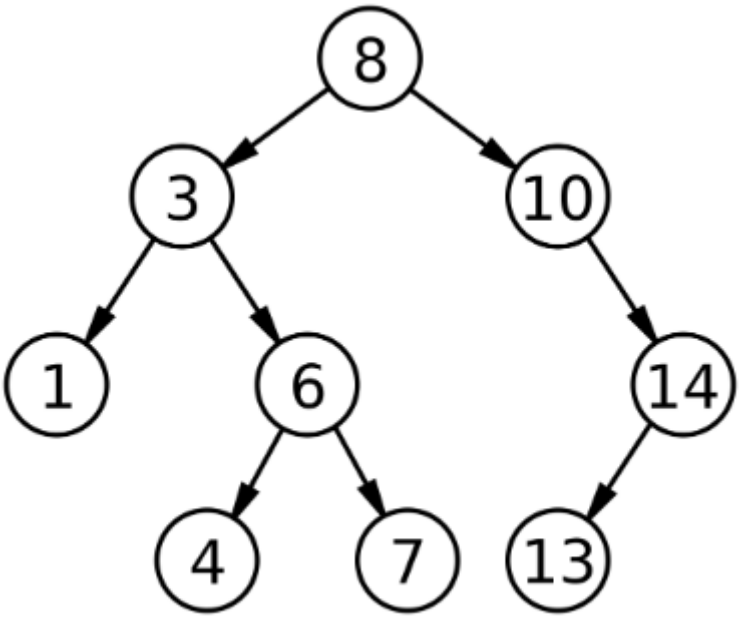
# <이진탐색트리(Binary Search Tree)>

## 1. BST 란?

Binary search tree(이하 BST)는 각 노드에 값이 있으며 그 값이 특정한 순서로 정렬이 가능할 때, 그 순서에 따라 정렬이 되어 있는 Binary tree를 의미한다.
노드의 순서는 대체로 대상 노드의 왼쪽 자식 트리는 대상 노드의 값보다 작은 값들을 가지며, 오른쪽 자식 트리는 대상 노드의 값보다 큰 값들을 가진다.



Binary search tree

BST는 정렬 및 검색을 효율적으로 할 수 있도록 도와준다.

## 2. 기능

- 검색 (Searching)
- 삽입 (Insertion)
- 삭제 (Deletion)
- 순회 (Traversal)
- 검증 (Verification)

## 3. 구현

### 1) 노드 타입 정의

```go
// Node is a single node in the tree
type Node struct {
    key   int
    value Item
    left  *Node
    right *Node
}
```

각 노드는 key와 value를 가지며, 2개의 자식 노드를 가리키는 포인터를 가진다. 자식 노드는 left와 right로 구별했다.

그리고 전체 트리 구조를 파악 할 수 있도록 루트 노드를 가리키는 BST 타입을 정의한다.

```go
// Bst is the binary search tree
type Bst struct {
    root *Node
}
```

### 2) 탐색 연산

원하는 key를 입력했을 때 해당 키를 갖는 노드를 반환하는 함수를 구현한다. 만약 해당 key를 갖는 노드를 찾을 수 없는 경우 nil를 반환하도록 한다.

```go
// Search searches the target node using key from the tree
func (bst *Bst) Search(key int) *Node {
    return search(bst.root, key)
}

// search is the internal recursive function to find the node by key
// it returns nil if no node was found
func search(n *Node, k int) *Node {
    if n == nil || n.key == k {
        return n
    }
    if k < n.key {
        return search(n.left, k)
    }
    // k > n.key
    return search(n.right, k)
}
```

### 3) 삽입 연산

Key와 value를 입력으로 받아 대상 트리에 새로운 노드를 추가하는 함수를 구현한다. 만약 트리가 root 노드에 존재하지 않는 빈 트리라면, 새로운 노드를 root노드로써 추가한다.

```go
// Insert inserts the new node into the tree
func (bst *Bst) Insert(key int, value Item) {
    n := &Node{ key: key,  value: value,  left: nil,  right: nil}
    if bst.root == nil {
        bst.root = n
    } else {
        insert(bst.root, n)
    }
}
```

만약 트리에 이미 root가 존재하면, 새로운 노드를 추가할 올바른 위치를 찾아야 한다.

Key를 비교하며 새로운 노드가 추가될 위치를 찾는 재귀함수를 구현한다.

```
// insert is the internal recursive function to insert new node
// by finding proper position recursively
func insert(n *Node, new *Node) {
    if new.key < n.key {
        if n.left == nil {
            n.left = new
        } else {
            insert(n.left, new)
        }
    } else {
        if n.right == nil {
            n.right = new
        } else {
            insert(n.right, new)
        }
    }
}
```

#### 4) 삭제 연산

노드를 삭제한 뒤에도 전체 BST는 기능을 유지해야 한다. 따라서 어떤 노드가 삭제되는가에 따라 BST형태를 유지하기 위한 추가 처리가 필요 할 수 있다.

1. 자식 노드를 가지지 않는 leaf 노드를 삭제하는 경우
2. 하나의 자식 노드만 가지는 노드를 삭제하는 경우
3. 좌/우 자식을 모두 가지는 노드를 삭제하는 경우

먼저 BST에 대한 함수 Delete (k int)와 실제 삭제 처리를 담당한 내부 재귀 함수 Delete(n * Node, k int)는 다음과 같다.

```go
// delete finds target node recursively and delete it from the tree
func delete(n *Node, k int) *Node {
    // if the current node is not the target node
    if n == nil {
        return nil
    }
    if k < n.key {
        n.left = delete(n.left, k)
        return n
    }
    if k > n.key {
        n.right = delete(n.right, k)
        return n
    }
```

재귀함수 Delete (n *Node, k int)에서 현재 노드가 삭제할 노드일 경우에 대한 구현이다.

- Leaf 노드의 삭제

```go
// if the current node is the target node
if n.left == nil && n.right == nil {
    // current node doesn't have any child node
    n = nil
    return nil
```

- 자식을 하나만 가지는 노드의 삭제

```go
} else if n.left == nil {
    // current node has right child node only
    n = n.right
    return n
} else if n.right == nil {
    // current node has left child node only
    n = n.left
    return n
```

- 좌/우 자식을 모두 가지는 노드의 삭제

BTS특성

- 좌측 자식들을 해당 노드보다 작은 키 값을 가진다.
- 우측 자식들은 해당 노드보다 큰 키 값을 가진다.

삭제된 노드를 대체할 수 있는 자식노드는 다음과 같다.

- 좌측 자식들 중 가장 큰 키값을 가지는 노드
- 우측 자식들 중 가장 작은 키값을 가지는 노드

트리 내의 최대값과 최소값을 찾는 함수를 구현해본다.

```go
// max finds most right child
func max(n *Node) *Node {
    max := n
    for {
        if max != nil && max.right != nil {
            max = max.right
        } else {
            break
        }
    }
    return max
}

// min finds most left child
func min(n *Node) *Node {
    min := n
    for {
        if min != nil && min.left != nil {
            min = min.left
        } else {
            break
        }
    }
    return min
}
```

우측 자식들 중 가장 작은 키 값을 가지는 노드로 삭제된 노드를 대체하도록 구현했다. 현재 노드를 대체한 후에 우측 자식 트리 내에서 대체된 노드를 삭제해야 한다.

```
    } else {
        // current node has both left and right child
        n = max(n.right)
        delete(n.right, n.key)
        return n
    }
}
```

### 5) 순회

순회는 트리 구조 내의 노드를 순서대로 방문하며 각 아이템을 반환하는 로직이다.

재귀적인 순회 로직을 이용해 왼쪽 서브 트리부터 방문하여, 더 이상의 서브트리가 존재하지 않는 상황에서 오른쪽 서브트리에 대해 순회를 반복하는 것으로 순서대로 방문하는 것이 가능하다.

```
// Traverse visits all nodes in order
func (bst *Bst) Traverse(f func(Item)) {
    traverse(bst.root, f)
}

// traverse is the internal recursive function to visit the nodes
func traverse(n *Node, f func(Item)) {
    if n != nil {
        traverse(n.left, f)
        f(n.value)
        traverse(n.right, f)
    }
}
```

### 6) 검증

모든 노드를 대상으로 오른쪽 서브 트리는 해당 노드보다 큰 노드만이 존재하며, 왼쪽 서브 트리는 해당 노드보다 작은 노드만이 존재한다.

```go
func isBST(n *Node, min, max int) bool {
    if n == nil {
        return true
    }
    if n.key < min || n.key > max {
        return false
    }

    return isBST(n.left, min, n.key-1) && isBST(n.right, n.key+1, max)
}
```

**<Reference>**


1. https://kinchi22.github.io/2019/02/10/bst-using-go/