Assignments week 5

Son Cao

2025-10-01

Assignment 1 - Removing values from a binary heap

My code:

```cpp
#include <iostream>
#include "utils.h"
#include <vector>

//min-heap

using namespace std;

void BubbleUp(vector<int>& heap, int i){
    while(i > 0){
        int parent = (i - 1) / 2;
        if(heap[i] < heap[parent]){
            swap(heap[i], heap[parent]);
            i = parent;
        } else break;
    }
}

void BubbleDown(vector<int>& heap, int i){
    int n = heap.size();
    while(true){
        int left = 2 * i + 1;
        int right = 2 * i + 2;
        int smallest = i;

        if(left < n && heap[left] < heap[smallest]) smallest = left;
        if(right < n && heap[right] < heap[smallest]) smallest = right;

        if(smallest != i){
```

```cpp
            swap(heap[i], heap[smallest]);
            i = smallest;
        } else break;
    }
}

int main() {
    /* TODO:
        Write a program that reads a binary *min-heap* from its standard input, given as an
        and an integer that represents a value to be removed from the heap.

        The program must remove the specified value from the heap while maintaining the heap
        property, and print the resulting heap in the same format as the input. If the value
        is not present in the heap, the program must print the original heap unchanged.

        Example input:
            [3, 4, 10, 23, 50, 90] 50
        Example output:
            [3, 4, 10, 23, 90]
    */
    vector<int> heap;
    int value;
    cin >> heap >> value;

    int idx = -1;
    for(int i = 0; i < (int)heap.size(); i++){          //O(n)
        if(heap[i] == value){   //find index of value
            idx = i;
            break;
        }
    }

    if(idx != -1){
        //replace with last element
        heap[idx] = heap.back();
        heap.pop_back();   //O(1)

        if(!heap.empty()){
            if(idx > 0 && heap[idx] < heap[(idx - 1) / 2]){
                BubbleUp(heap, idx);             //O(log n)
            } else {
                BubbleDown(heap, idx);
            }
        }
    }
    cout << heap << endl;
```

```
    return 0;
}
```

Even though restoring the heap takes only O(log n), the dominant step is the linear search to find the element => that's why the overall complexity is O(n).

Time complexity: this algorithm has a time complexity of O(n), because the algorithm must linearly search for the value to remove (O(n)), and then restore the heap property, which takes O(log n). The linear search dominates the total time.

Assignment 2 - Heapify

My code:

```
#include <iostream>
#include "utils.h"   // for reading vectors
#include <vector>

using namespace std;

int BubbleDown(vector<int>& heap, int i){
    int swaps = 0;
    int n = heap.size();
    while (true){
        int left = 2 * i + 1;
        int right = 2 * i + 2;
        int smallest = i;

        if(left < n && heap[left] < heap[smallest]) smallest = left;
        if(right < n && heap[right] < heap[smallest]) smallest = right;

        if(smallest != i){
            swap(heap[i], heap[smallest]);
            swaps++;
            i = smallest;
        } else break;
    }
    return swaps;
}

int heapify(vector<int>& arr){
    int swaps = 0;
    for(int i = arr.size()/2 - 1; i >= 0; --i){ //the last non-leaf node and goes up to the
        swaps += BubbleDown(arr, i);
    }
```

```
    return swaps;
}

//make_heap(arr.begin(), arr.end(), greater<int>()); doesnt return swaps

int main() {
    /* TODO:
        Write a program that reads an array from its standard input, given as an array betwe
        square brackets (e.g. `[1,3,5,7,9]` - see week 1), and prints the number of *swaps*
        would be performed when turning this array into a valid binary *min-heap* using
        the `std::make_heap` function from the C++ Standard Library.

        You can't use the `std::make_heap` function directly, instead you must implement the
        heapify algorithm as described in the lecture, and count the number of swaps perform
        during the process.

        The time complexity of your solution must be O(n), where n is the number of elements
        in the input array.

        Example input:
            [1, 5, 6, 2, 3, 4, 7]
        output:
            2
    */
    vector<int> arr;
    cin >> arr;

    int swaps = heapify(arr);
    cout << swaps << endl;
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n), because it builds the heap from the bottom up. Most elements are near the bottom of the tree and need only a few swaps to fix their position, so the total amount of work adds up to a value that grows roughly in proportion to the number of elements.

Assignment 3 - Running median

My code:

```
#include <iostream>
#include <queue>
#include <iomanip>

using namespace std;
```

```
int main() {
    /* TODO:
        Write a program that reads a sequence of numbers from its standard input and, after
        prints the median of all numbers read so far.

        You should use two binary heaps to implement the required functionality: a max-heap
        the lower half of the numbers, and a min-heap to store the upper half of the numbers

        The time complexity of your program must be O(n log n), where n is the total number
    */
    priority_queue<int> lowers;  //prioritty_queue<int> is a max-heap by default => keep the
    priority_queue<int, vector<int>, greater<int>> uppers; //min-heap => keep the upper hal

    double median;
    int x;

    while(cin >> x){
        if(lowers.empty() || x <= lowers.top()){
            lowers.push(x);
        } else {
            uppers.push(x);
        }

        if(lowers.size() > uppers.size() + 1){
            uppers.push(lowers.top());
            lowers.pop();
        } else if (uppers.size() > lowers.size() + 1){
            lowers.push(uppers.top());
            uppers.pop();
        }

        if(lowers.size() == uppers.size()){
            median = (lowers.top() + uppers.top()) / 2.0;
        } else if (lowers.size() > uppers.size()){
            median = lowers.top();
        } else {
            median = uppers.top();
        }
        cout << median << " ";  //cannot do cout << median << endl; cuz it prints ex: 1\n2\
    }
    cout << endl;
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n log n), because for each of the n numbers, insertion and balancing operations in a heap take O(log n) time (because inserting into a heap requires rearranging elements to keep the heap order). The constant-time median calculation doesn't affect the overall complexity.

Assignment 4 - Ordering coordinates

My code:

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <algorithm>

using namespace std;

struct Coord{
    int x, y;
    bool operator==(const Coord& other) const{
        return x == other.x && y == other.y;
    }
};

struct CoordHash{     //converts a key (like a coordinate) into an integer number that can b
    size_t operator()(const Coord& c) const{           //^ (XOR) -> mix the bits together
        return hash<int>()(c.x) ^ (hash<int>()(c.y) << 1);  //<< 1 -> shift one of them left
    }
};

int main() {
    /* TODO:
        Write a program that inputs a string consisting of the characters "U", "D", "L", and
        representing movements Up, Down, Left, and Right on a 2D grid, starting from the ori

        The program must then compute and print the top three of *the most visited unique co
        that are visited during the sequence of movements, including the starting point (0,

        Ties should be broken by the order in which the coordinates were first visited.

        For example, given the input string "UUDLURD", the program must print `(0, 1), (0, 2
        because the visited coordinates are (0, 0), (0, 1), (0, 2), (0, 1), (-1, 1), (-1, 2)

        The time complexity of your solution must be O(n + 3 log(n)), which can be simplifie
        where n is the length of the input string.
```

```
    To efficiently store and check for unique coordinates, you should use a hash table
    `std::unordered_set` or `std::unordered_map` in C++). Note that you will have to de
    function for the coordinate pairs, as well as an equality operator. Also, you must 
    into a vector or similar structure to be able to sort them by their visit counts.

    As a final note, you are not allowed to use the nth_element algorithm from the stan
*/
string moves;
cin >> moves;
unordered_map<Coord, pair<int, int>, CoordHash> visits;
//Coord => visits[{x, y}]
//pair<int, int> => ex: {1, step}
//first (1) = number of times this coordinate was visited
//second (step) = step index when this coordinate was first visited (used for tie-break

int x = 0, y = 0;
int step = 0;
visits[{x, y}] = {1, step++};  //ex: visits[{0,1}] = {3, 1}; => Coordinate (0,1) was vi

//the moves
for(char move : moves){
    if(move == 'U') y++;
    else if(move == 'D') y--;
    else if (move == 'L') x--;
    else if(move == 'R') x++;

    Coord current = {x, y};  //current will be used as a key in the unordered_map visit
    if(visits.find(current) == visits.end()){  //visits.find(current) looks for current
                                                //visits.end() represents "not found"
        visits[current] = {1, step++};
    } else {
        visits[current].first++;  //We do not change the first-visit step (second), bec
     }
}

//sort to get top 3                //unordered_map does not store items in order, so we
vector<pair<Coord, pair<int,int>>> all(visits.begin(), visits.end());  //copy all entri
// pair<Coord, pair<int,int>>
// all[i].first => the coordinate
// all[i].second.first => number of visits
// all[i].second.second => first-visit step

sort(all.begin(), all.end(), [](auto &a, auto &b){
    if((a.second.first != b.second.first))
        return a.second.first > b.second.first; //true if a goes b4 b ,false if a goes 
    return a.second.second < b.second.second;   //if visit counts differ => bigger coun
```

```
    });                                                 // if counts are equal => smaller firs

    int count = 0;
    for(auto &p : all){   //p: reference to each element
        cout << "(" << p.first.x << ", " << p.first.y << ")";
        count++;            //track how many coordinates we printed
        if(count == 3 || count == (int)all.size()) break;
        cout << ", ";
    }
    cout << endl;
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n + log n), because it processes each move once to update coordinates and counts using an unordered_map, which gives O(1) average time per operation, so this part is O(n). And it then sorts the coordinates to find the top 3 most visited ones. Sorting normally costs O(k log k) (where k is the number of unique coordinates), but since we only need the top 3, the cost simplifies to O(3 log n) ~= O(log n).

Assignment 5 - Linear probing

My code:

```
#include <iostream>
#include "utils.h"
#include <vector>
#include <string>
#include <bits/hash_bytes.h>

using namespace std;

struct string_hash{
    size_t operator()(const string& str) const{
        return std::_Hash_bytes(str.data(), str.size(), static_cast<size_t>(0xc70f6907UL));
    }
};

int main() {
    /* TODO:
        Write a program that inputs an integer followed by a list of strings (given as a co
        between square brackets, e.g. [apple, banana, cherry], use the utils.h header to re

        The list represents the strings to be inserted into a hash table that uses linear p
        resolution, and the integer represents the number of slots in the hash table.
```

```
    The program must insert the strings into the hash table in the order they are given,
    resulting hash table as a comma-separated list between square brackets, where empty
    represented by the string "EMPTY".

    The hash table must implement a *set*, meaning that duplicate strings should not be
    times. In other words, if the same string appears more than once in the input list,
    appear once in the output hash table.

    You must use the std::hash<std::string> struct from the C++ Standard Library to comp
    of a string. When determining whether a string is already present in the hash table,
    the entire table. Instead, you should use the same probing sequence that would be us
    stopping when you either find the string or reach an empty slot.

    Example input:
        ((2 3 4) 5 (6 7 8)) (3 5 10)
    Example output:
        [3, 5]
*/

// Hint: write a function to find an empty slot using linear probing
int table_size;
char c;
cin >> table_size >> c;
vector<string> table(table_size, "EMPTY");
string word;

string_hash hasher;

while(cin >> word){
    if(word.back() == ',') word.pop_back();  //check for "[" first
    if(word.back() == ']') word.pop_back();

    size_t index = hasher(word) % table_size;  //tryna find index in a table ( size_t i

    for(int i = 0; i < table_size; i++){
        size_t probe = (index + i) % table_size;  //wrap around the table check for emp

        if(table[probe] == "EMPTY"){
            table[probe] = word;
            break;
        } else if (table[probe] == word){
            //if duplicate => ignore
            break;
        }
    }
    if(word.back() == ']') break;
```

```
    }
    cout << "[";
    for(int i = 0; i < table_size; i++){
        cout << table[i];
        if(i != table_size - 1) cout << ", ";
    }
    cout << "]" << endl;
    return 0;
}
```

For each string in the input, the program first calculates its hash, which takes about O(k) time where k is the length of the string. Then it tries to insert the string into the hash table using linear probing. In the worst case, if the table is nearly full or there are lots of collisions, it might have to check every slot in the table, which is O(m) where m is the table size.

Time complexity: this algorithm has a time complexity of $O(n * (m + k)) = O(n * m + n * k)$

Assignment 6 - Separate chaining, rehashed

My code:

```
#include <iostream>
#include "utils.h"
#include <vector>
#include <forward_list>
#include <bits/hash_bytes.h>

using namespace std;

struct string_hash{
    size_t operator()(const string& str) const {
        return std::_Hash_bytes(str.data(), str.size(), static_cast<size_t>(0xc70f6907UL));
    }
};

//insert with separate chaining(prepending)
void insert_string(vector<forward_list<string>>& table, const string& word, string_hash& has
    size_t index = hasher(word) % table.size();

    //check for duplicate
    for(auto& s : table[index]){
        if(s == word) return;
    }
    table[index].push_front(word); //prepending
```

```
}

int main()
{
    /* TODO:
        Write a program that reads a integer followed by a list of strings from its standard
        comma-separated list between square brackets, e.g. `[apple,banana,cherry]` - see wee

        The list represents the strings to be inserted into a hash table that uses *separate
        collision resolution, and the integer represents the number of slots in the hash tab

        The program must insert the strings into the hash table in the order they are given,
        resulting hash table as a comma-separated list between square brackets, where each s
        as a linked list in the form `[<string1> -> <string2> -> ...]` (empty slots will be
        a pair of brackets, i.e. "[]").

        The hash table must implement a *set*, meaning that duplicate strings should not be
        times. In other words, if the same string appears more than once in the input list,
        appear once in the output hash table.

        To determine the index for a string, you *must* use the `std::hash<std::string>` str
        Standard Library to compute the hash value of the string.

        When determining whether a string is already present in the hash table, you must not
        table. Instead, you should use the same probing sequence that would be used for inse
        when you either find the string or reach the end of the linked list.
    */

    // Hint: use a vector of singly linked lists as the underlying data structure for the ha
    int k, n;
    vector<string> words;
    cin >> k >> words >> n;

    string_hash hasher;
    //build initial table
    vector<forward_list<string>> table(k);   //using forward_list because it's -> (forward_
    for(auto& word : words){
        insert_string(table, word, hasher);
    }

    //rehash into new table
    vector<forward_list<string>> new_table(n);
    for(auto& bucket : table){
        for(auto& word : bucket){
            insert_string(new_table, word, hasher);
        }
```

```
    }
    //print the final rehashed table
    cout << new_table << endl;
    return 0;
}
```