CENG311 – Programing Assignment 2 Report

Umut YILDIZ / 260201028

- Build
- Insert
- Print

And

- Get Depth
- Insert To Level
- Traverse Level
- Heapfiy
- Last words

Build(list)

- Start of build process. Argument and the return address register value is stored to stack.

```
#####################################
# Build Procedure
#####################################
build:
    # first creates a root node from the first entity of unorderedList
    # until the first value of the unorderedList is -1, it loads the value and calls insert
    # when the first value is -1 then it returns the root node.
    # arguments $a0: unorderedList
    # returns   $v0: root
    move $s0, $a0   #s0: unorderedList

    subu $sp, $sp, 8
    sw $ra, 0($sp)
    sw $a0, 4($sp)
```

- Root is created and the address is stored to s1.

```
 li $v0, 9
 li $a0, 16
 syscall
 sw $a0, 4($sp)
 move $s1, $v0   #s1: root

 lw $t0, 0($s0)
 sw $t0, 0($s1)
 sw $zero, 4($s1)
 sw $zero, 8($s1)
 sw $zero, 12($s1)
```

- Build loop, insertion for every value except –1 and the exit of build

```
buildLoop:
    addi $s0, $s0, 4
    lw $t0, 0($s0)              # t0: next value
    li $t1, -1                  # t1 =- 1
    beq $t0, $t1, buildLoopEnd  # if (t0 !== t1) {
                                #
    move $a0, $s1               #
    move $a1, $t0               #
    jal insert                  # insert(root, number)
    j buildLoop                 # }
buildLoopEnd:
    lw $ra, 0($sp)
    lw $a0, 4($sp)
    addu $sp, $sp, 8

    move $v0, $s1               # return root
    jr $ra
```

Insert(root, number)

- Start of insert process. Calling get depth

```
insert:
    # creates the newNode to insert then finds the floor where this newNode should be inserted.
    # calls insertToLevel to insert the newNode
    # after insertion calls heapify to make the node tree a max binary heap tree
    # arguments     # $a0: root, $a1: entry
    subu $sp, $sp, 16
    sw $a0, 0($sp)
    sw $a1, 4($sp)
    sw $s0, 8($sp)
    sw $ra, 12($sp)

    jal getDepth        # getDepth(root)
    move $t1, $v0       # t1: level
```

- Creating new node

```
li $v0, 9
li $a0, 16
syscall             # s0 = malloc(16)
move $s0, $v0       # s0: new node
lw $a0, 0($sp)      # fix a0 from

sw $a1, 0($s0)      # node.data = a1
sw $zero, 4($s0)    # node.left = 0
sw $zero, 8($s0)    # node.right = 0
sw $zero, 12($s0)   # node.parent = 0
```

- Calling insert to level and heapify, then exiting from insert.

```
lw $a0, 0($sp)      # root
move $a1, $s0       # newNode
move $a2, $t1       # level
jal insertToLevel   # insertToLevel(root, newNode, level)
# insertToLevel returns the newNode

move $a0, $v0
jal heapify         # heapify(newNode)

lw $a0, 0($sp)
lw $a1, 4($sp)
lw $s0, 8($sp)
lw $ra, 12($sp)
addu $sp, $sp, 16
jr $ra
```

Print(root)

- Start of print process. Calling get depth. Setting `i` and `max level` values for print loop.

```
#####################################
# Print Procedure
#####################################
print:
    # for each floor in tree node calls travesCurrentLevel function to print all tre
    # arguments  $a0: root
    subu $sp, $sp, 16
    sw $ra, 0($sp)
    sw $a0, 4($sp)
    sw $s7, 8($sp)
    sw $s6, 12($sp)

    jal getDepth    # getDepth(root)

    move $s7, $v1   #s7: level max
    li $s6, 0       #s6: i
```

- Print loop and calling traverse level for each floor of tree.

```
printLoop1:
    addi $t0, $s6, 1                # t0 = i + 1

    li $v0, 1
    move $a0, $t0
    syscall                         # print(t0)

    li $v0, 4
    la $a0, level
    syscall                         # print(" Level:\t")

    lw $a0, 4($sp)
    move $a1, $s6
    jal traverseLevel        # traverseLevel(node, level)

    li $v0, 4
    la $a0, newLine
    syscall                         # print(\n)

    beq $s7, $s6, printLoop1End     # if (i === levelMax)
    addi $s6, $s6, 1                # i++
    j printLoop1                    #

printLoop1End:
    lw $ra, 0($sp)
    lw $a0, 4($sp)
    lw $s7, 8($sp)
    lw $s6, 12($sp)
    addu $sp, $sp, 16

    jr $ra
```

GetDepth(node)

- Start of get depth, setting variables for depth finding loops.

```
getDepth:
    # returns the depth of the node tree
    # @arguments    # $a0: node
    # @returns      # $v0: min number $v1: max number
    # v0 is used to insert  value, v1 is used when printing
    subu $sp, $sp, 8
    sw $s7, 0($sp)
    sw $s6, 4($sp)

    li $s7, 0   # left
    li $s6, 0   # right
    move $t7, $a0   # left node
    move $t6, $a0   # right node
```

- Left and right loops to find depth of left and right.

```
getDepthLoop1:                          # while (nodeLeft.left) {
    lw $t0, 4($t7)                      # t0 = nodeLeft.left
    beq $t0, $zero, getDepthLoop1End    # t0 !== 0;
    lw $t7, 4($t7)                      # nodeLeft = nodeLeft.left
    addi $s7, $s7, 1                    # left++
    j getDepthLoop1                     # }
getDepthLoop1End:

getDepthLoop2:                          # while (nodeRight.right) {
    lw $t0, 8($t6)                      # t0 = nodeRight.right
    beq $t0, $zero, getDepthLoop2End    # t0 !== 0;
    lw $t6, 8($t6)                      # nodeRight = nodeRight.right
    addi $s6, $s6, 1                    # right ++
    j getDepthLoop2                     # }
getDepthLoop2End:
```

- Comparing left and right values and returning the depth.

```
slt $t0, $s7, $s6
beq $t0, $zero, getDepthLeftMax
j getDepthRightMax

getDepthLeftMax:                        # if (right > left) {
    move $v0, $s6                       # v0 = left
    move $v1, $s7                       # v1 = right
    j getDepthExit                      # {

getDepthRightMax:                       # if (right > right) {
    move $v0, $s7                       # v0 = right
    move $v1, $s6                       # v1 = left
    j getDepthExit                      # }

getDepthExit:
    lw $s7, 0($sp)
    lw $s6, 4($sp)
    addu $sp, $sp, 8
    jr $ra
```

InsertToLevel(root, newNode, level)

- Start of insert to level

```
insertToLevel:
    # recursively call himself to reach lower levels until level is 0.
    # when level is 0, tries to insert the newNode to a empty spot at that level.
    # if there is no empty spot then it returns v0 = 0 which means
    # the insertion is failed for that sub-recursion process.
    # after insertion returns the newNode

    # arguments     # $a0: root, $a1: newNode, $a2: level
    subu $sp, $sp, 12
    sw $ra, 0($sp)
    sw $a0, 4($sp)
    sw $a2, 8($sp)
```

- Trying to insert when level is 0

```
bne $a2, $zero, insertToLevelRecusive           # if (level === 0) {
                                                #
lw $t0, 4($a0)          #t0: a0.left            #
bne $t0, $zero, insertToLevelDoesntCheckLeft    #   if (!node.left) {
sw $a0, 12($a1)         #a1.parent: a0          #
sw $a1, 4($a0)          #a0.left: a1            #
move $v0, $a1                                   #
j insertToLevelEnd                              #   }
insertToLevelDoesntCheckLeft:                   #
                                                #
lw $t0, 8($a0)          #t0: a0.right           #
bne $t0, $zero, insertToLevelDoesntCheckRight   #   if (!node.right) {
sw $a0, 12($a1)         #a1.parent: a0          #
sw $a1, 8($a0)          #a0.right: a1           #
move $v0, $a1                                   #
j insertToLevelEnd                              #   }
insertToLevelDoesntCheckRight:                  #
                                                #
li $v0, 0                                       #
j insertToLevelEnd                              # }
insertToLevelRecusive:
```

- InsertToLevel recursion calls and the exit of insertToLevel

```
subu $a2, $a2, 1                                # level -= 1
lw $a0, 4($a0)                                  # a0 = a0.left
jal insertToLevel                               # const left = insertNodeToLevel(node.left, newNode, level - 1);
beq $v0, $zero, insertToLevelRecusiveNotLeft    # if (left) {
j insertToLevelEnd                              # }
insertToLevelRecusiveNotLeft:

lw $a0, 4($sp)  # refresh the root

lw $a0, 8($a0)                                  # a0 = a0.right
jal insertToLevel                               # const right = insertNodeToLevel(node.right, newNode, level - 1);
beq $v0, $zero, insertToLevelRecusiveNotRight   # if (right) {
j insertToLevelEnd                              # }
insertToLevelRecusiveNotRight:

insertToLevelEnd:
    lw $ra, 0($sp)
    lw $a0, 4($sp)
    lw $a2, 8($sp)
    addu $sp, $sp, 12
    jr $ra
```

TraverseLevel(node, level)

- Start of insertToLevel and checking for level and node value

```
traverseLevel:
    # recursively call himself to reach lower levels until level is 0.
    # when level is 0, prints the values on that level seperated by " ".
    # arguments $a0: node | $a1: level
    subu $sp, $sp, 12
    sw $ra, 0($sp)
    sw $a0, 4($sp)
    sw $a1, 8($sp)

    beq $a0, $zero, traverseLevelDone        # if (node === 0)
    beq $a1, $zero, traverseLevelPrintDone   # if (level === 0)
```

- Recursive calls traverseLevel to reach level 0

```
lw $a0, 4($a0)                      # a0 = a0.left
subu $a1, $a1, 1                    # a1 = t7
jal traverseLevel                   # traverseLevel(node.left, level - 1)

lw $a0, 4($sp)  # refresh a0
lw $a1, 8($sp)  # refresh a1

lw $a0, 8($a0)                      # a0 = a0.right
subu $a1, $a1, 1                    # a1 = t7
jal traverseLevel                   # traverseLevel(node.right, level - 1)

j traverseLevelDone
```

- Printing and exit of traverseLevel process

```
traverseLevelPrintDone:             # if (level === 0) {
    lw $t1, 0($a0)                  # t1 = node.data
    li $v0, 1                       #
    move $a0, $t1                   #
    syscall                         # print(t1)
                                    #
    li $v0, 4                       #
    la $a0, space                   #
    syscall                         # print(" ")
                                    # }
traverseLevelDone:
    lw $ra, 0($sp)
    lw $a0, 4($sp)
    lw $a1, 8($sp)
    addu $sp, $sp, 12
    jr $ra
```

Heapify(newNode)

- Start of heapify.

```
heapify:
    # until the newNode does not have a parent it continues to check if node.data is bigger than node.parent.data
    # if bigger than it swaps them, continues with newNode=newNode.parent
    # arguments  a0: newNode
    subu $sp, $sp, 4
    sw $s7, 0($sp)

    move $s7, $a0   #s7 = newNode
```

- Heapify check loop and the exit of heapify process

```
heapifyLoop1:
    lw $t0, 12($s7) # t0 = parent | s7 = node
    beq $t0, $zero, heapifyLoop1End             # node.parent !== 0
    lw $t1, 0($s7)  # node.data                 #
    lw $t2, 0($t0)  # parent.data               #
                                                #
    slt $t3, $t2, $t1                           # node.data > node.parent.data
    beq $t3, $zero, heapifyWithoutSwap          # if (node.data > node.parent.data) {
    sw $t2, 0($s7)                              #   node.data = parent.data
    sw $t1, 0($t0)                              #   parent.data = node.data
    heapifyWithoutSwap:                         # }

    lw $s7, 12($s7)
    j heapifyLoop1
heapifyLoop1End:

lw $s7, 0($sp)
addu $sp, $sp, 4
jr $ra
```

Last words

- This assembly program is inspired by my own code in "ceng311-p2.js" which is easier to understand.
- An example console output screenshot

```
Console
1 Level:  90
2 Level:  87 74
3 Level:  44 67 37 41
4 Level:  8 16 13 29 26 14 23 39

1 Level:  90
2 Level:  87 74
3 Level:  85 67 37 41
4 Level:  46 25 33 52 26 14 23 39
5 Level:  8 44 16 24 3 13 29 45
```

- Insert value argument order is different than what is expected.