Interesting Data Structures

Interesting data structures explained concisely

By

Fco Borja Lozano

Universidad Complutense de Madrid

September 22, 2019

Contents

1 Imtroduction

1.1 A programmer should not be close-minded

Programmers use only the data structures they know about. There are many data structures out there that most people know nothing about, and not necessarily because they have been surpassed by other data structures, but because they are only useful in at solving very specific problems.

The objective of this paper is explaining some of these data structures and brighten some curiosity on the reader so that he or she will continue to learn about more data structures in the future.

1.2 Problems have multiple solutions

Storing, accessing, and finding strings is, arguably, one of the main problems that have to be dealt with on a constant basis. First, we will explore some data structures that do just that, starting with the deterministic finite state automaton (DFSA) and ending with the gap buffer. This shows that something as simple as storing a word can be done in very different ways.

1.3 Solutions have multiple implementations

A binary search tree is one of the most basic data structures, but few people know about the skip lists, their long lost brother. They offer a clear example of how we can implement a concept differently and modify its properties.

1.4 Implementations can be improved

A data structure can be improved without changing it internally. A bloom filter, for example, that decreases the necessary searches on a data structure in a very simple and easy way. A data structure can also be added new functionalities. This is the case for the generic zipper data structure, which improves the efficiency when exploring and doing local modifications.

2 Deterministic Finite State Automaton

2.1 Introduction

Lets start defining a state automaton. A state automaton is a model able to change from one state to another in response to external inputs. It is normaly denoted as a tuple $M = (Q, \Sigma, q_0, \delta, F)$ where:

- Q : the set of states
- Σ : the alphabet.
- q_0 : initial state.
- δ the transition function $Q \times \Sigma \rightarrow Q$.
- F is the subset of Q that are final states.

As you can imagine, in a finite state automaton, F is finite. And deterministic implies that Σ is injective.

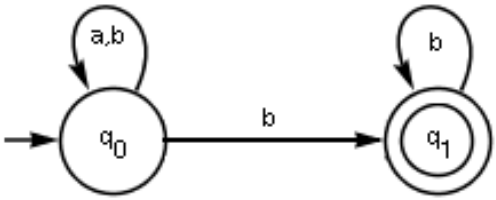

Figure 1: Example of a **not** deterministic automaton.

2.2 As a Data Structure

A string is said to be accepted by the DFSA if there is a sequence of transitions from the initial state passing through the characters of the string in order. For example, let $w = a_1a_2a_3a_4\ldots a_n$ be a string where $a_i \in \Sigma \quad i = 1,\ldots,n$. The automaton M is said to accept the string w if $\delta(\ldots\delta(\delta(q_0,a_1),a_2)\ldots,a_n) \in F$. This way, there isn't a node which contains the string; the strings stored are represented by the set of transitions from q_0 to any $q \in F$.

2.2.1 Trie

A trie is the most basic implementation of a DFSA as a data structure. It is a tree where the root is the initial state and the leaves are the final states.

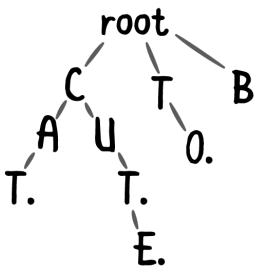

Figure 2: Example of a trie.

2.2.2 Radix Tree

The problem with a trie is that when there is a node for each character, it can have negative effects on the efficiency of memory use. A radix tree is a compressed version of a trie, we can use more than one character per transition decreasing the required node quantity.

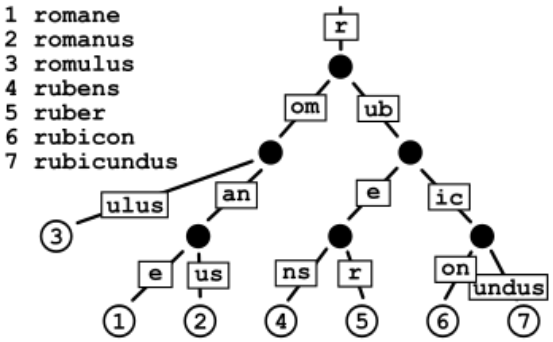

Figure 3: Example of a radix tree.

2.2.3 Deterministic Acyclic Finite State Automaton

There is also another way to decrease the number of nodes required, and that is to join the nodes that are the destination of two different transitions when possible.

2.2.4 Comparison

Even tho both DAFSA and radix tree are optimized versions of a trie, these two are much harder to implement. And due to the great efficiency of the trie, it is many times deemed unnecessary to use a DAFSA or a radix tree instead. Let's compare a trie to a hash table.

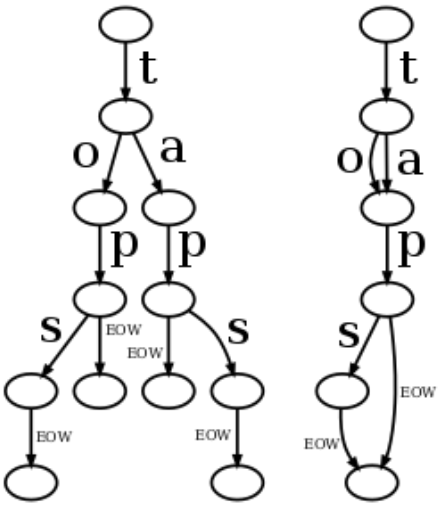

Figure 4: Comparison between a trie and a DAFSA

Due to the structure of a trie, deletion, insertion and look-up are all almost equal in cost ($O(k)$ where k stands for the length of the key). That is why the worst case scenarios of insertions and deletions of a hash table ($O(n)$ where n stands for the number of keys) is much worse. That said, a hash table is faster in most scenarios, and due to its popularity, there are much more well-optimized and clear implementations. One thing that a trie can easily do that a hash table can't is a search for keys with the same suffixes of prefixes, which can be very useful.

2.3 Code

```
#pragma once
#include <vector>

template<class T>
class trie {

    struct Node;

    struct Node {
        bool isEnd;
        T key;
        std::vector<Node*> children;
        int num_of_children;

        Node(T key, bool isEnd) : isEnd(isEnd), key(key), num_of_children(0) {
            children = std::vector<Node*>();
            children.resize(10);
        }
```

```
        void insert(T key, bool isEnd) {
            if (children.size() == num_of_children)
                children.resize(children.size() * 2);
            children[num_of_children] = new Node(key, isEnd);
            num_of_children++;
        }
    };

    Node* _root;

    Node* findLastCommonAncestor(const T* string, int size, int& pos) const {
        Node* parent = _root;
        bool found_letter = true;

        for (int i = 0; i < size && found_letter; i++) {
            bool found_letter = false;

            for (int s = 0; s < parent->num_of_children; s++) {
                if (parent->children[s]->key == string[i]) {
                    found_letter = true;
                    parent = parent->children[s];
                    pos = i;
                    break;
                }
            }
        }

        return parent;
    }

public:
    Trie() : _root(new Node(T(),false)) {}
    Trie(Node* root) : _root(root) {}

    ~Trie() {
        clear(_root);
    }

    bool find(const T* string, int size) const {
        int i;
        Node* node = findLastCommonAncestor(string, size,i);
        return node != _root && node->isEnd && node->key == string[size-1];
    }

    void insert(const T* string, int size) {
        int pos;
        Node* node = findLastCommonAncestor(string, size, pos);

        int i = (node != _root) ? pos+1 : 0;

        for (; i < size; i++) {
            node->insert(string[i],i==size-1);
            node = node->children[node->num_of_children - 1];
```

```
        }
    }

    void clear(Node* node) {
        for (int i = 0; i < node->num_of_children; i++){
            clear(node->children[i]);
        }
        delete node;
    }

    Trie& getSubtrie(const T* string, int size) {
        int i;
        Node* node = findLastCommonAncestor(string, size, i);
        Trie* trie;

        if (_root == node)
            trie = new trie();
        else
            trie = new trie(node);

        return &*trie;
    }
};
```

3 Rope

3.1 Introduction

A rope is a way to store a string, different to the usual array of characters. We use a binary tree and store some characters in the leaves and store in each node the number of characters to the left. That way to divide a string or delete a part of it we just have to cut a part of the tree and to append or concatenate two strings we just have to join the trees as the children of a new root. There are different possible implementations of a rope. Normally we set a maximum size of characters per leave and try to keep them filled, so the amount of nodes doesn't end up being the number of characters of the string.

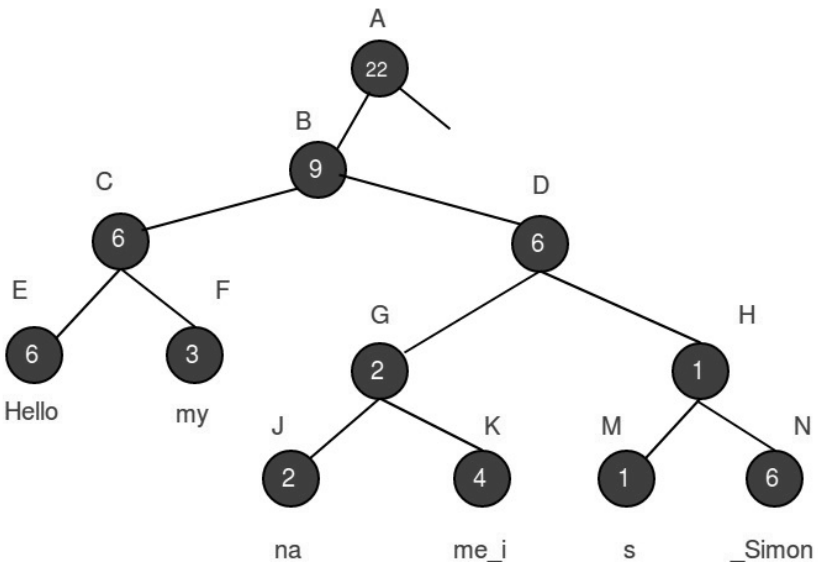

Figure 5: Example of a rope.

3.2 Ropes vs. strings

Let's compare its advantages and disadvantages over a traditional string.

Advantages:

- ropes drastically cut down the cost of appending two strings.
- Unlike arrays, ropes do not require large contiguous memory allocations.
- ropes do not require O(n) additional memory to perform operations like insertion / deletion / searching.

Disadvantages:

- The complexity of source code increases. Much more if we want fast iteration, that can be implemented $O(1)$.
- Extra memory required to store parent nodes.
- Time to access i-th character increases.

Performance		
Operation	rope	String
Index	$O(log(n))$	$O(1)$
Substring	$O(1)$	$O(1)$
Concatenate	$O(1)$	$O(n)$
Insert	$O(log(n))$	$O(n)$
Append	$O(1)$	$O(n)$
Delete	$O(log(n))$	$O(n)$
Build	$O(n)$	$O(n)$
Split	$O(log(n))$	$O(1)$
Iteration	$O(1)$	$O(1)$

Table 1: Rope comparison table of a well implemented rope.

3.3 Code

```
#pragma once
#include <algorithm>
#include <iostream>
#include <string>

using namespace std;

class rope {
    Rope *l, *r, *p;
    Rope *last;
    char *str;
    int count;
    int size;
public:
    static void createRope(Rope * &node, rope *par, char str[], int const& l,
        int const& r, int& size) {
        Rope* temp = new rope(size);
        temp->p = par;
        temp->size = size;

        if (r - l <= size) {
            char* str_temp = new char[size+1];
            for (int i = 0; i <= r - l; i++)
                str_temp[i] = str[l + i];
            temp->str = str_temp;
            temp->count = r - l + 1;
```

```
            temp->last = temp;
        }
        else if (r - l <= 2 * size) {
            createRope(temp->l, temp, str, l, l+size-1,size);
            createRope(temp->r, temp, str, l+size, r, size);
            temp->count = size;
        }
        else {
            createRope(temp->l, temp, str, l, (r + l) / 2, size);
            createRope(temp->r, temp, str, (r + l) / 2+1, r, size);
            temp->count = temp->l->l->count + temp->l->r->count;
        }

        if(par)
            par->last = temp->last;

        node = temp;
    }

    Rope(int size) {
        l = r = p = nullptr;
        str = nullptr;
        count = 0;
        this->size = size;
    }

    void insert(char str[], int l, int r) {
        bool inserted = false;

        if ((last)->count < size) {
            int m = std::min(r - l, size - (last)->count);
            for (int i = 0; i < m; i++)
                (last)->str[i + (last)->count] = str[l + i];
            (last)->count += m;
            insert(str, l + m, r);
        }
        else {
            Rope* temp;
            createRope(temp, nullptr, str, l, r, this->size);
            append(temp);
        }

    }
    void append(Rope *rope) {
        Rope *temp = new rope(this->size);

        temp->l = this->l;
        temp->r = this->r;
        temp->count = this->count;
        temp->last = this->last;

        temp->p = this;
        rope->p = this;
```

```
        this->l = temp;
        this->r = rope;

        (this->last) = (r->last);
    }

    const char* build(int& length) const
    {
        if (str) {
            length = this->count;
            return str;
        }
        else if (r && l) {
            int ll, lr;
            const char* right = r->build(lr);
            const char* left = l->build(ll);
            char* str = new char[ll+lr];
            for (int i = 0; i < ll; i++)
                str[i] = left[i];
            for (int i = 0; i < lr; i++)
                str[ll+i] = right[i];
            length = ll + lr;
            return  str;
        }
    }

    Rope* concat(Rope* rope) {
        Rope *temp = new rope(rope->size);

        temp->l = this;
        temp->r = rope;
        temp->last = this->last;
        temp->count = this->count + this->r->count;

        this->p = temp;
        rope->p = temp;

        ((this->last))->count -= 1;
        (this->last) = (r->last);

        return temp;
    }

    char at(int i) {
        if (i <= count)
            if (str)
                return str[i];
            else
                return l->at(i);
        else
            return r->at(i - count);
    }
```

```
    void clear() {
        if (l) {
            l->clear();
            delete l;
        }
        if(r){
            r->clear();
            delete r;
        }
        if(str) {
            delete[] str;
        }
    }
};
```

4 Gap Buffer

4.1 Introduction

A gap buffer is the most common data structure used to store and manage a string. As opposed to a list, it keeps the gap which can be filled in the middle of the contents.

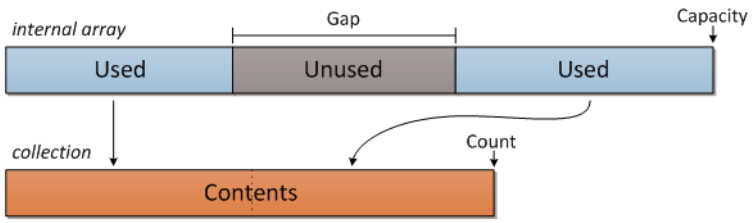

Figure 6: Structure of a gap buffer.

This structure can be implemented with a dynamic array which preserves a gap in the middle and remembers the adjacent positions to the gap. It's got two segments of text - one at the beginning of the buffer, which grows upwards; and one at the end of the buffer, which grows downwards.

In a buffer, we maintain the focus (cursor) on the last character before the gap. That way, to insert a substring we just fill the gap, and if necessary, expand the gap. If we want to move the focus back, we reallocate the character after the gap, and if we want to move it forward, we allocate the first character after the gap before the gap.

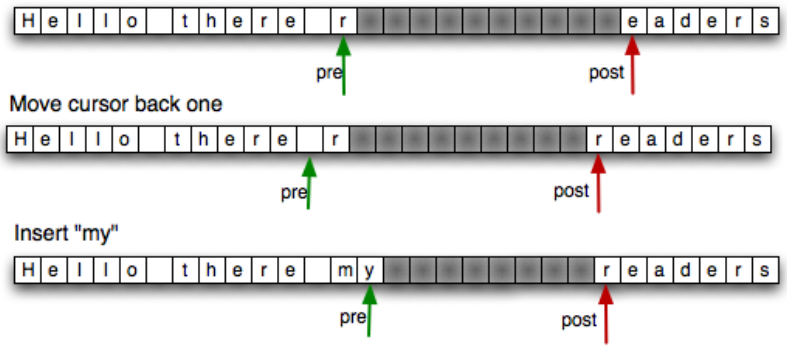

Figure 7: Example of a gap buffer.

4.2 Comparison to ropes

A gap buffer can move the cursor, remove or insert characters in $O(1)$. When we need to iterate over the string doing small edits, a gap buffer is much more efficient than a rope. However, operations at different locations in the text and ones that fill the gap (requiring a new gap to be created) may require copying most of the text. And of course, a gap buffer requires much more memory just to maintain the gap.

4.3 Code

```
#pragma once

#include <string>
#include <iostream>

class GapBuffer {
    char* c;
    int beforeGap, afterGap;
    int capacity;

    void resize(int size) {
        char* a = new char[size];
        for (int i = 0; i < beforeGap; i++)
            a[i] = c[i];

        int afterSize = capacity - afterGap;
        for (int i = 1; i <= afterSize; i++)
            a[size - i] = c[capacity - i];

        delete[] c;
        c = a;

        capacity = size;
        afterGap = capacity - afterSize;
    }

public:
    GapBuffer(int capacity) : capacity(capacity) {
        c = new char[capacity];
        beforeGap = -1;
        afterGap = capacity;

    }

    char& at(int i) {
        if (i <= beforeGap)
            return c[i];
        else
            return c[i-beforeGap+afterGap];
    }
```

```
void moveCursor(int i) {
    if (i < beforeGap) {
        int size = beforeGap - i;
        for (int aux = 0; aux < size; aux++)
            moveCursorLeft();
    }
    else if (i > beforeGap) {
        i = i - beforeGap;
        for (int aux = 0; aux < i; aux++)
            moveCursorRight();
    }
    beforeGap = i;
}

void insert(std::string const& w) {
    if (w.size() >= afterGap - beforeGap)
        resize(capacity * 2);

    for (int i = 1; i <= w.size(); i++)
        c[beforeGap + i] = w[i - 1];

    beforeGap += w.size();
}

void remove(const int& start,const int& end) {
    std::string w(c);

    moveCursor(end);
    beforeGap = start;

    if (capacity * 1 / 4 <= beforeGap + afterGap - capacity)
        resize(capacity / 2);
}

char getCursor() { return c[beforeGap]; }

int getSize() { return beforeGap + capacity - afterGap; }

void moveCursorLeft() {
    if (beforeGap == 0)
        return;

    c[afterGap-1] = c[beforeGap-1];
    afterGap--;
    beforeGap--;
}

void moveCursorRight() {
    if (afterGap == capacity)
        return;
    c[beforeGap + 1] = c[afterGap + 1];
    afterGap++;
```

```
            beforeGap++;
        }
};
```

Another example of a gap buffer implementation can be seen in Git Hub by Hsin Tsao (Lazy Hacker).

5 Skip List

5.1 Introduction

A skip list is a variation of an ordered linked list. The idea behind a skip list is pretty simple, if we can skip elements that are smaller than our key doing a look-up, we can greatly decrease the cost of the operations. The structure of a skip list consists of a series of linked lists, where we have the main layer, containing all the elements in the list, and the rest have fewer elements and serve as "fast lanes" and each layer has approximately half the nodes of the previous one.

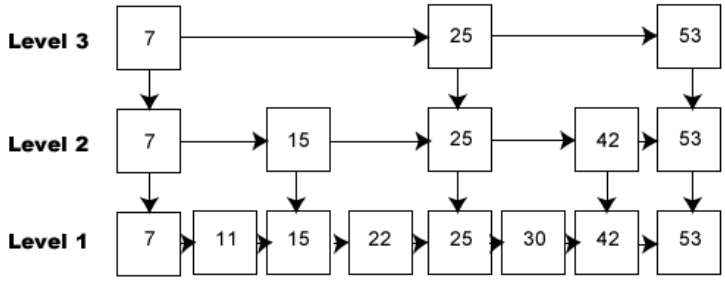

Figure 8: Example of a skip list.

Lets have a look at how insertion, deletion and look-up are done. As we can guess by looking at the image, a look-up would follow the next steps:

Algorithm 1 Look-up

```
1:  procedure FIND(element)
2:      node ← head
3:      while node.below ≠ null do
4:          if node.next ≠ null ∧ node.next.elem < element then
5:              node ← node.next
6:          else
7:              node ← node.below
8:      while node.next.elem < elem do
9:          node ← node.next
10:     return node.elem = element
```

In an insertion we first look-up where the element should be, and then, starting from the last layer, we insert the element and toss a coin. If the coin toss is a success, we move up a layer and repeat. That way we approximately have half the elements in each layer. The difference between the initial look-up of the insertion and the pseudo-code of the look-up

operation is that we have to remember the nodes we have used to go deeper in each layer, because if we need to insert the new node there, we will have to put the node between the one we used to go deeper and the next one.

The deletion is the same as the pseudo-code, but if the element of the next element is equal to the element to remove, we delete the node before going down.

5.2 Comparison to alternatives

The operations for insertion, deletion and look-up have the same cost ($O(log(n))$) as a BST. But while a BST can't iterate easily or look-up by index, the skip list can iterate easily and can have logarithmic look-up by index if each node stores the distance in nodes in the main layer between an element and the next. But it is not perfect. As we can see, the number of nodes required is much bigger, and thus, the memory use is greater.

5.3 Code

```
#pragma once

#include <stdlib.h>
#include <time.h>
#include <vector>
#include <iterator>
#include <memory>
#include <iostream>

template<class T>
class SkipList {
    struct Node;
    using Link = Node*;
    struct Node {
        Link next, below;
        T elem;
        int lvl;

        Node(T elem, Link next, Link below, int lvl) : next(next), below(below),
            elem(elem), lvl(lvl) {}
        Node() : next(nullptr), below(nullptr), elem(T()), lvl(0) {}
    };

    Link head, NIL;

    bool coinToss() {
        return std::rand() % 2 == 0;
    }

    std::vector<Link>* getInfima(T elem) {
        std::vector<Link>* infima = new std::vector<Link>();
        Link current = head;
```

```
        // While we are in the skipping lanes
        while (current->below != NIL) {
            // While we can skip
            while (current->next != NIL && current->next->elem < elem) {
                current = current->next;
            }

            // If we can go down
            if (current->below != NIL) {
                //If we can't skip
                if (current->next == NIL || current->next->elem >= elem) {
                    infima->push_back(current);
                    current = current->below;
                }
            }
        }

        while (current->next != NIL && current->next->elem < elem) {
            current = current->next;
        }

        infima->push_back(current);

        return infima;
    }

public:

    class Iterator {
        Link link;

    public:
        Iterator(Iterator const& iterator) {
            link = iterator.link;
        }
        Iterator(Link const& link) { this->link = link; }

        bool operator==(Iterator& it) {
            return it.link == link;
        }
        Iterator& operator=(Iterator const& iterator) {
            return link = iterator.link;
        }
        Iterator& operator++() {
            link = link->next;
            return *this;
        }
        T& operator*() const {
            return link->elem;
        }
    };
```

```
SkipList() {
    head = new Node();
    NIL = new Node();

    head->next = NIL;
    head->below = NIL;
}

~SkipList() { clear(); delete head; delete NIL; }

Iterator& begin() {
    Link current = head;

    while (current->below != NIL)
        current = current->below;

    Iterator* iter = new Iterator(current->next);
    return *iter;
}
Iterator& end() {
    Iterator* iter = new Iterator(NIL);
    return *iter;
}

bool search(T elem) {
    std::vector<Link>* sup = getinfima(elem);
    return (sup->at(sup->size()-1)->elem == elem);
}

void insert(T elem) {
    std::vector<Link>* sup = getinfima(elem);
    Link below = NIL;

    bool isHeads = true;
    int i = sup->size()-1;

    Link node = NIL;
    while (isHeads && i >= 0) {
        node = new Node(elem, sup->at(i)->next, below, sup->at(i)->lvl);
        sup->at(i)->next = node;
        below = node;

        isHeads = coinToss();
        i--;
    }

    if (isHeads) {
        Link newHead = new Node();

        newHead->next = new Node(elem, NIL, node, head->lvl + 1);
        newHead->below = head;
        newHead->lvl = head->lvl + 1;
```

```
            head = newHead;
        }
        delete sup;
    }
    void erase(T elem) {
        std::vector<Link>* sups = getinfima(elem);

        for(int i = 0; i < sups->size(); i++){
            if (sups->at(i)->next->elem == elem) {
                Link aux = sups->at(i)->next->next;
                delete sups->at(i)->next;
                sups->at(i)->next = aux;
            }
        }

        delete sups;
    }

    void print() {
        Link lane_head = head;

        while (lane_head != NIL) {
            Link current = lane_head;
            std::cout << "HEAD->";
            while (current->next != NIL) {
                std::cout << current->next->elem << "->";
                current = current->next;
            }
            std::cout << "NIL\n";
            lane_head = lane_head->below;
        }
    }

    void clear() {
        Link lane_head = head;
        while (lane_head != NIL) {
            Link current = lane_head;
            Link next_lanehead = lane_head->below;

            while (current->next != NIL) {
                Link next_current = current->next;
                delete current;
                current = next_current;
            }

            lane_head = next_lanehead;
        }
        head = new Node();

        head->next = NIL;
        head->below = NIL;
    }
};
```

6 Bloom filter

6.1 Introduction

A bloom filter is a probabilistic data structure that is able to tell us whether an element is **definitely** not part of a set or may be part of the set.

In the implementation, the bloom filter is usually a hash table and its hash function $h : \Sigma \longrightarrow \mathbb{N}$, where Σ stands for the data that can be stored.

When we add an element w to the set, we set the position $h(w)$ of the table as filled.

To check whether the element w is or isn't in the set we basically check whether the position $h(w)$ of the bloom filter is filled. If it is filled there might be an element with the same hash, so it is not definite that the element is in the set, but if it isn't filled, the element is definitely not in the set.

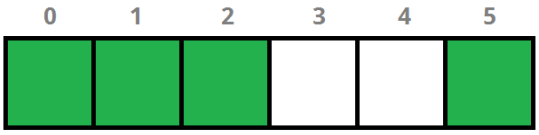

Figure 9: Example of a bloom filter's table.

6.2 Why use the bloom filter and what are the downsides

A bloom filter is a way to make sure we are not making unnecessary searches when a search is very expensive.

A problem with it is that for removing an element from the set we have to do complex processes to decide whether the bloom filter should be modified. This is due to not knowing if there exists another element in the set with the same hash.

6.3 Code

A bloom filter could be implemented as a wrapper of the set's class. This way the algorithms for look-up and insertion would be as follows:

Algorithm 2 Insertion

```
1: procedure INSERT(element)
2:     hash ← Hash(element)
3:     table.setFilled(hash)
4:     set.insert(element)
```

Algorithm 3 Look-up

```
1: procedure FIND(element)
2:     hash ← Hash(element)
3:     if table.isFilledAt(hash) then
4:         elementFound ← set.find(element)
5:     else
6:         elementFound ← false
7:     return elementFound
```

A possible implementation of this would be:

```
#pragma once
#include <vector>
#include <functional>

template<class T>
class Set {
    virtual void insert(T const& e) = 0;
    virtual bool find(T const& e) = 0;
    virtual void remove(T const& e) = 0;
};

template<class T = int, typename HashFunction = std::hash<T>>
class BloomFilter {
private:
    int capacity, n;
    std::vector<bool> bits;
    HashFunction h;
    Set<T> set;

    void checkHashDeletion(T const& e) {
        // Some complex process
    }
public:
    BloomFilter(int capacity, Set const& set) : capacity(capacity), n(0),
        bits(std::vector<bool>(capacity,false)), h(HashFunction()), set(set) {}

    void insert(T const& e) {
        int hash = h(e);
        bits[hash%capacity] = 1;
        set.insert(e);
    }

    bool find(T const& e) {
        if (bits[h(e) % capacity])
            return set.find(e);
        else
            return false;
    }
```

```
    void remove(T const& e) {
        if (bits[h(e) % capacity]) {
            set.remove(e);
            checkHashDeletion(e);
        }
    }
};
```

7 Zippers

7.1 Introduction

As described by Wikipedia:

> "A zipper is a technique of representing an aggregate data structure so that it is convenient for writing programs that traverse the structure arbitrarily and update its contents, especially in purely functional programming languages."

The theory of zippers is also applied to trees making quick exploration of trees. See Git Hub.

In fact, a complex data structure can be "zippered". This is known as the Generic zipper technique. This is explained in detail by Chung-chieh Shan and Oleg Kiselyov in "From walking to zipping".

7.2 How does a zipper look like?

Lets have a zipped array as an example. Let's say we are in the i-th element (our focus is on the i-th element). A zipper keeps the first $i-1$ elements inverted. To move one step backwards.

For example, the array [1 2 3 4 5 6] focused on the 3 would be represented as:

[2 1] 3 [4 5 6]

Now, you can easily change the 3 to something else. You can also easily move the focus left or right.

[1] 2 [3 4 5 6] [3 2 1] 4 [5 6]

A zippered tree is the same idea. You represent the tree as the focused element plus the context (up to parents, down to children) which gives you the whole tree in a form where it's easily modifiable at the focus and it's easy to move the focus up and down.

7.3 To zipper, or not to zipper

Zipping a data structure can be quite a challenge in some cases, but it might be worth the effort. It depends on how you will access the data structure. Obviously, if you need to go one by one doing modifications on some of them, a zipper would excel at it. But if you need to continuously search for elements, this structure might not be the wisest choice.

7.4 Code

```
#include <stack>
#include <queue>

template<class T>
class ListZipper {
    T current;
    std::stack<int> l,r;
    bool empty;
public:
    ListZipper() : empty(true) {}
    void insert(T e) {
        if (empty) {
            empty = false;
            current = e;
        }
        else {
            r.push(current);
            current = e;
        }
    }
    void moveLeft() {
        if (l.empty())
            throw std::domain_error("");

        r.push(current);
        current = l.top();
        l.pop();
    }
    void moveRight() {
        if (r.empty())
            throw std::domain_error("");

        l.push(current);
        current = r.top();
        r.pop();
    }
    void erase() {
        if (r.empty()) {
            if (l.empty())
                empty = true;
            else {
                current = l.top();
                l.pop();
            }
        }
        else {
            current = r.top();
            r.pop();
        }
    }
```

```
    T& cursor() {
        if (empty)
            throw std::domain_error("");

        return current;
    }
};
```

References

[1] Trie | (Insert and Search)
https://www.geeksforgeeks.org/trie-insert-and-search/

[2] Top 10 Algorithms and Data Structures for Competitive Programming
https://www.geeksforgeeks.org/top-algorithms-and-data-structures-for-competitive-programming/

[3] Trie
https://en.wikipedia.org/wiki/Trie

[4] The Trie Data Structure (Prefix Tree)
https://medium.freecodecamp.org/trie-prefix-tree-algorithm-ee7ab3fe3413

[5] Introducing the rope data structure!
https://brianbondy.com/blog/90/introducing-the-rope-data-structure

[6] Ropes: Theory and practice
https://www.ibm.com/developerworks/library/j-ropes/index.html

[7] Ropes Data Structure (Fast String Concatenation)
https://www.geeksforgeeks.org/ropes-data-structure-fast-string-concatenation

[8] Ropes: Theory and practice
https://www.ibm.com/developerworks/library/j-ropes/index.html

[9] Rope(Data Structure)
https://en.wikipedia.org/wiki/Rope_%28data_structure%29

[10] Beap Implementation
https://github.com/pfalcon/beap

[11] Beap (Data Structure)
https://en.wikipedia.org/wiki/Beap

[12] What is a Zipper?
https://stackoverflow.com/questions/380438/what-is-the-zipper-data-structure-and-should-i-be-using-it

[13] Beap Implementation
https://ece.uwaterloo.ca/~dwharder/aads/Algorithms/Beaps

[14] What is a Zipper?
https://stackoverflow.com/questions/380438/what-is-the-zipper-data-structure-and-should-i-be-using-it

[15] From walking to zipping
http://conway.rutgers.edu/~ccshan/wiki/blog/posts/WalkZip1/

[16] Zipper (Data Structure)
https://en.wikipedia.org/wiki/Zipper_(data_structure)

[17] Gap Buffer
https://en.wikipedia.org/wiki/Gap_buffer

[18] Gap Buffer Implementation
https://github.com/lazyhacker/gapbuffer

[19] What is a Gap Buffer?
https://www.codeproject.com/articles/20910/generic-gap-buffer

[20] Gap Buffers, or, Don't Get Tied Up With Ropes?
http://www.goodmath.org/blog/2009/02/18/gap-buffers-or-dont-get-tied-up-with-ropes/