1 The Binary Tree ADT

In class, we have developed a binary tree abstract data type whose interface comprises the following components:

- The type `tree_type`, which represents a binary tree.
- The type `tree_item_type`, which represents the type of object stored at each node in a tree.
- The constructor `tree_cons`, which takes a tree item and two trees as arguments, and builds a new tree with the item at its root and the two existing trees as its left and right children.
- The selector `tree_data`, which takes a tree as its argument and returns the data item at the root of that tree.
- The selectors `left_child` and `right_child`, which take a tree as argument and return the left subtree and right subtree of that tree respectively.
- The predicate `empty_tree`, which takes a tree as its argument and returns `TRUE` or `FALSE` according to whether the tree contains any nodes.
- The constant `the_empty_tree`, which represents the tree that contains no nodes.

2 Implementation of Binary Trees

This section presents one implementation of the binary tree ADT presented above. This implementation follows closely our implementation of linked lists. We assume that `tree_item_type` has been defined to be the type of object we want to store at each node of the tree. The underlying data structures are defined as follows:

```
typedef struct tree_tag *tree_type;

typedef struct tree_tag {
  tree_item_type data;
  tree_type left;
  tree_type right;
} tree_struct_type;
```

Because a tree is a pointer in this implementation, it is natural to use `NULL` to represent the empty tree:

```
#define the_empty_tree NULL
```

The code for the five basic functions is straightforward; comparison with the list routines presented in class reveals an almost token–for–token similarity:

```
tree_type
tree_cons(tree_item_type value, tree_type left, tree_type right)
{
  tree_type result;
  result = malloc(sizeof(tree_struct_type));
  assert(result != NULL);
  result->data = value;
  result->left = left;
  result->right = right;
  return(result);
}

tree_item_type
tree_data(tree_type tree)
{
  assert(tree != the_empty_tree);
  return(tree->data);
}

tree_type
left_child(tree_type tree)
{
  assert(tree != the_empty_tree);
  return(tree->left);
}

tree_type
right_child(tree_type tree)
{
  assert(tree != the_empty_tree);
  return(tree->right);
}

BOOL
empty_tree(tree_type tree)
{
  return(tree == the_empty_tree);
}
```

3 Tree Traversal

A *traversal* routine is one that visits every node of a tree, performing a given operation on each node when it is visited. Three orderings of the nodes during traversal are in common use:

- *pre–order traversal:* Each node is visited before any of its descendants is visited.
- *in–order traversal:* A node is visited after all of its left descendants have been visited, but before any of its right descendants is visited.
- *post–order traversal:* Each node is visited after all of its descendants have been visited.

The following code will perform any one of these traversal methods:

```
typedef enum {preorder, inorder, postorder} traversal_type;

void
traversal(tree_type tree, traversal_type method, void (*operation)(tree_type))
{
  if (!empty_tree(tree)) {
    if (method == preorder)
      operation(tree);
    traversal(left_child(tree), method, operation);
    if (method == inorder)
      operation(tree);
    traversal(right_child(tree), method, operation);
    if (method == postorder)
      operation(tree);
  }
}
```

As an example of the use of the `traversal` routine, the following code will print the contents of a tree of integers:

```
void
print_int_tree_data(tree_type tree)
{
  printf("%d  ", tree_data(tree));
}

void
print_int_tree(tree_type tree, traversal_type type)
{
  traversal(tree, type, print_int_tree_data);
  putchar('\n');
}
```

4 Tree Search

A search is like a traversal, except that when a desired node in the tree is encountered, the search is terminated. A search returns the desired node, or `the_empty_tree` if no appropriate node is found.

Two orderings of the nodes in a search are in common use:

- *depth–first search:* Nodes are searched in the same order that they are visited in a pre–order traversal.
- *breadth–first search:* All the nodes at a given level of the tree are searched before any node at a deeper level is searched.

Each of the search routines in the following subsections takes two arguments: a tree to search, and a predicate that determines whether an acceptable node has been found.

4.1 Depth–First Search

Here is an implementation of depth–first search. It uses recursion to search downward in the tree:

```
tree_type
depth_first_search(tree_type tree, BOOL (*predicate)(tree_type))
{
  tree_type result;
  if (empty_tree(tree))
    return(the_empty_tree);
  else if (predicate(tree))
    return(tree);
  else {
    result = depth_first_search(left_child(tree), predicate);
    if (!empty_tree(result))
      return(result);
    else return(depth_first_search(right_child(tree), predicate));
  }
}
```

Here is an implementation of depth–first search that uses an explicit stack. The stack is used to keep track of which nodes are yet to be examined:

```
tree_type
dfs_stack(tree_type tree, BOOL (*predicate)(tree_type))
{
  stack_type stack;
  tree_type guess;
  stack = create_stack();
  push(tree, stack);
  while (!empty_stack(stack)) {
    guess = pop(stack);
    if (empty_tree(guess))
      continue;
    if (predicate(guess))
      return(guess);
    push(right_child(guess), stack);
    push(left_child(guess), stack);
  }
  return(the_empty_tree);
}
```

4.2 Breadth–first Search

Here is an implementation of breadth–first search. Notice that a queue is used instead of a stack; otherwise, this routine is virtually identical to the second implementation of depth–first search:

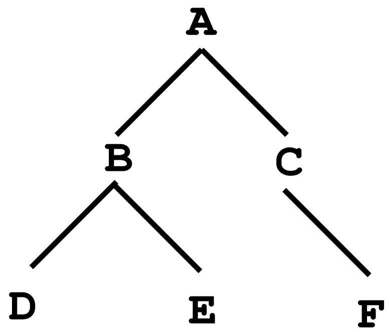
```
tree_type
bfs_queue(tree_type tree, BOOL (*predicate)(tree_type))
{
  queue_type queue;
  tree_type guess;
  queue = create_queue();
  enqueue(tree, queue);
  while (!empty_queue(queue)) {
    guess = dequeue(queue);
    if (empty_tree(guess))
      continue;
    if (predicate(guess))
      return(guess);
    enqueue(left_child(guess), queue);
    enqueue(right_child(guess), queue);
  }
  return(the_empty_tree);
}
```

5 Exercises

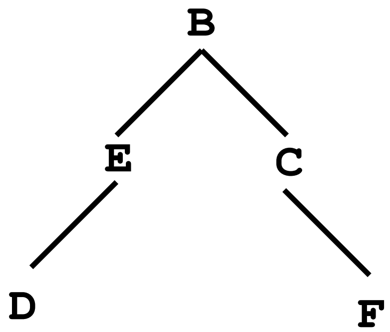
1. Write the `expand_leaf` function. `Expand_leaf` takes three arguments: a binary tree, and two values. It returns a new tree that is identical to the original tree, except that every node that was a leaf in the original tree now has both a left and a right child, whose values are equal to the second and third arguments to `expand_leaf` respectively. `Expand_leaf` must not modify the original tree in any way.

2. Write the `height` function. `Height` takes a binary tree as its argument, and returns the height of the tree. The height of a tree is the maximum level of any node in the tree. The level of a node in a tree is its distance from the root. Assume that the root of a tree has level 0, the children of the root have level 1, *etc.* `Height` should return `-1` if given the empty tree as argument.

3. Write the `delete_root` function. `Delete_root` takes a binary tree as its argument, and returns a new tree that contains every value stored in the original tree except for the value stored at its root. The root of the new tree should be one of the children of the root of the original tree (the other child of the original tree must remain where it was); each child of the root of the new tree should be either a child or a grandchild of the root of the old tree (with nodes not promoted to fill a blank remaining where they were); and so on. For example, if `delete_root` were called on the tree:



it might return the tree:



Note that other solutions are possible; your program may return any legal solution. If `delete_root` is called on the empty tree, or on a tree containing exactly one node, it should return the empty tree. `Delete_root` must not modify its argument in any way; it should produce a brand new tree.