Islamic University of Technology

CSE 4810

Algorithm Engineering Lab

Lab 2

Tasnimul Hasnat

190041113

CSE 1A

March 17, 2024

Task 1

Solve sum of all elements smaller or equal to the Kth element

```go
type Node struct {
  val, height, size int
  left, right       *Node
}

func (n *Node) Height() int {
  if n == nil {
    return 0
  }
  return n.height
}

func (n *Node) Size() int {
  if n == nil {
    return 0
  }
  return n.size
}

func (n *Node) Update() {
  n.height = max(n.left.Height(), n.right.Height()) + 1
  n.size = n.left.Size() + n.right.Size() + 1
}

func (n *Node) balanceFactor() int {
  if n == nil {
    return 0
  }
  return n.left.Height() - n.right.Height()
}

func (x *Node) rotateLeft() *Node {
  y := x.right
  x.right = y.left
  y.left = x
  x.Update()
  y.Update()
  return y
}

func (y *Node) rotateRight() *Node {
  x := y.left
  y.left = x.right
  x.right = y
  y.Update()
  x.Update()
  return x
}
```

```
func (n *Node) insert(val int) *Node {
  if n == nil {
    return &Node{val: val, height: 1, size: 0}
  }
  if val < n.val {
    n.left = n.left.insert(val)
  } else if val > n.val {
    n.right = n.right.insert(val)
  } else {
    return n // no duplicate allowed
  }
  n.Update()

  bf := n.balanceFactor()

  // LL case
  if bf > 1 && val < n.left.val {
    return n.rotateRight()
  }
  // RR case
  if bf < -1 && val > n.right.val {
    return n.rotateLeft()
  }
  // LR case
  if bf > 1 && val > n.left.val {
    n.left = n.left.rotateLeft()
    return n.rotateRight()
  }
  // RL case
  if bf < -1 && val < n.right.val {
    n.right = n.right.rotateRight()
    return n.rotateLeft()
  }
  return n
}

func (n *Node) sumSmallerOrEqualKthElem(k int) int {
  if n == nil {
    return 0
  }
  if k == n.Size() {
    return n.val + n.left.Size()
  } else if k < n.Size() {
    return n.left.sumSmallerOrEqualKthElem(k)
  } else {
    return n.val + n.left.Size() + n.right.sumSmallerOrEqualKthElem(k-1-n.left.Size())
  }
}

func main() {
  var root *Node
  tree := []int{7, 2, 13, 9, 10, 8}
  // build
```

```
  for _, i := range tree {
    root = root.insert(i)
  }
  var k int
  fmt.Print("Enter K = ?\b")
  fmt.Scanf("%d", &k)
  fmt.Print(root.sumSmallerOrEqualKthElem(k))
}
```

The solution creates an AVL Tree and on that searches for the sum. In order to find the sum an extra value is kept, `size`, which denotes the subtree size of each element. `sumSmallerOrEqualKthElem` function checks if the k is equal to the root subtree size then returns their own value with the left subtree sum. If its less then just returns the left subtree sum. Thus it computes the total sum.

Time Complexity: $O(\log n)$

Task 2

Given an array arr[] of size N where each element denotes a pair in the form (price, weight) denoting the price and weight of each item. Given Q queries of the form [X, Y] denoting the price range. The task is to find the element with the highest weight within a given price range for each query.

```
type item struct {
  price  int
  weight int
}

type segmentTree struct {
  tree []int
  size int
}

func buildSegmentTree(arr []item, start, end int, st *segmentTree) {
  if start == end {
    st.tree[st.size*start+1] = arr[start].weight
    return
  }
  mid := (start + end) / 2
  buildSegmentTree(arr, start, mid, st)
  buildSegmentTree(arr, mid+1, end, st)
        st.tree[st.size*start+1]     =     max(st.tree[st.size*2*start+2],
st.tree[st.size*2*start+3])
}

func max(a, b int) int {
  if a > b {
    return a
  }
  return b
}

func query(st *segmentTree, qs, qe int, start, end int) int {
  if qs <= start && qe >= end {
    return st.tree[st.size*start+1]
  }

  if qs > end || qe < start {
    return 0
  }

  mid := (start + end) / 2
  return max(query(st, qs, qe, start, mid), query(st, qs, qe, mid+1, end))
}

func processQueries(arr []item, queries [][]int) []int {
  n := len(arr)
  st := &segmentTree{
    tree: make([]int, 4*n), // 4*n to accommodate worst-case tree size
    size: n,
```

```go
  }
  buildSegmentTree(arr, 0, n-1, st)

  results := make([]int, len(queries))
  for i, _query := range queries {
    low, high := _query[0], _query[1]
    results[i] = query(st, low, high, 0, n-1)
  }
  return results
}

func main() {
  arr := []item{
    {price: 24, weight: 6},
    {price: 30, weight: 8},
    {price: 21, weight: 7},
  }
  queries := [][]int{
    {10, 24},
    {20, 30},
  }
  results := processQueries(arr, queries)
  fmt.Println(results)
}
```

Using Segment Tree we can efficiently query the highest weight for any given range.

Task 3

Given a binary 2D matrix, find the number of islands. A group of connected 1s forms an island.

```go
package main

import "fmt"

type Grid struct {
  data [][]byte
  rows int
  cols int
}

func NewGrid(grid [][]byte) *Grid {
  return &Grid{
    data: grid,
    rows: len(grid),
    cols: len(grid[0]),
  }
}

func (g *Grid) NumOfIslands() int {
  if g.rows == 0 {
    return 0
  }

  var count int

  for i := 0; i < g.rows; i++ {
    for j := 0; j < g.cols; j++ {
      if g.data[i][j] == 1 {
        g.dfs(i, j)
        count++
      }
    }
  }

  return count
}

func (g *Grid) dfs(i, j int) {
  if i < 0 || j < 0 || i >= g.rows || j >= g.cols || g.data[i][j] != 1 {
    return
  }
  g.data[i][j] = 0

  g.dfs(i+1, j)
  g.dfs(i-1, j)
  g.dfs(i, j+1)
  g.dfs(i, j-1)
}
```

```go
func main() {
  mat := [][]byte{
    {1, 1, 0, 0, 0},
    {0, 1, 0, 0, 1},
    {1, 0, 0, 1, 1},
    {0, 0, 0, 0, 0},
    {1, 0, 1, 0, 0},
  }
  g := NewGrid(mat)
  fmt.Println(g.NumOfIslands())
}
```

Using a simple `DFS` we can compute the number of connected components aka *"islands"* in the given matrix.

Point to note, the output given in the slide is wrong unless the matrix is a torus only in the x axis i.e the matrix wraps around on the x axis.

Correct Output: 5 (instead of 4)

Task 4

The right view of a Binary Tree is a set of nodes visible when the tree is visited from the Right side. Given a Binary Tree, print the Right view of it.

```go
func rightViewRec(tree []byte, index, curLevel byte, maxLevel *byte) {
  if index >= byte(len(tree)) {
    return
  }
  if *maxLevel < curLevel {
    fmt.Print(tree[index], " ")
    *maxLevel = curLevel
  }
  rightViewRec(tree, 2*index+2, curLevel+1, maxLevel) // right
  rightViewRec(tree, 2*index+1, curLevel+1, maxLevel) //left
}
func rightView(tree []byte) {
  maxLevel := byte(0)
  rightViewRec(tree, 0, 1, &maxLevel)
}
func main() {
  var n byte
  fmt.Scanln(&n)
  tree := make([]byte, n)
  for i := byte(0); i < n; i++ {
    fmt.Scanf("%v", &tree[i])
  }
  rightView(tree)
}
```

We can view a tree from the right side using a recursive approach where we do a pre-order traversal but instead of traversing the left subtree first we will traverse the right subtree first. Using an **if** check if we have already visited that level through the right subtree, then ignore printing that node ensuring any unwanted values.