Project 4 – Pseudocode

**Tree**

**insertNode(String presData)**

create *newNode* of type Node

assign *presData* to *newNode*

if *root* is null

make *root* point to *newNode*

// end if

else // if node in root

make Node *current* point to *root*

declare Node *parent*

loop while true

make *parent* point to *current*

if *presData* less than *current*’s data

make *current* point to *current*’s left child

if *current* is null

set *parent*’s left child to *newNode*

return

// end if

// end if

else // greater than current node

make *current* point to *current*’s right child

if *current* is null

set *parent*’s right child to *newNode*

return

// end if

// end else

// end loop

// end else

// end insertNode()

**deleteNode(String key)**

make Node *current* point to *root*

make Node *parent* point to *root*

declare boolean *isLeftChild* and set it to true

loop while *current*’s data does not contain *key*

make *parent* point to *current*

if *key* is less than *current*’s data

set *isLeftChild* to true

make *current* point to *current*’s left child

// end if

else // greater than current node; go right

set *isLeftChild* to false

make *current* point to *current*’s right child

// end else

if *current* is null

return false

// end if

// end loop

if *current*’s left child and right child are both null

if *current* is *root*

make *root* point to null

// end if

else if *isLeftChild* is true

set *parent*’s left child to null

// else if

else // *isLeftChild* is false

set *parent*’s right child to null

// end else

// end if

else if *current*’s left child is null

if *current* is *root*

make *root* point to *current*’s right child

// end if

else if *isLeftChild* is true

set *parent*’s left child to *current*’s right child

// end else if

else // *isLeftChild* is false

set *parent*’s right child to *current*’s right child

// end else

// end else if

else if *current*’s right child is null

if *current* is *root*

make *root* point to *current*’s left child

// end if

else if *isLeftChild* is true

set *parent*’s left child to *current*’s left child

// end else if

else // isLeftChild is false

set *parent*’s right child to *current*’s left child

// end else

// end else if

else // node has two children

declare Node *successor* and set to *current*’s successor

if *current* is *root*

make *root* point to *successor*

// end if

else if *isLeftChild* is true

set *parent*’s left child to *successor*

// end else if

else // isLeftChild is false

set *parent*’s right child to *successor*

// end else

// end else

return true

// end deleteNode()

**getSuccessor(Node deleteNode)**

declare Node *successorParent* and set it to *deleteNode*

declare Node *successor* and set it to *deleteNode*

declare Node *current* and set it to *deleteNode*’s right child

loop while current is not null

set *successorParent* to *successor*

set *successor* to *current*

set *current* to *current*’s left child

// end loop

if *successor* is not *deleteNode*’s right child

set *successorParent*’s left child to *successor*’s right child

set *successor*’s right child to *deleteNode*’s right child

// end if

return successor

// end getSuccessor()

**iterativePreOrder(Node localRoot)**

if *localRoot* is null

return

// end if

create a NodeStack object

push *root* onto the *nodeStack*

loop while *nodeStack* is not empty

pop *nodeStack* and set to Node *n*

display *n*

if *n*’s right child is not null

push *n*’s right child onto *nodeStack*

// end if

if *n*’s left child is not null

push *n*’s left child onto *nodeStack*

// end if

// end loop

// end iterativePreOrder()

**iterativeInOrder(Node localRoot)**

if *localRoot* is null

return

// end if

create a NodeStack object

declare Node *current* and set to *root*

loop while *nodeStack* is not empty or *current* is not null

if *current* is not null

push *current* onto *nodeStack*

set *current* to *current*’s left child

// end if

else

pop *nodeStack* and set to Node *n*

display *n*

set *current* to *n*’s right child

// end else

// end loop

// end iterativeInOrder()

**iterativePostOrder(Node localRoot)**

if *localRoot* is null

return

// end if

create a NodeStack object

declare Node *current* and set to *root*

loop while true

if *current* is not null

if *current*’s right child is not null

push *current*’s right child onto *nodeStack*

// end if

push *current* onto *nodeStack*

set *current* to *current*’s left child

continue

// end if

if *nodeStack* is empty

return

// end if

pop *nodeStack* and set to *current*

if *current*’s right child is null and *nodeStack* is not empty and *current*’s right child is at top of stack

pop *nodeStack*

push *current* onto *nodeStack*

set *current* to *current*’s right child

// end if

else

display *current*

set *current* to null

// end else

// end loop

// end iterativePostOrder()

**recursivePreOrder(Node localRoot)**

if *localRoot* is not null

display *localRoot*

call recursivePreOrder() and pass *localRoot*’s left child as an argument

call recursivePreOrder() and pass *localRoot*’s right child as an argument

// end if

// end recursivePreOrder()

**recursiveInOrder(Node localRoot)**

if *localRoot* is not null

call recursiveInOrder() and pass *localRoot*’s left child as an argument

display *localRoot*

call recursiveInOrder() and pass *localRoot*’s right child as an argument

// end if

// end recursiveInOrder()

**recursivePostOrder(Node localRoot)**

if *localRoot* is not null

call recursivePostOrder() and pass *localRoot*’s left child as an argument

call recursivePostOrder() and pass *localRoot*’s right child as an argument

display *localRoot*

// end if

// end recursivePostOrder()

**displayIteratively()**

print header

call displayHeader()

call iterativePreOrder() and pass *root* as an argument

print header

call displayHeader()

call iterativeInOrder() and pass *root* as an argument

print header

call displayHeader()

call iterativePostOrder() and pass *root* as an argument

// end displayIteratively()

**displayRecursively()**

print header

call displayHeader()

call recursivePreOrder() and pass *root* as an argument

print header

call displayHeader()

call recursiveInOrder() and pass *root* as an argument

print header

call displayHeader()

call recursivePostOrder() and pass *root* as an argument

// end displayRecursively()

**displayAfterDelete()**

call displayHeader()

call recursiveInOrder and pass *root* as an argument

// end displayAfterDelete()

**displayHeader()**

print formatted header with column labels for “LAST NAME”, “PARTY”, and “YEARS IN OFFICE”

// end displayHeader()