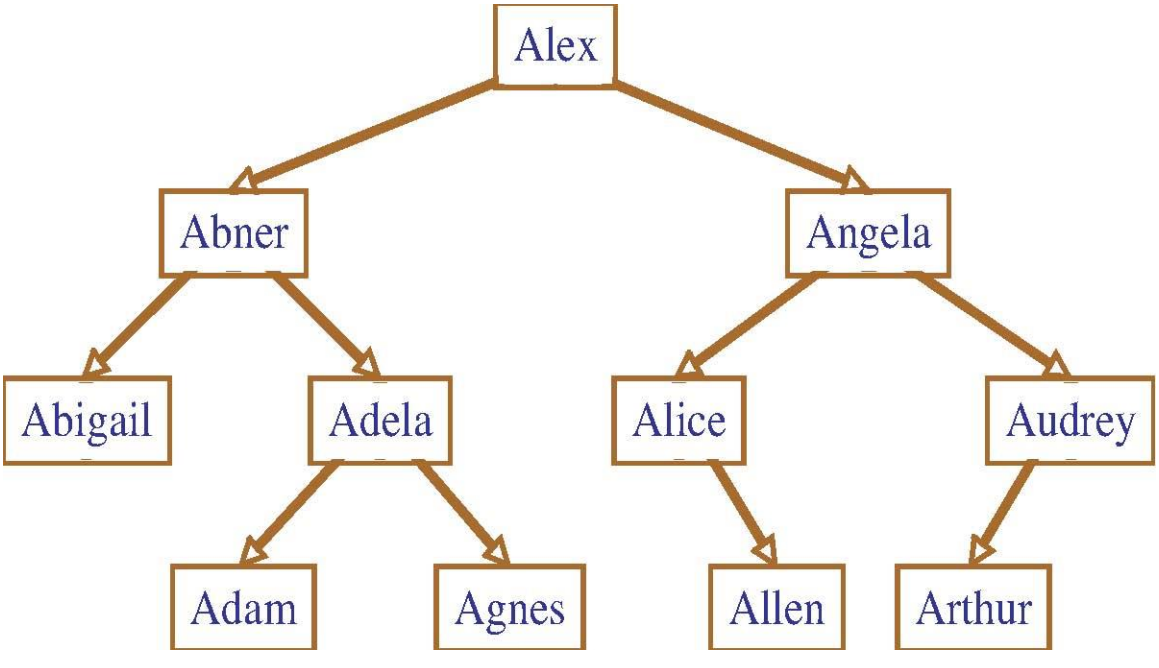
Worksheet 30: Binary Search Tree Iterator

Group 11

In preparation: If you have not done so already, complete worksheet 29 to learn the basic features of the binary search tree.

To make the BST a useful container we need an iterator. This iterator must produce the elements in sequence. As we noted previously, this means making an in-order traversal of the tree. Examine



the following tree and see if you can guess an algorithm that will do this. Remember that, in our implementation, nodes do not point back up to their parent nodes. How will you remember the path you have followed down the tree?

As you might have guessed, one way to do this is to have the iterator maintain an internal stack. This stack will represent the current path that has been traversed . Notice that the first element that our iterator should produce is the leftmost child of the root, and that furthermore all the nodes between the root and this point should be added to the stack. A useful routine for this purpose is slideLeft:

```
void slideLeft (Node n)
  while n is not null
    stack n
    n = left child of n
```

Using the slideLeft routine, verify that the following algorithm will produce the desired iterator traversal of the binary search tree. The remove portion of the iterator interface is more complex, and we will not consider it here.

BSTIteratorInit
to initialize, create an empty stack
int BSTIteratorHasNext
if stack is empty
perform slideLeft on root
otherwise
let n be top of stack. Pop topmost element.
slideLeft on right child of n
return true if stack is not empty

double BSTIteratorNext
let n be top of stack. Return value of n

Can you characterize what elements are being held on the stack? How large can the stack grow?

Stack will hold pointers to the nodes in the tree. Because we are using array for stack implementation, a stack can grow and shrink as much as needed.

Assuming that we have already defined a Stack that will hold pointers to nodes, show the implementation of the functions that implement the iterator algorithms described above.

```
struct BSTIterator {
    struct DynArr  *stk;
    struct BSTree  *tree;
};
void BSTIteratorInit (struct BSTree *tree, struct BSTIterator *itr)
{

itr->tree = tree;

itr->stk = newDynArr(); /* Dyn arr imp of a stack */


}


int BSTIteratorHasNext (struct BSTIterator * itr) {



BNode *n;

if(isEmptyDynArr(itr->stk))

{

slideLeft(itr->tree->root);
```

```
}else {

n = topDynArr(itr->stk);

popDynArr(itr->stk);

slideLeft(n->right);

}

if(!isEmptyDynArr(itr->stk))

return 1;

else return 0;

}

TYPE BSTIteratorNext (struct BSTIterator *itr) {

return (topDynArr(itr->stk))->val;

}

void _slideLeft(struct Node *cur, struct BSTIterator *itr)
 {

While(cur != null)

{

  dynArrayPush(Itr->stk, cur->val);

  cur = cur->left;

 }
 }
```

reference:
http://classes.engr.oregonstate.edu/eecs/fall2013/cs261-001/Week_10/Worksheet30.ans.pdf