Assignment 4

CS 1037
Computer Science Fundamentals II

Due date: November 28, 2024, at 11:59 PM

Learning Outcomes

Upon completion of this assignment, students will be able to:

- Implement AVL trees, m-way trees, and hash tables for efficient data management.
- Perform insertion, deletion, merging, and traversal operations on AVL trees.
- Optimize search and indexing using trees and hashing.
- Develop modular, reusable, and efficient code.

Preparation

You should use a working directory called 'asn4' to store programs for this assignment. Check the submission requirement at the end of this document for the required submission program files and how to submit them.

Introduction

This assignment aims to enhance your C programming skills by designing and implementing efficient data structures and algorithms for managing hierarchical and associative data. You'll work with m-way trees, AVL trees, and hash tables to perform insertion, search, deletion, traversal, and aggregation operations, ensuring structural integrity and optimized performance for various applications.

Question 1 – AVL Tree for Record Data Processing [15 points]

Write C programs *myrecord_avl.h* and *myrecord_avl.c* to manage record data using AVL trees. You will also implement functions to merge two AVL trees, compute statistics, and clean data structures. You will need to use *avl.h* and *avl.c*, provided in the OWL assignment folder.

Functional Requirements

Data Structure

`AVLDS`: Holds the root pointer of the AVL tree and statistical metrics (count, mean, standard deviation) of the scores.
Defined as:

```
typedef struct {
    AVLNODE *root;
    int count;
    float mean;
    float stddev;
} AVLDS;
```

Function Implementations:

1. **`merge_avl`** – Merges a source AVL tree into a destination AVL tree without modifying the source tree.
    - Parameters:
        - rootp_dest: Pointer to the root of the destination tree.
        - rootp_source: Pointer to the root of the source tree.
    - Returns: Updates the destination tree with all nodes from the source tree.
2. **`merge_avlds`** – Merges two AVLDS structures.
    - Parameters:
        - `source`: Pointer to the source `AVLDS`.
        - `dest`: Pointer to the destination `AVLDS`.
    - Returns:
        - Updates `dest` with all nodes from `source`, recalculating statistics (count, mean, and stddev).
        - Clears `source`.
3. **`avlds_clean`** – Clears the AVL tree in the given `AVLDS` structure.
    - Parameters:
        - `ds`: Pointer to the `AVLDS` structure.
    - Returns: Resets the AVL tree and metrics (`count=0`, `mean=0`, `stddev=0`).
4. **`add_record`**
    - Adds a record to the AVLDS structure.
    - Updates the AVL tree and recalculates the statistics.
    - Parameters:
        - `ds`: Pointer to the `AVLDS`.
        - `data`: Record to be added.
    - Returns: 1 if the record is added; 0 otherwise.
5. **`remove_record`**
    - Removes a record from the AVLDS structure by name.
    - Updates the AVL tree and recalculates the statistics.
    - Parameters:
        - `ds`: Pointer to the `AVLDS`.
        - `name`: Name of the record to be removed.
    - Returns: 1 if the record is removed; 0 otherwise.
6. **`calculate_stats`**
    - This function calculates the mean and standard deviation of the scores stored in the AVL tree represented by the AVLDS structure.
    - It traverses the tree to sum the scores and their squares, which are then used to compute the mean and standard deviation.
    - Parameters:
        - `ds`: Pointer to the `AVLDS`.

Note—This function checks for an empty tree, traverses the tree to compute sums, and calculates the mean and standard deviation. It is typically called after any modifications to the AVL tree to ensure accurate statistics.

7. **`avl_count_nodes:`** Count the number of nodes in the tree
   - Parameters:
     - `rootpt`: Pointer to the root of the AVL tree
   - Returns: Count of all nodes in a tree

Compilation and Output:

Create a test program named `myrecord_avl_ptest.c` that will be used while compiling and running commands as follows:

- `gcc avl.c myrecord_avl.c myrecord_avl_ptest.c -o q1`
- `./q1`

Expected Output:

```
Test: merge_avl
is_avl(avlA): 1
is_avl(avlB): 1
merge_avl(avlA avlB): 1

Test: merge_avlds
display_stats(avldsA): count 10 mean 55.0 stddev 28.7
display_stats(avldsB): count 10 mean 55.0 stddev 28.7
display_stats(avldsC): count 20 mean 55.0 stddev 28.7
merge_avlds(avldsA avldsB)-display_stats(avldsA): count 20 mean 55.0
stddev 28.7
avldsA.count: 20
avldsA.mean: 55.0
avldsA.stddev: 28.7
avldsB.count: 0
avldsB.mean: 0.0
avldsA.count==avldsC.count: 1
avldsA.mean==avldsC.mean: 1
avldsA.stddev==avldsC.stddev: 1
```

Additional Note:

- Ensure AVL tree properties are maintained after every operation.
- The statistics (count, mean, and standard deviation) must be updated efficiently with each operation.
- You can add any helper function to the `myrecord_avl.h` if you find it necessary and will help generate the expected output.

Question 2 – M-Way Tree Implementation [10 points]

Write C programs *mway_tree.h* and *mway_tree.c* to implement an m-way search tree. The implementation should include fundamental operations such as adding keys, searching for keys, performing in-order traversal, and displaying the tree structure.

Functional Requirements

Data Structure

M-Way Node (TNODE): Represents a node in the m-way tree.
Defined as:

```
typedef struct node {
    int count; // number of keys in the node
    int key[m-1]; // array to hold up to m-1 keys
    struct node *child[m]; // array of pointers to child nodes
} TNODE;
```

Here, m a predefined constant represents the order of the tree.

Function Implementations

1. **`create_node`**
   - Allocates memory for a new m-way tree node.
   - Initializes key count to 0 and child pointers to NULL.
   - Parameters: None.
   - Return: Pointer to the newly created node.
2. **`insert_key`** – Inserts a key into the appropriate position in the m-way tree while maintaining sorted order.
   - Parameters:
     - `root`: Pointer to the root node.
     - `key`: Key to be inserted.
   - Return: Updates the tree structure.
3. **`search_key`** – Searches for a key in the m-way tree.
   - Parameters:
     - `root`: Pointer to the root node.
     - `key`: Key to search for.
   - Return: 1 if the key is found; 0 otherwise.
4. **`print_inorder`** – Performs an in-order traversal of the m-way tree, visiting keys in increasing order.
   - Parameters:
     - `root`: Pointer to the root node.
   - Output: Prints keys in sorted order.

5. **`display_tree`:**
   - Displays the tree structure, showing keys and their respective child pointers.
   - Parameters:
     - `root`: Pointer to the root node.
6. **`delete_key`:**
   - Deletes a key from the m-way tree and rebalances it if necessary.
   - Parameters:
     - `root`: Pointer to the root node.
     - `key`: Key to be deleted.
   - Return: 1 if the key is removed; 0 otherwise.

Compilation and Output:

Create a test program named `mway_tree_ptest.c` that will be used while compiling and running commands as follows:

- `gcc mway_tree.c mway_tree_ptest.c -o q2`
- `./q2`

Expected Output:

```
Test: Insert and Display
insert_key(root, 22)
insert_key(root, 5)
insert_key(root, 10)
display_tree(root):
Node keys: [5, 10, 22]
Child pointers: [NULL, NULL, NULL, NULL]

Test: In-Order Traversal
print_inorder(root): 5 10 22

Test: Search Key
search_key(root, 10): Found
search_key(root, 15): Not Found

Test: Delete Key
delete_key(root, 10)
display_tree(root):
Node keys: [5, 22]
Child pointers: [NULL, NULL, NULL]
```

Additional Note:

- Ensure tree properties are maintained during all operations.

- Handle edge cases such as duplicate key insertion and deletion of non-existent keys.

Question 3 – Hash Table Implementation [15 points]

Write C programs *hash.h* and *hash.c* to implement a chained hash table that stores HASHDATA key-value pairs. Define the structures and function prototypes in *hash.h* and implement the functions in *hash.c*.

Functional Requirements

Data Structures

Define a structure HASHDATA with the following fields:

- `key` – a string of up to 20 characters representing the key.
- `value` – an integer value associated with the key.

Define a structure HASHNODE to represent a linked list node within the hash table:

- `key` – a string with the key.
- `value` – the integer value associated with the key.
- `next` – a pointer to the next node in the list.

Define a structure HASHTABLE for the chained hash table:

- `hna` – a pointer to an array of pointers to HASHNODE nodes.
- `size` – an integer representing the size of the hash table (i.e., the number of buckets).
- `count` – an integer representing the number of elements stored in the hash table.

Function Implementations

Implement the following functions to manage the hash table:

1. **`hash`**: This function hashes a key string to an integer using the hash table's size to determine the index.
    - Parameters:
        - `key` – the input key string.
        - `size` – the size of the hash table.
    - Return: An integer index in the range [0, size-1] is calculated as the sum of ASCII values of the characters in key modulo size.
2. **`new_hashtable:`** This function creates a new hash table of a given size.
    - Parameters:
        - `size` – the size of the hash table.
    - Return: A pointer to the newly created HASHTABLE structure.

- **`hashtable_insert:`** This function inserts a HASHDATA entry into the hash table, ensuring that entries within each linked list are sorted by key.
    - Parameters:
        - `ht` – a pointer to the hash table.
        - `key` – the key string.
        - `value` – the integer value associated with the key.
    - Return: Returns 0 if the key already exists and updates its value. Otherwise, it inserts a new entry and returns 1.

- **`hashtable_search:`** This function searches for a HASHNODE in the hash table with the specified key.
    - Parameters:
        - `ht` – a pointer to the hash table.
        - `key` – the key string to search for.
    - Return: A pointer to the HASHNODE if found or NULL if not found.

- **`hashtable_delete:`** This function deletes a node from the hash table where the key matches the specified key.
    - Parameters:
        - `ht` – a pointer to the hash table.
        - `key` – the key string of the node to delete.
    - Return: Returns 1 if the node was successfully deleted or 0 if the key was not found.
- **`hashtable_clean:`** This function deletes all nodes in the hash table, clears all linked lists, and resets the count to zero.
        - Parameters: `ht` – a pointer to a pointer to the hash table.

Compilation and Output:

Create a test program named `hash_ptest.c` that will be used while compiling and running commands as follows:

- `gcc hash.c hash_ptest.c -o q3`
- `./q3`

Expected Output:

```
Test for hash:
hash(a): 2
hash(b): 3
hash(c): 4
hash(d): 0

Test for new_hashtable:
new_hashtable(5): size 5 count 0

Test for hashtable_insert:
hashtable_insert(a 0): count 1 2 (a 0)
hashtable_insert(b 1): count 2 2 (a 0) 3 (b 1)
hashtable_insert(d 3): count 3 0 (d 3) 2 (a 0) 3 (b 1)

Test for hashtable_search:
search(a): a
search(b): b
search(f): NULL
search(h): NULL

Test for hashtable_delete:
hashtable_delete(a):
count 6 0 (d 3) 1 (e 4) 2 (f 5) 3 (b 1) (g 6) 4 (c 2)
hashtable_delete(b):
count 5 0 (d 3) 1 (e 4) 2 (f 5) 3 (g 6) 4 (c 2)
```

What to submit?

- Submit your completed files as follows:
  1. Question 1: `myrecord_avl.c and myrecord_avl.h`
  2. Question 2: `mway_tree.c and mway_tree.h`
  3. Question 3: `hash.c and hash.h`

  through the designated dropbox on OWL Brightspace.
- Do not submit any other files including executables.
- Ensure your code compiles and runs without errors on the university lab computers.

Submission Platform

- All submissions must be made through OWL Brightspace. Please do not email your files to the instructors or TAs, as emailed submissions will not be marked.

Deliverables

- Place all required .c files into a single folder.
- Name the folder with your student number (e.g., 12345678).
- Compress the folder into a .zip file with the same name (e.g., 12345678.zip).

Resubmissions: You may resubmit your assignment as many times as needed before the deadline. Your last submitted version will be considered for grading. In addition, any resubmissions made after the deadline will incur late penalties.

Assignment 4

Compatibility Reminder: Ensure your code compiles and runs on the university lab computers. Code that only works on your local IDE may result in lost marks.

Additional Rules

- Do not upload the .exe files! Penalties will be applied for this.
- Submit the assignment on time. Late submission policy will be applied.
- Forgetting to submit is not a valid excuse for submitting late.
- Assignment files must NOT be emailed to the instructor(s) or TA(s). They will not be marked if sent by email.

Files to submit

Remember **you must do** all the work on your own. **Do not copy** or even look at the work of another student. All submitted code will be run through similarity-detection software.

Your submission will be evaluated using the grading criteria. To maximize your score, ensure that your implementation addresses all the aspects outlined above.

Good Luck ☺