Preparation P0

P0

Programming refreshment, practice with standards and header files, practice with trees, traversals, and file IO.
Submission:

```
/accounts/classes/janikowc/submitProject/submit_cs4280_P0
SubmitFileOrDirectory
```

Write a program to build a tree and print it using different traversals. The program will be invoked as

- `P0 [`*`file`*`]`
  where *`file`* is an optional argument. If the *`file`* argument is not given the program will read data from the keyboard as a file device (10%). If the argument is given, the program reads data file *`file`*`.fs17`. (note that *`file`* is any name and the extension is implicit, 70%). Programs improperly implementing file name or executable will not be graded. The remaining 20% is for style. Example invocations:
  - `P0` // read from the keyboard until simulated keyboard EOF
  - `P0 <` *`somefile`* // same as above except redirecting from *`somefile`* instead of keyboard, testing keyboard input
  - `P0` *`somefile`* // read from *`somefile`*`.fs17`

- Assume you do not know the size of the input file
- Assume the input data is all integers (signed 32-bit) separated with standard WS separators
- If the input file is not readable for whatever reason, or command line arguments are not correct, the program will abort with an appropriate message
- The program will read the data left to right and put them into a tree, which is a binary search tree (BST) with respect to the least significant digit in each node, but it is extended as EBST
  - A node contains
    - a digit
    - left EBST subtree
    - right EBST subtree
    - a list of the strings already seen that end with the same digit as the node's digit
  - Tree is never balanced nor restructured other than growing new nodes
- The program will subsequently output 3 files corresponding to the 3 traversals, named *file*.preorder, *f*ile.inorder and *file*.postorder. Note that *file* is the name of the input file if given, and it is out if the input is from the keyboard.
    - Traversals
      - preorder
      - inorder
      - postoder
    - Printing in traversal
      - EBST node will print the node's digit intended by 2 x depth of the node followed by the list of strings from the node so that the newer numbers show up first
- Example (x is null pointer)
  File `xxx.fs17` contains
  `14 104 101 25 375 14 12 222`
  - invocation: > `P0 xxx`

        - Output files: xxx.inorder xxx.preorder xxx.postorder
    - Invocation: > `P0 < xxx.fs17`
        - Output files: out.inorder out.preorder out.postorder

- Standards related requirements:
    - Have the following functions minimum in addition to the main function (the types and arguments are just suggested, the names are required)
      ```
      node_t *buildTree(FILE *);
      void traverseInorder(node_t*, const char[]); // parameters: tree root, and output basefilename
      void traversePreorder(node_t*, const char[]);
      void traversePostorder(node_t*, const char[]);
      ```
    - Put the above four functions into 2 files (1 and 3) with proper headers (`buildTree.h` and `traversals.h`)
    - Define the node type in `node.h`
    - Use this functional architecture even if you program in a language different from C

Traversals

Taken from the 3130 textbook:

Preorder:

- process root
- process children left to right

Inorder:

- process left child
- process root
- proccess right child

Postorder:

- process children left to right
- process root

Suggestions and some Architectureal Requirements

Using top-down decomposition you have 3 tasks in main:

1. Process command arguments, set up file to work regardless of source, check if file readable, set the basename for the output file, etc.
2. Build the tree
3. Traverse the tree three different ways generating outputs

The main function should handle the 3 functionalities. #1 should be handled inside of main, functions for #2 should be in a separate source, and functions for #3 should be in another separate source. Any node types should be defined in a separate header file.

For development purposes, do either 1 or 2 first. 3 should follow 2, first with one traversal only.

Processing either keyboard or file input can be done in either way:

1. If keyboard input, read the input into temporary file, after which the rest of the program always processes file input

2. If keyboard input, set file pointer to stdin. es;e set file pointer to the actual file, then process always from the file pointer

Files:

- `node.h, main.c, traversals.c+treversals.h, buildTree.c+buildTree.h`
- `main.c`

  -
    ```
    #include "node.h"
    #include "traversals.h"
    #include "buildTree.h"

    int main(int argc, char* argv[]) {
      // process command line arguments and make sure
    file is readable, error otherwise
      // set up keyboard processing so that below the
    input is not relevant
      node_t *root = buildTree(file);
      preorder(root);
      inorder(root);
      postorder(root);
      return 0;
    }
    ```

Print tree with identation

```
static void printParseTree(nodeType *rootP,int level) {
  if (rootP==NULL) return;
  printf("%*c%d:%-9s ",level*2,' ',level,NodeId.info); // assume some info
printed as string
  printf("\n");
  printParseTree(rootP->child1,level+1);
  printParseTree(rootP->child2,level+1);
}
```

Testing

This section is non-exhaustive testing of P1

1. Create test files:
   1. `P0_test1.fs17` containing empty file
   2. `P0_test2.fs17` containing one string:
      adam
   3. `P0_test3.fs17` containing one string repeated five times across same line and multiple lines:
      ```
      111 11
      2221
      ```
   4. `P0_test4.fs17` containing multiple strings with some repetitions:
      ```
      1115 121 33 1 233 17 175
      15 27
      ```
2. For each test file, draw by hand the tree that should be generated. For example, `P0_test3.fs17` should create just one node `'1'` with 3 numbers
3. Decide on invocations and what should happen, what should be output filenames if no error, and what the output files should look like - using the hand drawn trees for each file
4. Run the invocations and check against predictions

1. $ P0 < P0_test1
   Error
2. $ P0 < P0_test3.fs17
   Outputs out.inorder out.preorder out.postorder, each containing
   1 111 11 2221
3. $ P0 P0_test3
   Outputs P0_test3.inorder P0_test3.preorder P0_test3.postorder containing as above
4. $ P0 P0_test4
   Outputs as above containing trees as determined by hand

P0 sample on data in the requirements

P0.pdf