Visite di alberi binari

Laboratorio di Algoritmi e Strutture Dati

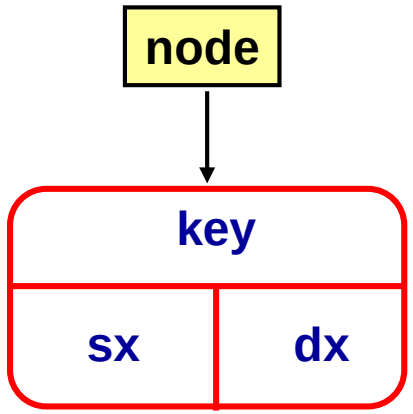
Visita di Alberi

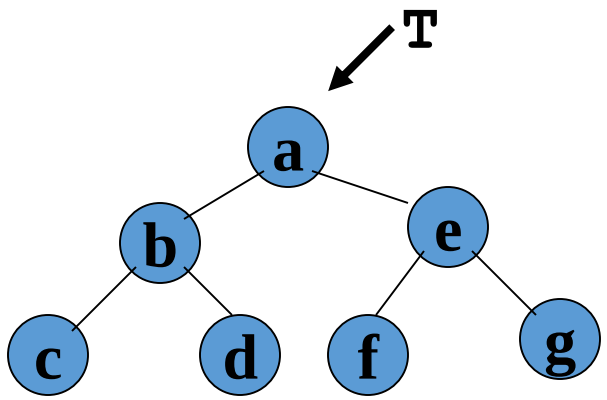
Gli alberi possono essere visitati (o attraversati) in diversi modi:

- ***Visita in Preordine: prima si visita il nodo e poi i suoi sottoalberi;***
- ***Visita Inordine (se binario): prima si visita il sottoalbero sinistro, poi il nodo e infine il sottoalbero destro;***
- ***Visita in Postordine : prima si visitano i sottoalberi, poi il nodo.***

Visita di Alberi Binari: in profondità preordine

```
Visita-Preordine(T)
    IF T ≠ NIL THEN
        "vista T"
        Visita-Preordine(T->sx)
        Visita-Preordine(T->dx)
```

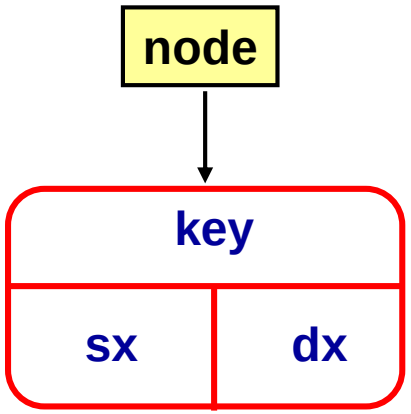


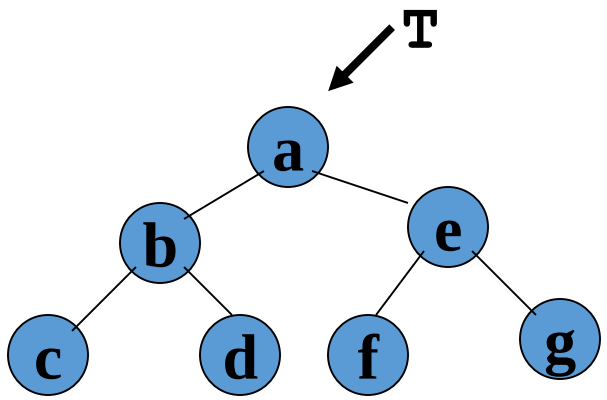


Sequenza: a b c d e f g

Visita di Alberi Binari: in profondità preordine

```
Visita-Preordine(node *T) {
    IF (T) {
        Vista(T);
        Visita-Preordine(T->sx);
        Visita-Preordine(T->dx);
     }
 }
```

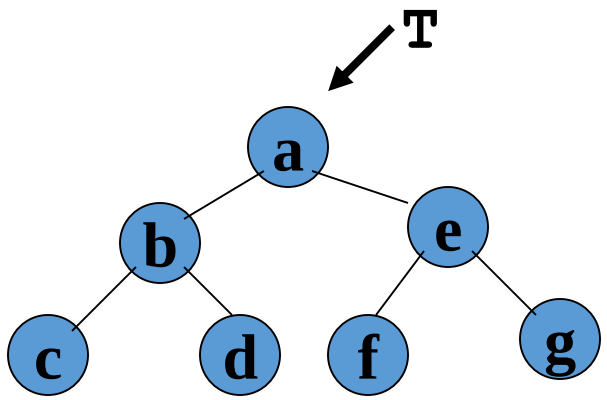




Sequenza: a b c d e f g

Visita di Alberi Binari: in profondità inordine

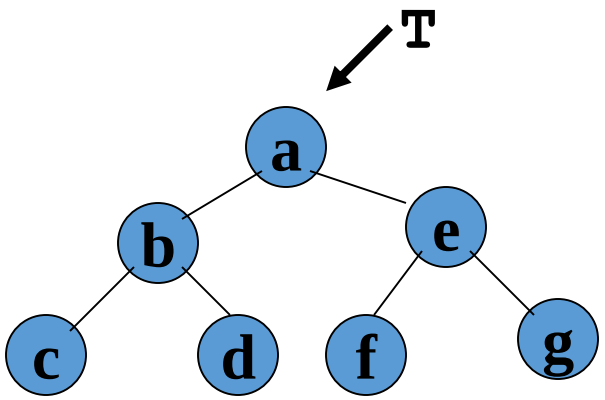
```
Visita-Inordine(T)
     IF T ≠ NIL THEN
       Visita-Inordine(T->sx)
       "vista T"
       Visita-Inordine(T->dx)
```



Sequenza: c b d a f e g

Visita di Alberi Binari: in profondità postordine

```
Visita-Postordine(T)
     IF T ≠ NIL THEN
       Visita-Postordine(T->sx)
       Visita-Postordine(T->dx)
       "vista T"
```



Sequenza: c d b f g e a

Visita Preorder Iterativa

```
Visita-preorder-iter(T)
    stack = NIL
    curr = T
    WHILE (stack ≠ NIL OR curr ≠ NIL) DO
        IF (curr ≠ NULL) THEN /* Discesa sx */
            "vista curr"
            push(stack,curr)
            curr = curr->sx
        ELSE          /* Risalita e discesa dx */
            curr = top(stack)
            pop(stack)
            curr = curr->dx
```

```
Visita-preorder-iter(node * T) {
     stack *st = NULL;
     node *curr = T;
   while(st || curr) {
        if(curr) {      /* Visita e discesa a sx */
              Visita(curr);
              st = push(st,curr);
              curr = curr->sx
        } else {                    /* Risalita */
              curr = top(st);
              st = pop(st);
              curr = curr->dx; /* Discesa a dx */
        }
    }
 }
```

Visita Inorder Iterativa

```
Visita-inorder-iter(T)
    stack = NIL
    curr = T
    WHILE (stack ≠ NIL OR curr ≠ NIL) DO
        IF (curr ≠ NULL) THEN /* Discesa sx */
             push(stack,curr)
             curr = curr->sx
        ELSE                          /* Risalita */
            curr = top(stack)
            pop(stack)
            "vista curr"
            curr = curr->dx /* Discesa dx */
```

```
Visita-inorder-iter(node * T) {
     stack *st = NULL;
     node *curr = T;
   while(st || curr) {
        if(curr) {   /* Discesa a sx */
              st = push(st,curr);
              curr = curr->sx
         }
        else {       /* Visita e discesa a dx */
              curr = top(st);
              st = pop(st);
              Visita(curr);
              curr = curr->dx; /* Discesa a dx */
        }
    }
}
```

Visita Postorder Iterativa

```
Visita-postorder-iter(T)
    stack = NIL
    curr = T, last = NIL
   WHILE (stack ≠ NIL OR curr ≠ NIL) DO
       IF (curr ≠ NIL) THEN          /* Discesa sx */
            push(stack,curr)
            curr = curr->sx
       ELSE
         curr = top(stack)              /* Risalita */
         IF (curr->dx ≠ NIL AND last ≠ curr->dx) THEN
              curr = curr->dx      /* Discesa dx */
         ELSE
              "vista curr"
              last = curr
              pop(stack)
              curr = NIL
```

```
Visita-postorder-iter(node * T) {
   stack *st = NULL;
   node *last, curr = T;
   while(st || curr) {
        if(curr) {                  /* Discesa a sx */
              st = push(st,curr);
              curr = curr->sx
         }
        else {                  /* Visita o discesa a dx */
            curr = top(st);
            if (curr->dx && last != curr->dx)
                  curr = curr->dx; /* Discesa a dx */
            else {
                  Visita(curr);
                  last = curr;
                  st = pop(st);
                  curr = NULL;
             }}
   }}
```

```
CONTA-NODI-ITER(T)
   stack = stack_a = NIL; curr = T; last = NIL; ret = 0
   WHILE (stack ≠ NIL OR curr ≠ NIL) DO
       IF (curr ≠ NIL) THEN   /* Discesa a sx */
            push(stack,curr)
            curr = curr->sx
            ret = 0      /* Azzera il valore di ritorno */
       ELSE
         curr = top(stack)    /* Risalita al padre */
         IF (curr->dx ≠ NIL AND last ≠ curr->dx) THEN /* Risalita da sx */
               a = ret                    /* Assegna il valore di ritorno ad a */
               push(stack_a,a)            /* Salva il valore di a nello stack */
               curr = curr->dx            /* Discesa a dx */
               ret = 0                    /* Azzera il valore di ritorno */
         ELSE          /* Risalita da dx oppure da sx con dx = NIL */'
             IF (curr->dx = NIL) THEN  /* Risalita da dx */
               a = ret                 /* Recupera il valore di a dallo stack */
               b = 0                   /* Copia il valore di ritono in b */
             ELSE                      /* Risalita da sx e dx = NIL */
               a = top(stack_a)        /* Copia il valore di ritono in a */
               b = ret                 /* Metti il risultato 0 in b */
             ret = a + b + 1           /* Somma a+1 al valore di ritorno in b */
             pop(stack); pop(stack_a)  /* Ripulisci gli stack */
             last = curr; curr = NIL   /* Prepara la risalita */
   return ret;
```

```
CONTA-NODI(T)
  ret = 0
  IF T ≠ NIL THEN
     a = CONTA-NODI(T->sx)
     b = CONTA-NODI(T->dx)
     ret = a + b + 1
  return ret
```

```
CONTA-NODI-ITER(T)
   stack = stack_a = NIL; curr = T; last = NIL; ret = 0
   WHILE (stack ≠ NIL OR curr ≠ NIL) DO
       IF (curr ≠ NIL) THEN   /* Discesa a sx */
            push(stack,curr)
            curr = curr->sx
            ret = 0      /* Azzera il valore di ritorno */
       ELSE
         curr = top(stack)    /* Risalita al padre */
         IF (curr->dx ≠ NIL AND last ≠ curr->dx) THEN /* Risalita da sx */
               a = ret                    /* Assegna il valore di ritorno ad a */
               push(stack_a,a)            /* Salva il valore di a nello stack */
               curr = curr->dx            /* Discesa a dx */
               ret = 0                    /* Azzera il valore di ritorno */
         ELSE          /* Risalita da dx oppure da sx con dx = NIL */'
            IF (curr->dx = NIL) THEN  /* Risalita da dx */
               a = ret                 /* Recupera il valore di a dallo stack */
               b = 0                   /* Copia il valore di ritono in b */
            ELSE                       /* Risalita da sx e dx = NIL */
               a = top(stack_a)        /* Copia il valore di ritono in a */
               b = ret                 /* Metti il risultato 0 in b */
            ret = a + b + 1            /* Somma a+1 al valore di ritorno in b */
            pop(stack); pop(stack_a)   /* Ripulisci gli stack */
            last = curr; curr = NIL    /* Prepara la risalita */
   return ret;
```

```
CONTA-NODI(T)
   ret = 0
   IF T ≠ NIL THEN
      a = CONTA-NODI(T->sx)
      b = CONTA-NODI(T->dx)
      ret = a + b + 1
   return ret
```

```
DUPLICA-ITER(T)
   stack = NIL; stack2 = NIL; curr1 = T; ret = NIL; last = NIL
   WHILE (stack ≠ NIL OR curr1 ≠ NIL) DO
       IF (curr1 ≠ NIL) THEN          /* Discesa a sx */
            curr2 = new_nodo(curr1->key)
            push(stack1,curr1)
            push(stack2,curr2)
            curr1 = curr1->sx; curr2 = NIL
       ELSE
         curr1 = top(stack1); curr2 = top(stack2)      /* Risalita */
         IF (curr->dx ≠ NIL AND last ≠ curr->dx) THEN /* Risalita da sx con dx ≠ NIL */
                                                  T'
                curr2->sx = ret;                  /* Collega padre al filgio sx nel duplicato */
                curr1 = curr1->dx
                curr2 = NIL
         ELSE   /* Risalita da dx oppure da sx con dx = NIL */
              IF (curr1->dx = NILL) THEN        /* Risalita da dx */
                   curr2->sx = ret              /* Assegna valore di ritorno a curr2->dx */
               ELSE                          /* Risalita da sx con dx = NIL */
                   curr2->dx = ret              /* Assegna valore di ritorno a curr2->sx */
               ret = curr2                      /* Memorizza valore di ritorno */
               pop(stack1); pop(stack2)
               last = curr1; curr1 = NIL        /* Prepara la risalita
   return ret
```

```
DUPLICA(T)
  T' = NIL
  IF T ≠ NIL THEN
    T' = new_nodo(T->key)
    T'->sx = DUPLICA(T->sx)
    T'->dx = DUPLICA(T->dx)
  return T'
```

```
DUPLICA-ITER(T)
   stack = NIL; stack2 = NIL; curr1 = T; ret = NIL; last = NIL
   WHILE (stack ≠ NIL OR curr1 ≠ NIL) DO
        IF (curr1 ≠ NIL) THEN          /* Discesa a sx */
              curr2 = new_nodo(curr1->key)
              push(stack1,curr1)
              push(stack2,curr2)
              curr1 = curr1->sx; curr2 = NIL
        ELSE
          curr1 = top(stack1); curr2 = top(stack2)       /* Risalita */
          IF (curr->dx ≠ NIL AND last ≠ curr->dx) THEN /* Risalita da sx con dx ≠ NIL */
                curr2->sx = ret;                  /* Collega padre al filgio sx nel duplicato */
                curr1 = curr1->dx
                curr2 = NIL
          ELSE   /* Risalita da dx oppure da sx con dx = NIL */
               IF (curr1->dx = NILL) THEN          /* Risalita da dx */
                     curr2->sx = ret               /* Assegna valore di ritorno a curr2->dx */
                ELSE                          /* Risalita da sx con dx = NIL */
                     curr2->dx = ret               /* Assegna valore di ritorno a curr2->sx */
                ret = curr2                        /* Memorizza valore di ritorno */
                pop(stack1); pop(stack2)
                last = curr1; curr1 = NIL          /* Prepara la risalita
   return ret
```

```
DUPLICA(T)
   T' = NIL
   IF T ≠ NIL THEN
      T' = new_nodo(T->key)
      T'->sx = DUPLICA(T->sx)
      T'->dx = DUPLICA(T->dx)
   return T'
```

T'

Esercizio 3

- Si realizzi una libreria per la gestione di ***Alberi Binari di Ricerca Generici***, che offra le stesse funzionalità della libreria all'***Esercizio 2***, ma realizzando tutte le funzioni in maniera ***iterativa***.
- Le funzioni iterative realizzate devono esibire ***lo stesso livello di efficienza*** di quelle ricorsive realizzate per l'Esercizio 2.
- Al fine di realizzare alcune delle funzionalità richieste per la libreria, sarà necessario modificare opportunamente gli algoritmi di visita descritti nei lucidi presentati durante questa lezione.