1 Bitdiddle Post-Breadth Tree Traversal [ / 31 ]

1.1 Balanced Tree Example [ / 3 ]

Ben Bitdiddle really wants to get his name on a traversal ordering. Even without a real world application for its use, he has invented what he calls the post-breadth ordering. His primary demonstration example is an exactly balanced, binary search tree with the numbers 1-15.

Your first task is to make a neat diagram of this tree in the box on the right.

For this example, Ben decrees that the PrintPostBreadth function should output:

Solution:

```
LEVEL 0: 1 3 5 7 9 11 13 15
LEVEL 1: 2 6 10 14
LEVEL 2: 4 12
LEVEL 3: 8
```

1.2 Un-Balanced Tree Example [ / 3 ]

Alyssa P. Hacker rolls her eyes at Ben but agrees to help him with the implementation. However, before tackling the implementation she wants to make sure that Ben’s idea is sound. She sketches the unbalanced tree shape on the left.

Your second task is to place the numbers 1-10 in this diagram so it is a proper binary search tree.

This unbalanced tree initially confuses Ben. But he thinks for a while and decides that for his new traversal ordering, level 0 is defined to be all of the leaves of the tree, level 1 is the parents of the leaves, level 2 is the grandparents, etc. So he decrees that for this second example, the output of the PrintPostBreadth function is:

Solution:

```
LEVEL 0: 2 5 7 9
LEVEL 1: 3 4 8 10
LEVEL 2: 1 6
```

Alyssa studies Ben’s sample output carefully and then asks Ben if the traversal ordering will ever contain repeated elements. Ben says no, each element in the structure should be output exactly once. Alyssa suggests that they add a boolean mark member variable to the Node class since it will be helpful for an efficient implementation. This flag will help ensure the traversal ordering does not contain duplicates.
1.3 CollectLeaves Implementation

Alyssa’s Node class is on the right. She further suggests starting with the implementation of a helper function named CollectLeaves. This is a void recursive function that takes in two arguments: ptr is a pointer to a Node (initially the root of the tree), and leaves is an STL list of pointers to Nodes (the list is initially empty) that will collect all of the leaves of the tree.

She also indicates that this function should initialize all of the mark variables. Only the leaf nodes should be marked true.

Complete the implementation below.

```cpp
class Node {
public:
    // CONSTRUCTOR
    Node(int v) : value(v), mark(false), left(NULL), right(NULL), parent(NULL) {} 
    // REPRESENTATION
    int value;
    bool mark;
    Node* left;
    Node* right;
    Node* parent;
};
```

```cpp
void CollectLeaves(Node *ptr, std::list<Node*>& leaves) {
    if (ptr == NULL) return;
    if (ptr->left == NULL && ptr->right == NULL) {
        ptr->mark = true;
        leaves.push_back(ptr);
    } else {
        ptr->mark = false;
        CollectLeaves(ptr->left, leaves);
        CollectLeaves(ptr->right, leaves);
    }
}
```

1.4 PrintPostBreadth Implementation

Now finish the implementation of the PrintPostBreadth function:

Solution:

```cpp
void PrintPostBreadth(Node *root) {
    // call the helper function
    std::list<Node*> current;
    CollectLeaves(root, current);
    int count = 0;
    while (current.size() > 0) {
        std::cout << "LEVEL " << count << " : " << std::endl;
        // prepare a list of the parents of the current level
        std::list<Node*> next;
        for (std::list<Node*>::const_iterator itr = current.begin(); itr != current.end(); itr++) {
            std::cout << " " << (*itr)->value;
            if ((*itr)->parent != NULL && !(*itr)->parent->mark) {
                next.push_back((*itr)->parent);
                (*itr)->parent->mark = true;
            }
        }
        // increment & switch to the new level
        count++;
        current = next;
        std::cout << std::endl;
    }
}
```
Louis B. Reasoner has taken a job at a genome sequencing startup working on algorithms to detect differences between the genomes of different species. He came up with the sketch of the data structure on the right and showed it to his manager and got approval to start implementation.

He’s defined two typedefs named `count_t` and `kmer_t` to improve the readability of his code. Here’s an example of how this data structure is constructed using the `Add` function:

```c
kmer_t kmers;
count_t totals;
Add(totals,kmers,"human","ACT");
Add(totals,kmers,"human","ACT");
Add(totals,kmers,"human","ACT");
Add(totals,kmers,"human","GAG");
Add(totals,kmers,"human","TAG");
Add(totals,kmers,"human","TAG");
Add(totals,kmers,"human","TAG");
Add(totals,kmers,"dog","ACT");
Add(totals,kmers,"dog","ACT");
Add(totals,kmers,"dog","GAG");
Add(totals,kmers,"dog","TAG");
Add(totals,kmers,"dog","TAG");
Add(totals,kmers,"dog","TAG");
Add(totals,kmers,"dog","TAG");
Add(totals,kmers,"fruit fly","ACT");
Add(totals,kmers,"fruit fly","ACT");
Add(totals,kmers,"fruit fly","ACT");
Add(totals,kmers,"fruit fly","GAG");
```

Two of the key operations for this data structure are to query the number of matches of a given k-mer for a particular species and to find the most frequently occurring k-mer for a species. Here are several example usages of the `Query` and `MostCommon` functions:

```c
assert (Query(kmers,"human","ACT") == 3);   assert (MostCommon(kmers,"human") == "TAG");
assert (Query(kmers,"human","CAT") == 0);   assert (MostCommon(kmers,"fruit fly") == "ACT");
assert (Query(kmers,"human","TAG") == 4);   assert (MostCommon(kmers,"cat") == ");
assert (Query(kmers,"dog","ACT") == 1);
```

Finally, we can compute the difference between two species. The **k-mer fraction** is the percent of a species total k-mers that match the particular k-mer. The **k-mer difference** is the absolute value of the difference between the k-mer fractions for each of the species. And the overall difference between two species is the sum over all k-mers of the k-mer difference. Here is the math to calculate the difference between a human and a dog:

- **ACT**: \( \frac{2}{5} - \frac{3}{9} = 0.067 \)
- **CAT**: \( 0 \)
- **GAG**: \( \frac{1}{5} - \frac{2}{9} = 0.022 \)
- **TAG**: \( \frac{2}{5} - \frac{4}{9} = 0.044 \)
- **Overall**: \( = 0.133 \)

Here is code to call the `Difference` helper function:

```c
std::cout << "Difference between human & dog 
        " << Difference(totals,kmers,"human","dog") << std::endl;
std::cout << "Difference between human & fruit fly 
        " << Difference(totals,kmers,"human","fruit fly") << std::endl;
std::cout << "Difference between dog & fruit fly 
        " << Difference(totals,kmers,"dog","fruit fly") << std::endl;
```

And the resulting output:

- Difference between human & dog 0.133
- Difference between human & fruit fly 0.889
- Difference between dog & fruit fly 0.800
2.1 The typedefs [ / 4 ]

First, fill in the typedef declarations for the two shorthand types used on the previous page.

Solution:

typedef std::map<std::string, int> count_t;
typedef std::map<std::string, count_t> kmer_t;

2.2 Add Implementation [ / 7 ]

Next, finish the implementation of the Add function.

Solution:

```cpp
void Add(count_t& totals, kmer_t& kmers, const std::string& species, const std::string& kmer) {
    totals[species]++;
    kmers[kmer][species]++;
}
```

If the data structure contains $s$ different species, and $k$ unique k-mers, and each animal contains $p$ total k-mers, what is the order notation for the running time of a single call to Add? Write 2-3 concise and well-written sentences justifying your answer.

Solution: The first operator[] costs $O(\log s)$ because there are $s$ species in the map. The second operator[] costs $O(\log k)$ because there are $k$ unique k-mers in the outer k-mers map. The third operator[] costs $O(\log s)$ because there are $s$ species in the inner k-mers map. These quantities are simply added together (we don’t search every inner map, just one!). Overall: $O(\log s + \log k)$.

2.3 Query Implementation [ / 6 ]

Solution:

```cpp
int Query(const kmer_t& kmers, const std::string& species, const std::string& kmer) {
    kmer_t::const_iterator itr = kmers.find(kmer);
    if (itr == kmers.end())
        return 0;
    count_t::const_iterator itr2 = itr->second.find(species);
    if (itr2 == itr->second.end())
        return 0;
    return itr2->second;
}
```

2.4 MostCommon Implementation [ / 7 ]

Solution:

```cpp
std::string MostCommon(const kmer_t& kmers, const std::string &species) {
    std::string answer = "";
    int count = -1;
    for (kmer_t::const_iterator itr = kmers.begin(); itr != kmers.end(); itr++) {
        count_t::const_iterator itr2 = itr->second.find(species);
        if (itr2 != itr->second.end() &&
            (answer == "" || count < itr2->second)) {
            answer = itr->first;
            count = itr2->second;
        }
    }
    return answer;
}
```
2.5 Difference Implementation [ / 12 ]

Solution:

```cpp
float Difference(const count_t& totals, const kmer_t& kmers,
                 const std::string& speciesA, const std::string& speciesB) {
    float diff = 0;
    count_t::const_iterator itrA = totals.find(speciesA);
    count_t::const_iterator itrB = totals.find(speciesB);
    if (itrA == totals.end() || itrB == totals.end()) {
        std::cerr << "ERROR! One or both species are unknown" << std::endl;
        return -1;
    }
    float totalA = itrA->second;
    float totalB = itrB->second;
    for (kmer_t::const_iterator itr = kmers.begin(); itr != kmers.end(); itr++) {
        int countA = Query(kmers,speciesA,itr->first);
        int countB = Query(kmers,speciesB,itr->first);
        diff += fabs(countA/float(totalA)-countB/float(totalB));
    }
    return diff;
}
```

If the data structure contains $s$ different species, and $k$ unique k-mers, and each animal contains $p$ total k-mers, what is the order notation for the running time of a single call to Difference? Write 2-3 concise and well-written sentences justifying your answer.

Solution: Finding the two species in the totals map is $O(\log s)$. We loop over all $k$ unique k-mers, and for each of them (multiplication) we look up the species in the inner map. If we use Query (we did above), this will be $O(\log s + \log k)$. If instead we inline a portion of this function (more code, but faster) it will be $O(\log s)$.

Overall (w/ Query): $O(\log s + k \times (\log s + \log k))$, which simplifies to $O(k \times (\log s + \log k))$.
Overall (w/o Query): $O(\log s + k \times \log s)$, which simplifies to $O(k \times \log s)$.

3 Prescribed Prefix Ordering [ / 21 ]

In this problem we will create an algorithm to construct a binary search tree from the desired prefix traversal order. The driver function (below) takes in this sequence as a STL `vector`. If the contents of the vector is not a valid prefix traversal order of a binary search tree, the function should return NULL.

```cpp
template <class T> class Node {
public:
    Node(T v) : value(v), left(NULL), right(NULL) {}
    T value;
    Node* left;
    Node* right;
};

template <class T> void destroy(Node<T>* root) {
    if (root == NULL) return;
    destroy(root->left);
    destroy(root->right);
    delete root;
}

// "driver" function (starts the recursive function that does the actual work)
template <class T> Node<T>* MakePrefixTree(const std::vector<T>& values) {
    if (values.size() == 0) return NULL;
    return MakePrefixTree(values, 0, values.size() - 1);
}
```
3.1 Test Cases

First, create 4 different test cases of input for this problem. Each input vector should contain the numbers 1-7. The first two should be valid prefix orderings for a binary search tree containing these 7 numbers. Draw the corresponding tree for these cases. The other two test case inputs should be invalid prefix orderings.

valid: 4 2 1 3 6 5 7    valid: 2 1 3 4 5 6 7

valid: 4 2 1 3 6 5 7    valid: 2 1 3 4 5 6 7

invalid: 5 4 3 6 2 1 7    invalid: 4 6 5 7 2 1 3

Solution: (many correct answers!)

3.2 Finish the MakePrefixTree Implementation

Note: If you discover the input sequence is an invalid prefix ordering for a binary search tree, make sure you do not leak any memory!

Solution:

```cpp
template <class T>
Node<T>* MakePrefixTree(const std::vector<T>& values, int start, int end) {
    assert (start <= end);
    // find the split between the left & right branches
    int split = start+1;
    // the split is the first element that is greater than the "root"
    while (split <= end && values[split] < values[start]) {
        split++;
    }
    // check that all elements after the split are also greater than the "root"
    for (int i = split; i <= end; i++) {
        if (values[i] < values[start]) {
            // failure
            return NULL;
        }
    }
    // make the new node
    Node<T>* answer = new Node<T>(values[start]);
    // if there is at least one node to the left, recurse left
    if (start+1 <= split-1) {
        answer->left = MakePrefixTree(values,start+1,split-1);
        // if the left tree is NULL (failure), cleanup
        if (answer->left == NULL) {
            destroy(answer);
            return NULL;
        }
    }
    // if there is at least one node to the right, recurse right
    if (split <= end) {
        answer->right = MakePrefixTree(values,split,end);
        // if the right tree is NULL (failure), cleanup
        if (answer->right == NULL) {
            destroy(answer);
            return NULL;
        }
    }
    return answer;
}
```
4 iClicked, uClicked, not-quite-all-of-usClicked [ 0 ]

Grading Note: -2pts each iclicker unanswered or incorrect.

4.1 Which of the following statements about binary search tree iterators is false?
(A) When we are part way through an iteration of a tree using the vector or stack of pointers method, and someone calls insert or erase on that tree, we must assume the iterator is now invalid.
(B) Once we have a fully debugged \texttt{operator++} for tree iterators, we can simply swap all the \texttt{lefts} & \texttt{rights} to implement \texttt{operator--}.
(C) If the tree iterator is pointing at the node containing the last element in sorted order, that node must be a leaf node.
(D) Sometimes the “next” node in the tree is $O(\log n)$ steps away!
(E) None of the above.

Solution: C

4.2 Which of the following statements about STL container types is true?
(A) A program that uses an STL \texttt{vector} can easily be changed to use an STL \texttt{list} instead, with little or no performance impact.
(B) A program that uses an STL \texttt{set} can easily be changed to use an STL \texttt{map} instead, with little or no performance impact.
(C) A program that uses an STL \texttt{set} can easily be changed to use an STL \texttt{list} instead, with little or no performance impact.
(D) A program that uses an STL \texttt{map} can easily be changed to use an STL \texttt{vector} of STL \texttt{pairs} instead, with little or no performance impact.
(E) A program that uses an STL \texttt{map} can easily be changed to use an STL \texttt{set} instead, with little or no performance impact.

Solution: B

4.3 Which of the following is true for the STL map iterators?
(A) Data is accessed in the order it was inserted.
(B) Since STL map has an \texttt{operator[]}, it is like STL vector and thus I can move a map iterator forward not just one spot (using \texttt{itr++}), but I can also jump forward an arbitrary number of spots (e.g., 25) using \texttt{itr + 25}.
(C) STL list and STL map iterators are typedef-ed as simple pointers to a data element in the container.
(D) Visiting every element in an STL map is faster than visiting every element in an STL vector.
(E) None of the above.

Solution: E

4.4 Which of the following statements is least true?
(A) STL Maps are magical!
(B) In C++, local integer variables are not initialized to 0.
(C) If you can code something with fewer keystrokes it is always better software.
(D) STL Maps are like Python dictionaries.
(E) If your data contains duplicates and/or you don’t want it sorted (you need it in a specific non-sorted order), you probably want an STL vector (or list), STL maps probably won’t be helpful.

Solution: C

4.5 Which of the following is false for the STL pair class?
(A) The first item in an STL pair is always const and cannot be changed.
(B) I don’t need to put \texttt{#include <utility>} if I’m using an STL map in the same file (because it is indirectly included with \texttt{#include <map>}).
(C) Pairs are cool, they can be used to return two values from a function (more intuitive than that weird trick using pass-by-reference arguments).
(D) \texttt{first} and \texttt{second} are public member variables of the STL pair struct, therefore they can be accessed and edited directly by the user (we don’t need to use a \texttt{get} or \texttt{set} member function).
(E) For a previous homework I have (or could have easily) written my own STL pair-like class glueing together two related items.

Solution: A

4.6 Which of the following guidelines for designing operator overloading is false?
(A) Don’t change the intuitive meaning of an operator.
(B) Only use friend functions if a function cannot be implemented as a member function and implementation as a non-member function would require writing accessors/modifiers to private data that are otherwise unnecessary and poor class design.
(C) You can overload operators for every symbol on your keyboard!
(D) The input & output stream operators should return the stream object by reference so calls may be nested/chained.
(E) Do not attempt to return a local variable by reference. The compiler will give you a warning that should not be ignored.

Solution: C