LaB 6: Binary Search Tree

```
struct dict::bstNode {
  Elem info;
  bstNode *leftChild;
  bstNode *rightChild;
};
```

createEmptyDictionary

```
Dictionary dict::createEmptyDict(){
  Dictionary d = emptyDictionary;          // emptyDictionary = nullptr
  return d;
}
```

isEmpty

```
bool dict::isEmpty(const Dictionary& d){
  return (d == emptyDictionary);           // emptyDictionary = nullptr
}
```

insertElement

```
Error dict::insertElem(const Key k, const Value v, Dictionary& d){
  if (isEmpty(d)) {                  // caso base (se il nodo è vuoto)
    bstNode *node = new bstNode;     // creo il nodo . . .
    node -> info.key = k;            // . . . con chiave k . . .
    node -> info.value = v;          // . . . e valore v.
    node -> leftChild = emptyNode;   // creo il figlio sinistro del nodo
    node -> rightChild = emptyNode;  // creo il figlio destro del nodo
    d = node;      // d, che rappresenta il nodo che stiamo controllando, diventa node
    return OK;     // la funzione restituisce ok per indicare che l'operazione di inserimento è riuscita
  }
  // se la chiave del nodo corrente è uguale alla chiave k che si vuole inserire, l'inserimento fallisce, per
    evitare duplicati
  if (d -> info.key == k) {
    return FAIL;
  }
  // se la chiave k è maggiore della chiave del nodo corrente (d->info.key), l'inserimento procede in maniera
    ricorsiva nel sottoalbero destro (d->rightchild)
  if (d -> info.key < k) {
    return insertElem(k, v, (d -> rightChild));
  // se k è minore, l'inserimento procede ricorsivamente nel sottoalbero sinistro (d->leftchild)
  } else {
    return insertElem(k, v, (d -> leftChild));
  }
}
```

search

```
Value dict::search(const Key k, const Dictionary& d){
  // Se l'albero è vuoto, non è possibile trovare la chiave k, quindi viene restituito emptyValue
  if (isEmpty(d)) {
    return emptyValue;
  }
  // Se la chiave del nodo corrente è uguale alla chiave k che sto cercando, la
funzione restituisce il valore associato a quella chiave

  if (d -> info.key == k) { // Se la chiave del nodo corrente è uguale alla chiave che sto cercando
    return d -> info.value; // La funzione restituisce il valore associato a quella chiave
  }
  if (d -> info.key < k) {                    // Se è k maggiore del nodo corrente
    return search(k, (d -> rightChild));      // cerca a destra
  } else {                                    // Se k minore del nodo corrente
    return search(k, (d -> leftChild));       // cerca a sinistra
  }
}
```

deleteMin

// funzione ausiliaria per deleteElem

```
Elem deleteMin(Dictionary& d) {
  // Quando il sottoalbero sinistro del nodo corrente (d->leftChild) è vuoto, significa che d è il nodo più a
    sinistra e quindi il nodo minimo
  if ((d -> leftChild) == emptyNode) {
    Elem temp = d -> info;     // Il valore del nodo minimo viene salvato in una variabile temp
    Dictionary tempChild = d -> rightChild;      // figlio dell'ultimo nodo salvato
    delete d;            // elimino d (nodo corrente)
    d = tempChild;       // sottoalbero viene collegato al padre del nodo minimo sostituendo d con
                           d->rightChild.
    return temp;         // ritorno il valore del nodo minimo salvato in temp
  } else {
    return deleteMin(d -> leftChild); // Se il nodo corrente (d) non è il nodo minimo, la funzione
                                         richiama se stessa sul sottoalbero sinistro (d->leftChild)
  }
}
```

deleteElement

```
Error dict::deleteElem(const Key k, Dictionary& d){
  // se l'albero è vuoto, significa che la chiave k non è presente, quindi l'operazione fallisce e restituisce FAIL
  if (isEmpty(d)) {
    return FAIL;
  }
  // se la chiave del nodo corrente è uguale a k, significa che abbiamo trovato il nodo da eliminare
  if (d -> info.key == k) {
    // caso senza figli
    if (d -> rightChild == emptyNode && d -> leftChild == emptyNode) {
      delete d;          // il nodo viene semplicemente eliminato con delete d
      d = emptyNode;     // d viene impostato su emptyNode per rappresentare un nodo vuoto
    // caso solo figlio sinistro
    } else if (d -> rightChild == emptyNode) {
      Dictionary temp = d -> leftChild;    // si salva il puntatore al figlio sinistro in temp
      delete d;                 // il nodo corrente viene eliminato (delete d)
      d = temp;                 // il puntatore d viene aggiornato a temp, collegando il figlio sinistro
                                   al genitore del nodo eliminato
    // caso solo figlio destro
    } else if (d -> leftChild == emptyNode) {
      Dictionary temp = d -> rightChild;   // si salva il puntatore al figlio destro in temp
      delete d;                 // il nodo corrente viene eliminato (delete d)
      d = temp;                 // il puntatore d viene aggiornato a temp, collegando il figlio destro
                                   al genitore del nodo eliminato
    // caso entrambi i figli
    } else {
      // il nodo deve essere sostituito con il suo successore in ordine (il nodo con la chiave più piccola nel
      suo sottoalbero destro)
      d -> info = deleteMin(d -> rightChild);
    }
    return OK;     // ritorno ok perché l'elemento è stato eliminato
  }
  // se il nodo corrente non è quello da eliminare, si continua la ricerca nell'albero
  if (d -> info.key < k) {                        // se k è maggiore della chiave corrente
    return deleteElem(k, (d -> rightChild));      // funzione chiamata sul sottoalbero destro
  } else {                                        // se k è minore della chiave corrente
    return deleteElem(k, (d -> leftChild));       // funzione chiamata sul sottoalbero sinistro
  }
}
```

print

```
void print(const Dictionary& d) {
  if (isEmpty(d)) {
    return;
  }
  if (d -> leftChild != emptyNode) {
    print(d -> leftChild);
  }
  cout << d -> info.key << " : " << d -> info.value << endl;
  if (d -> rightChild != emptyNode) {
    print(d -> rightChild);
  }
}
```