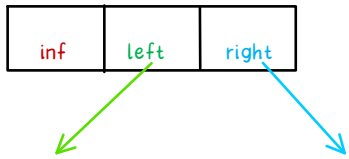
3.2-Alberi Binari

– Gli **alberi binari** sono un caso particolare di albero nario dove il numero dei figli per ogni nodo è al più **due.**

– La struttura dati per rappresentare i nodi degli alberi mantiene un riferimento esplicito al **figlio sinistro** e al **figlio destro.**

Ciascun **elemento** (node) contiene:
- un **campo informativo**
- un **puntatore al figlio di sinistra** (left)
- un **puntatore al figlio di destra** (right)



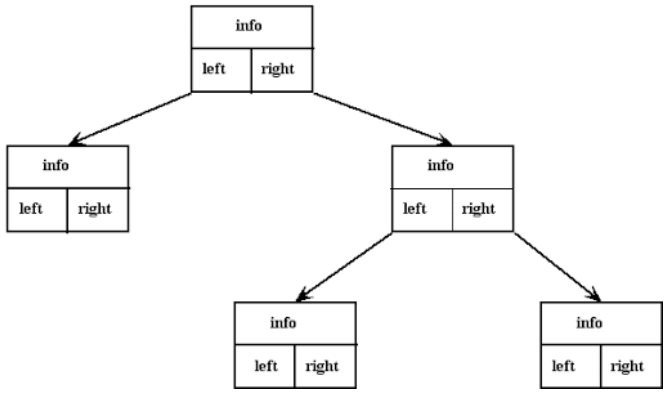
Nodo Albero

```
struct bnode {

    tipo_inf inf;
    bnode* left;
    bnode* right;
    bnode* parent; //opzionale

};

typedef bnode* btree;
```



Un **albero binario di ricerca** (binary search tree **BST**) è un albero binario che soddisfa le seguenti proprietà:

- Ogni nodo n ha :
  - un contenuto informativo **value(n)**;
  - una chiave **key(n)** presa da un dominio totalmente ordinato (ovvero su cui è definita una relazione d'ordine totale <);
    - Sia **n'** un nodo nel **sottoalbero sinistro** di n allora key (n') ≤ key(n).
    - Sia **n'** un nodo nel **sottoalbero destro** di n allora key (n') > key(n).

Nodo Albero BST

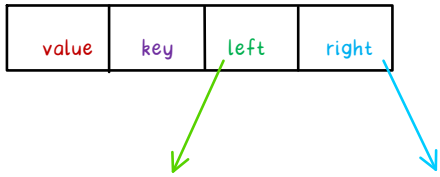
```
typedef int tipo_key;
typedef char* tipo_inf;

struct bnode {

    tipo_key key;
    tipo_inf inf;
    bnode* left;
    bnode* right;
    bnode* parent; //opzionale

};

typedef bnode* bst;
```



Primitive

NEW_NODE

```
bnode* bst_newNode(tipo_key k, tipo_inf i){

    bnode* n = new_bnode;
    copy(n->inf,i);
    copy(n->key,k)
    n->right = n->left = n->parent = NULL;

    return n;
}
```

GET_KEY

```
tipo_key get_key(bnode* n){

    return n->key;
}
```

GET_VALUE

```
tipo_inf get_value(bnode* n){

    return n->value;

 }
```

GET_LEFT

```
bst get_left(bst t){

    return t->left;
}
```

GET_RIGHT

```
bst get_right(bst t){

    return t->right;
}
```

GET_PARENT

```
bnode* get_parent(bnode* n){

    return n->parent;
}
```

Primitiva BST_INSERT

-Funzione che aggiunge un nodo all'albero di ricerca.

Versione Iterativa

```
void bst_insert(bst &b, bnode* n){

    bnode* x;
    bnode* y;

    if(b == NULL)
        b = n;

    else{
        x = b;

        while(x != NULL){

            y = x;

            if(n->key < x->key)
                x = get_left(x);
            else
                x = get_right(x);
        }

    n->parent = y;

    if(n->key < y->key)
        y->left = n;
    else
        y->right = n;
    }
 }
```

Versione Ricorsiva

```
void bst_insert(bst &b, bnode* n){

 if(b==NULL){
    b=n;
    return;
  }

 if(compare_key(get_key(n),get_key(b))<0){

    if(get_left(b)!=NULL)
        bst_insert(b->left,n);

    else {
        b->left=n;
        n->parent=b;
        }
  }

 else{

     if(get_right(b)!=NULL)
        bst_insert(b->right,n);

     else {
        b->right=n;
        n->parent=b;
        }
  }
}
```

Come funziona BST_insert?

Partendo dalla radice, il percorso di scansione dipende dall'esito del confronto tra il nodo corrente z e il nodo da inserire **n: key(x) e key(n)**

- Se key(n) < key(x) la scansione prosegue nel sottoalbero sinistro
- Se key(x) < key(n) la scansione prosegue nel sottoalbero di destra
- La scansione termina quando il sottoalbero selezionato è vuoto ovvero ha valore NULL
- Al termine della scansione si inserisce il nuovo nodo

Prima

Dopo

Primitiva PRINT_BST

-Funzione che stampa tutti i nodi di un albero, attraverso una visita DFS in-order

```
void print_BST(bst b){

    if(get_left(b) != NULL)
        print_BST(get_left(b));

    cout<<(get_key(b));
    cout<<" ";
    print(get_value(b));
    cout<<endl;

    if(get_right(b)! = NULL)
        print_BST(get_right(b));
}
```

Primitiva SEARCH

-Funzione che restituisce:
- un puntatore a un nodo con chiave data, se esiste
- il valore NULL altrimenti

```
bnode* bst_search(bst b,tipo_key k){

    while (b != NULL) {

        if (compare_key(k,get_key(b)) == 0)
            return b;

        if (compare_key(k,get_key(b)) < 0){
            b = get_left(b);
        }
        else {
            b = get_right(b);
        }
    }
    return NULL;
 }
```

Come funziona la primitiva SEARCH?

- Scansione dell'albero a partire dalla radice verso il basso il percorso di scansione dipende dall'esito del confronto tra la chiave del nodo corrente z e la chiave da cercare **k: key(z) e k.**
  - Se k = key(z) restituisco z
  - Se k < key(z) la scansione prosegue nel sottoalbero sinistro
  - Se key(z) < k la scansione prosegue nel sottoalbero di destra

Primitiva BST_DELETE

```
void bst_delete(bst& b, bnode* n){

    bnode* new_child;
        if (get_left(n) == NULL) {
            if (get_right(n) == NULL)
                new_child = NULL;
            else {
                cout << "Nodo con solo figlio destro\n";
                new_child = get_right(n);
            }
        }
        else if(get_right(n) == NULL) {
                cout << "Nodo con solo figlio sinistro\n";
                new_child = get_left(n);
            }
            else {
                cout << "Nodo con entrambi i figli\n";
                bnode* app = get_left(n);
                while (get_right(app) != NULL)
                    app = get_right(app);

                if(get_left(app) == NULL){
                    update_father(app, NULL);
                }
                else{
                    (app->parent)->right = get_left(app);
                    (app->left)->parent = get_parent(app);
                }
                app->left = get_left(n);
                app->right = get_right(n);
                if(get_left(app) != NULL)
                    (app->right)->parent = app;
                if(get_left(app) != NULL)
                    (app->left)->parent = app;
                new_child = app;
            }


        if(new_child != NULL)
             new_child->parent = get_parent(n);
        if(n == b)
             b = new_child;
        else
             update_father(n, new_child);
        delete n;
}
```

Come funziona BST_delete?

- Tre casi possibili:
  - Il nodo n **è una foglia** -> Il nodo viene semplicemente cancellato
  - Il nodo n ha un **solo figlio** -> Il nodo n viene cancellato creando un collegamento tra il padre e il figlio di n
  - Il nodo n ha **due figli** ->
    - Sicuramente n' il minore dei successori non ha un figlio sinistro (ha al più un figlio destro) Perché?
      - Perché se n' avesse un figlio sinistro s-n' allora s-n' sarebbe sicuramente minore di n' ma maggiore di n, trovandosi alla destra di n ovvero n < s-n'