 0)
     return recursive_tree_find (root->left, n); // cerca a sx
  else     // n > root->data
     return recursive_tree_find (root->right, n); // cerca a dx
}

...

t_tree mytree;
t_node * node_found;
node_found = recursive_tree_find (mytree, "hello")
if (node_found != NULL)
  printf ("Informazione trovata %s, %s\n",
          node_found->data.key, node_found->data.campo1);
```

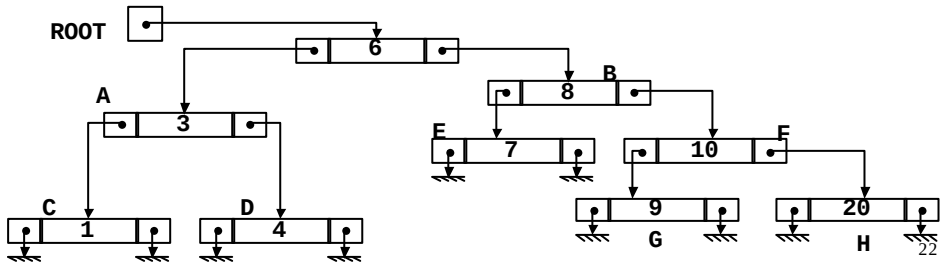
ABR: Visita in ordine

```
void visita (t_tree root)
{
  // visita (root->left);
  // stampa (root);
  // visita (root->right);
}
```




```
ROOT,
  A,
    C,        1
  A           3
    D,        4
ROOT,         6
  B,
    E,        7
  B,          8
    F,
      G,      9
    F,        10
      H,      20
```

```
void visita (t_tree root)
{
  // visita (root->left);
  // stampa (root);
  // visita (root->right);
}
```



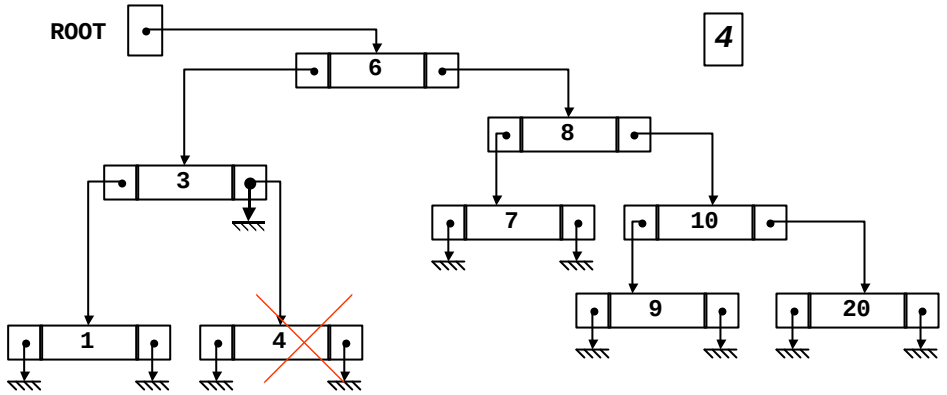
ABR: Visita in ordine

```
void inordine (t_tree root)
{
  if (root != NULL)
  {
    inordine (root->left);
    printf ("%s, %s\n", root->data.key, root->data.campo1);
    inordine (root->right);
  }
}
```


ABR: Visita in pre-ordine e post-ordine

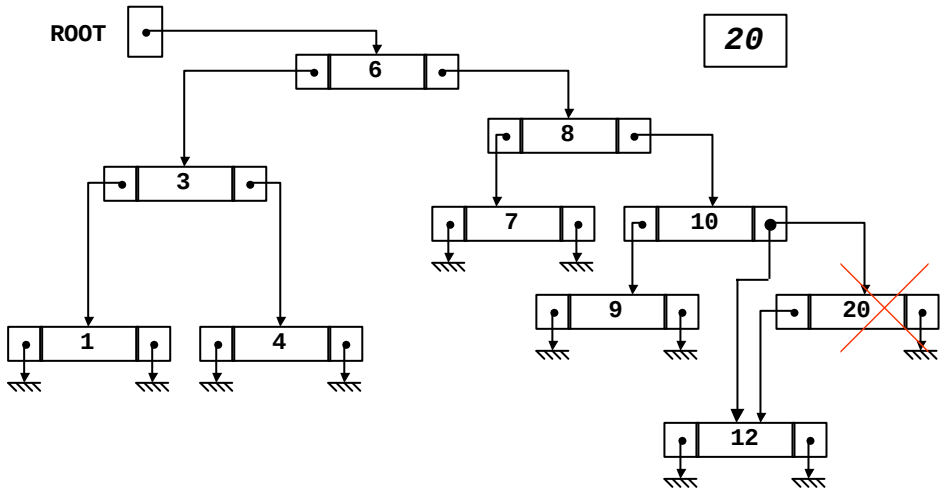
```
void preordine (t_tree root)
{
  if (root != NULL)
  {
    printf ("%s, %s\n", root->data.key, root->data.campo1);
    preordine (root->left);
    preordine (root->right);
  }
}

void postordine (t_tree root)
{
  if (root != NULL)
  {
    postordine (root->left);
    postordine (root->right);
    printf ("%s, %s\n", root->data.key, root->data.campo1);
  }
}
```

ABR: Cancellazione – caso 1 – leaf node



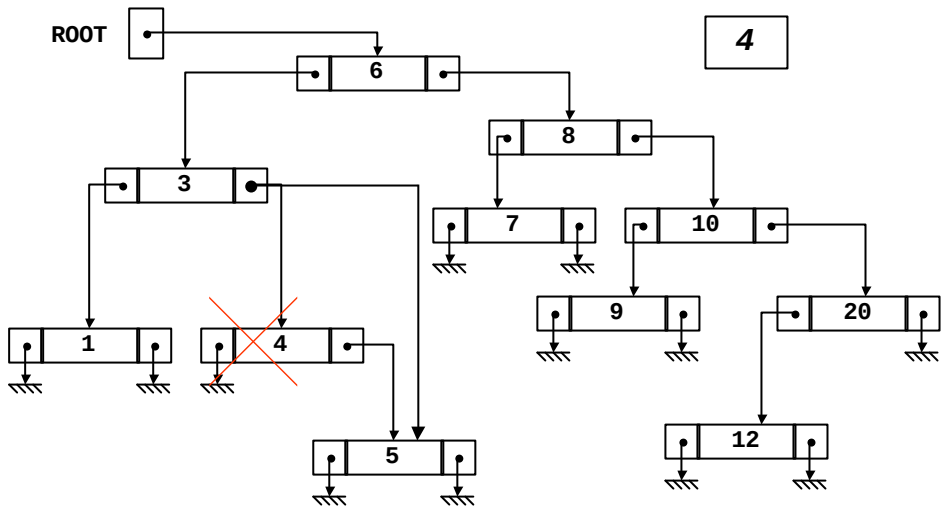
25

ABR: Cancellazione – caso 2 – non-leaf node



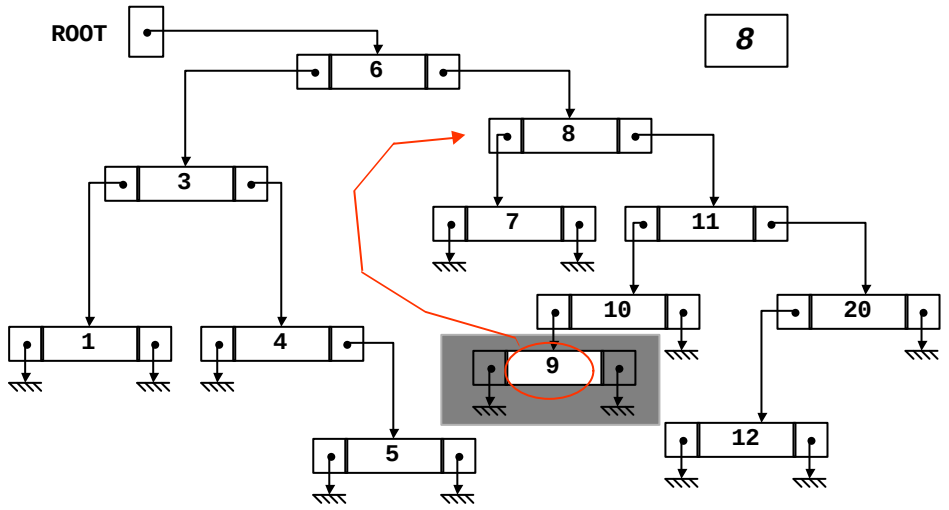
26

ABR: Cancellazione – caso 3 – non-leaf node



27

ABR: Cancellazione – caso 4 – non-leaf node



28

ABR: Cancellazione

```
int tree_delete (t_tree * root, char * key)
{
  t_node * aux;
  t_element data;

  if (*root == NULL)
    return 0;
  if (strcmp (key, (*root)->data.key)) < 0)
     return tree_delete (&(*root)->left, n); // cerca a sinistra
  else if (strcmp (key, (*root)->data.key)) < 0)
     return tree_delete (&(*root)->right, n);  // cerca a destra
...
```


ABR: Cancellazione

```
...
  else {
     if ( ((*root)->left == NULL) && ((*root)->right == NULL) ) {
       free (*root); *root = NULL;
     }
     else if ((*root)->left == NULL) {
       aux = *root;
       *root = (*root)->right; free (aux);
     }
     else if ((*root)->right == NULL) {
       aux = *root;
       *root = (*root)->left; free (aux);
     }
     else
       (*root)->data = find_min (&(*root)->right);
     return 1;
  }
}
```

ABR: Cancellazione

```
t_element find_min (t_tree * root)
{
  if ((*root)->left != NULL)
    return find_min (&(*root)->left);
  else {
    t_element data = (*root)->data;
    free (*root);
    *root = NULL;
    return data;
  }
}
```