Final Exam

COP3502 Computer Science 1 – Summer 2022

Reflective Questions (50 points)

1. Merge sort has the best big-O performance of the fundamental sorting algorithms. Why do we switch away from it for small lists? Although merge sort is the quickest sorting algorithm, we switch away from it when we're dealing with small lists because of its space complexity. Whenever we merge our list, our merge function creates one or two temporary arrays, which stores and compares some n value, then modifies our list.

2. Explain what happens when a binary search tree becomes sufficiently unbalanced; explain the negative consequences for its purpose *as* a binary search tree; and explain how an AVL tree prevents them. When a binary search tree becomes sufficiently unbalanced, its performance declines. Finding an element in the tree could take O(n) in its worst case, instead of O(log n). An AVL tree prevents this from happening, ensuring that our tree is balance, which ultimately increase our overall performance to O(log n).

3. Consider a tree that has a relatively high (between 10 to 30) typical number of children of each node. Under what conditions would a static child pointer array implementation be usable, and under what conditions would it be a better choice? Under what conditions would a dynamic child pointer array implementation be usable, and under what conditions would it be a better choice? A static child pointer array will be usable when we're not too worry about memory usage. If each parent in the tree is guaranteed to have n number of children, we can implement our tree using the stack. However, if each parent is not guaranteed to have the same number of children, a dynamic child pointer array will be the better choice. We'll only be creating space for as many children needed for the specific parent. This implementation guarantees that we're not wasting memory.

4. Compare and contrast a leftmost-child-right-sibling tree with a linked list. What are the similarities and differences? A linked list is a linear data structure type, whereas a leftmost-child-right-sibling tree is non-linear. However, in a LCRS tree, the children nodes behave like a linked-list, where each child points to the next child within the subtree.

5. Give a task that a binary search tree would be suitable for that a heap would not. Explain why.

   A binary search tree is better suited for searching, given that its items are already sorted. It takes about O(log n) to search, whereas a heap will take O(n) in its worst case. Although each parent in a heap holds a value greater (maxheap) or less (minheap) than its children, we'll need to traverse the heap, like we do in a linked list, to find our item.

Performance Analysis (25 points)

1. A naïve linear searching algorithm with negligible startup overhead takes two tenths of a second to find an item in a list of 75,000 entries. The time budget for the searching algorithm in your program is three fourths of a second. How big does the list of entries need to get before you need a better searching algorithm?

   2/10 = 1/5 of a second = 200 ms   big-O of Linear Search = O(n)   ¾ of a second = 750ms

   75,000x = 200m   x = 1/375   1/375n = 750ms   n = 281,250 entries

   The list will need to have 281,250 entries before you need a better searching algorithm.

2. A binary searching algorithm has logarithmic $\mathcal{O}(\log n)$ performance. It takes ten seconds to find an item in a list of 10,000 entries. Somebody decided to throw it onto a server with a moderately large database without modification, and it is now searching a list of 100,000,000,000 entries. How long do you expect it to take to find an item? How did you arrive at your answer?

   Big-O of Binary Search = O(log n)   log 10,000 = 4   log 100,000,000,000 = 11

   10 x 11/4 = 27.5 seconds. **It will take about 27.5 seconds to find an item**

3. Given the location of the parent, what big-$\mathcal{O}$ time does adding a child take for a dynamic-child-pointer-array tree? Is it different if we want to insert a child into the middle of the existing children? Why? It would take about O(n) to insert a child into a dynamic-child-pointer-array-tree. Inserting a child into the middle of the existing children doesn't improve or worsen our time performance. We'll still need to traverse the children list to find our middle child.

4. Does implementing a trie using a dynamic child pointer array rather than a static child pointer array increase **the big-$\mathcal{O}$ time** of lookups? Why or why not? Implementing a trie using a dynamic child pointer array doesn't necessarily improve the big-O time of lookups. Instead, it helps the programmer make sure that they're not wasting memories each time a child node is created and inserted.

5. Consider an open hash (that is, one that deals with collisions by using a linked list) with half a billion entries. The hash algorithm takes 200ns to run, each comparison of objects takes 10ns, and everything else involved in the lookup takes trivial time. How large does the hash table need to be to use less time on chained comparisons than on the initial hash lookup? **(Note that you'll always have to compare once!)**

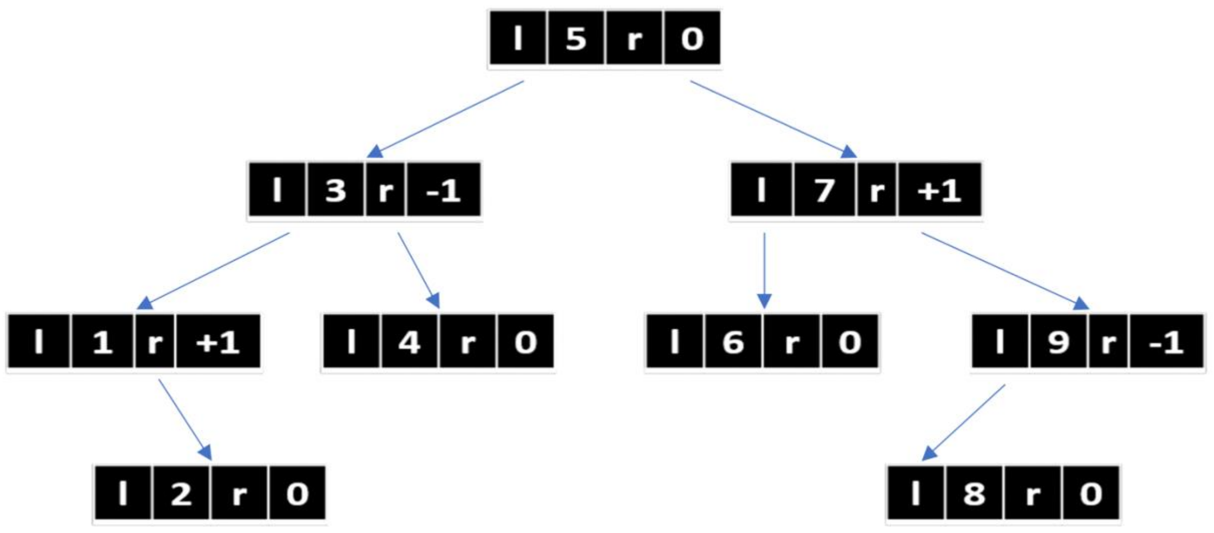
   Hashes Big-O = O(1)   1x = 200ns   1/200 = 1ns

   Comprarison Big-O = O(n)   500,000,000x = 10ns   50,000,000 = 1ns

   1/200 * 50,000,000 = **250,000 entries**

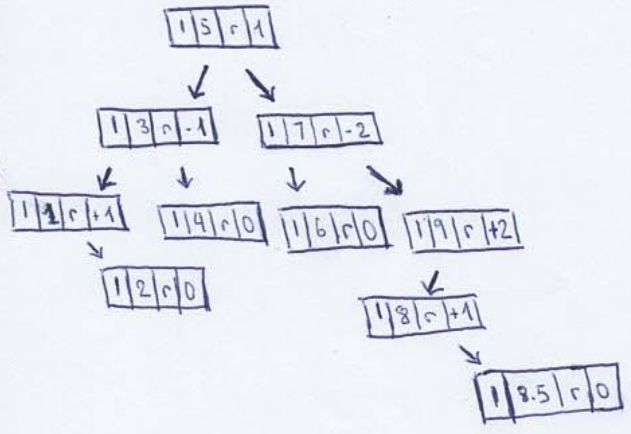
AVL Trees (25)

Consider the following familiar AVL tree:

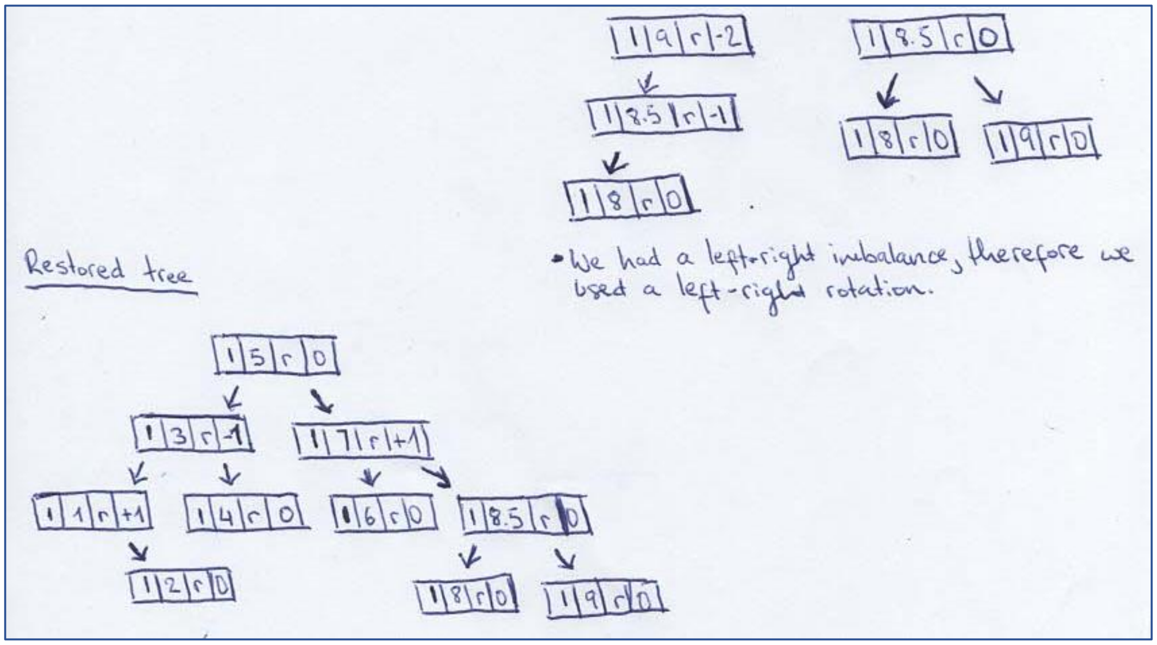


- (10) Add the value 8.5, then restore the AVL tree property. Show the tree before and after the rotations you use to restore it.

Gradi Tshielekeja Mbuyi



- We've added 8.5 to the AVL tree
- Now we have to perform a double rotation

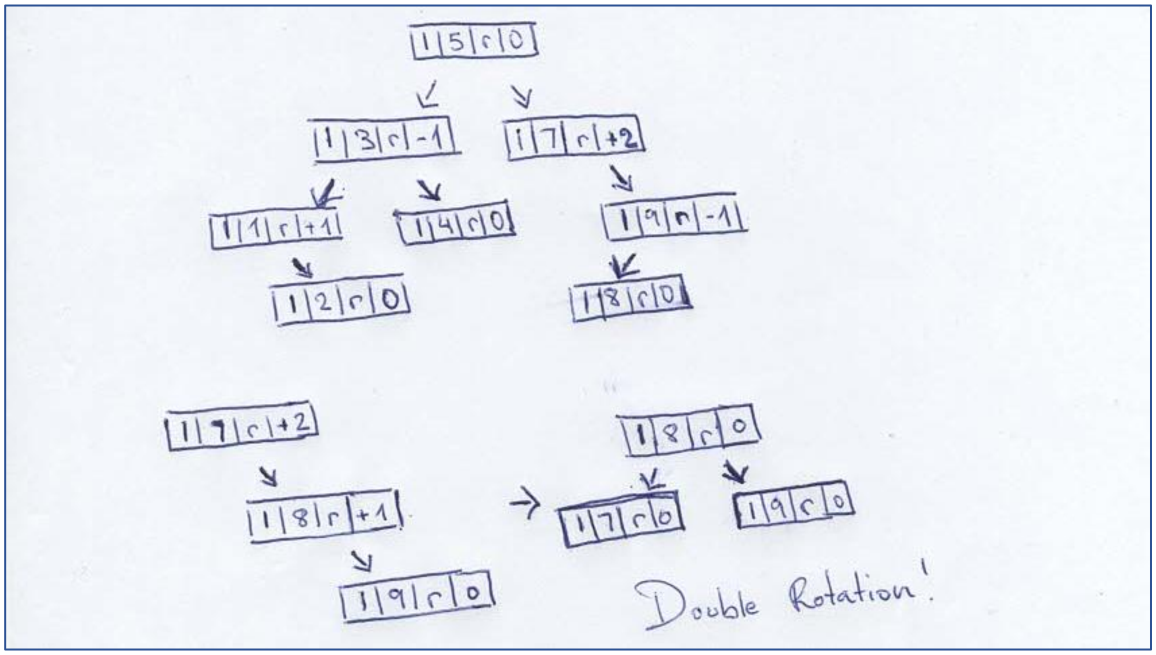


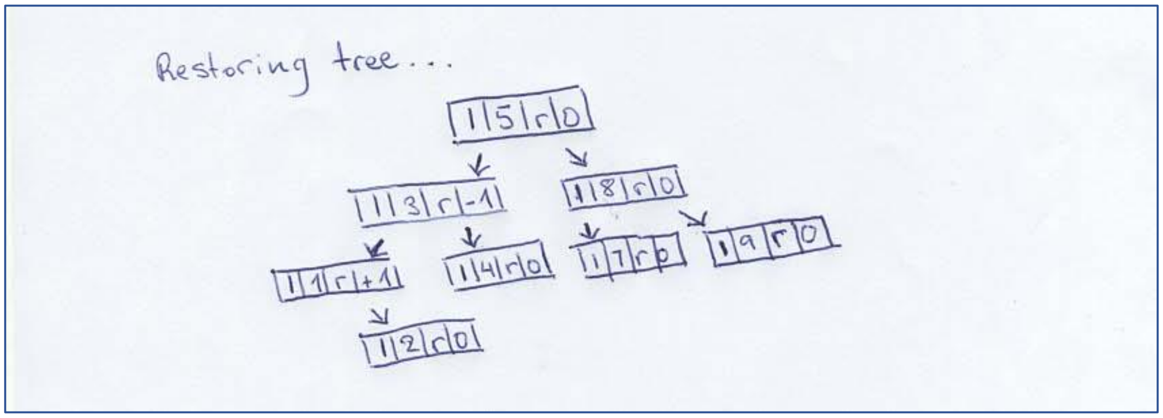
• We had a left-right imbalance, therefore we used a left-right rotation.

- (5) Would adding the value 4.5 to the tree require rotations? Why or why not?

  Adding the value 4.5 wouldn't require a rotations, because the height of the AVL tree will remain balanced.

- (10) Going back to the original tree, delete the value 6, then restore the AVL tree property. Show the tree before and after the rotations you use to restore it.





Hashes (25)

Considering the following actual monster structure, implicit linked list structure, and list functions pre-defined for you:

```
struct monster_struct {
    char name[64];
    char type[64];
    unsigned short int weight;
    unsigned short int monsteriness;
};
typedef struct monster_struct monster;

monster_linked_list *mll_new();
void mll_add(monster_linked_list *l, monster *m);
void mll_remove(monster_linked_list *l, monster *m);
monster *mll_find(monster_linked_list *l, char *name);
```

...implement an open-hashed hashtable by defining an **mhash** structure, and implementing the following functions:

```
mhash *mhash_new();
unsigned short int mhash_hashvalue(char *name);
void mhash_add(mhash *h, monster *m);
void mhash_remove(mhash *h, monster *m);
monster *mhash_find(mhash *h, char *name);
```

Notes:

- You may assume an **unsigned short int** is 16 bits.
- Your hashing function should return the product of the ASCII value of the first and last character of the name.
- Your **mhash** table should be large enough to hold the entire range of the hashing function.
- For this question don't worry about disposal.

```
typedef struct mhash_struct {
    monster_linked_list entries[65025];
} mhash;

mhash *mhash_new(){
    mhash *h = malloc(sizeof(mhash));
    for(int i = 0; i < 65025; i++){
        h->entries[i] = NULL;
    }
}

unsigned short int mhash_hashvalue(char *name){
    int first = (int) name[0];
    int last = (int) name[strlen(name) - 1];

    return first * last;
}

void mhash_add(mhash *h, monster *m){
    int hashval = mhash_hashvalue(name);
    monster_linked_list *l = h->entries[hashval];

    if(l == NULL) l = mll_new();
    mll_add(l, m);
    h->entries[hashval] = l;
}

void mhash_remove(mhash *h, monster *m){
    int hashval = mhash_hashvalue(name);
    monster_linked_list *l = h->entries[hashval];

    if(l == NULL) return;
    else mll_remove(l, m);
}

monster *mhash_find(mhash *h, char *name){
    int hashvalue = mhash_hashvalue(name);
    monster_linked_list *l = h->entries[hashvalue];

    if(l == NULL) return NULL;
    else mll_find(l, name);
}
```

Reflection (5)

What are the worst-case big-$\mathcal{O}$ times of your add, remove and find functions? When are those times reflective of real-world performance, and when is real-world performance better?

In big-O time, the worst-case runtime to my add, remove, and find functions will be O(n), given that we're working with hashes storing lists of monsters. Though it's relatively quick to find which hash to explore, we still have to traverse our list in order to add, remove, and find a monster.

Trees (25 points)

Coding (16 points)

Using the following leftmost-child-right-sibling tree structure:

```
struct tree_node_struct {
   int payload;
   struct tree_node_struct *child;
   struct tree_node_struct *sibling;
   unsigned int nchildren;
};
```

And with new nodes created by the following function:

```
tree_node *new_tree_node(int p) {
   tree_node *t = malloc(sizeof(tree_node));
   t->payload = p;
   t->child = NULL;
   t->sibling = NULL;
   t->nchildren = 0;
   return t;
}
```

Write a function to add a child to a parent, but maintain the property that **the children of any given parent are sorted by payload value in decreasing order** (highest to lowest).

```
void new_sibling(tree_node *parent, tree_node *child){
    tree_node *previous = parent->child;
    tree_node *current = parent->child->sibling;

    int inserted = 0;

    while(current != NULL){

        if(previous->payload < child->payload && inserted == 0) {
            child->sibling = previous;
            if(previous = parent->child) parent->child = child;
            inserted = 1;

        } else if (current->payload < child->payload && inserted == 0){
            previous->sibling = child;
            child->sibling = current;
            inserted = 1;
        }

        previous = previous->sibling;
        current = current->sibling;
    }

    if(inserted == 0) previous->sibling = child;

}

void add_child(tree_node *parent, int childPayload){
    tree_node *child = new_tree_node(childPayload);
    parent->nchildren++;

    if(parent->child == NULL) {
        // given that the child is NULL we assume that sibling should also be NULL
        parent->child = child;
    } else {
        new_sibling(parent, child);
    }
}
```

Analysis (9 points)

1. In big-$\mathcal{O}$ terms, how long will your function take to run? O(n).
2. How could result (1) be improved? We can improve it by using a doubly-linked list for children.
3. What change to the tree data structure would (2) require? We'll need to keep track of both the leftmost and rightmost children.

Tries (25 points)

Consider a trie implemented using a dynamic-child-pointer-array structure:

```
struct trie_node_struct {
   char letter;
   int is_word;
   struct trie_node_struct *parent;
   struct trie_node_struct **children;
   unsigned int nchildren;
};

typedef struct trie_node_struct trie_node;
```

Write a function to delete a word from this trie, using the following prototype:

```
void delete_word(trie_node *trie, char *word);
```

Note the following:

- Make sure you deal with the case when the word is a prefix.
- You must gracefully fail (i.e., do nothing and return) when the word is not in the trie.

```
void deleting(trie_node *child){
    trie_node *parent = child->parent;
    if(child->is_word || child->nchildren > 0 || parent == NULL){
        return;
    } else {
        parent->nchildren;
        parent->children[letter - 'a'] = NULL;
        free(child);
        deleting(parent);
    }
}

void delete_word(trie_node *trie, char *word){
    int length = strlen(word);
    if(length == 0){
        if(trie->is_word){
            trie->is_word = 0;
            deleting(trie);
        } else {
            return;
        }
    } else {
        char c = word[0]; int lc;
        if(isalpha(c)) lc = tolower(c) - 'a';
        else lc = -1

        if(lc == -1) delete_word(trie, word + 1);
        trie_node *next = trie->children[lc];
        if(next == NULL) return;
        delete_word(next, word + 1);
    }

}
```