Notes on Data Structures and Algorithms - with implementation

Christian Popescu

November 2021

Contents

Part I

Introduction

Chapter 1

Bit Manipulation

[See Laaksonen 2017, - 2.3 Bit Manipulations]

1.1 Some interesting functions and examples

- set the k^{th} bit of x to 1.

  ```
  \x|(1<<k)
  ```

- set the k^{th} bit of x to 0.

  ```
  x&~(1<<k)
  ```

- inverts the k^{th} bit of x to 0.

  ```
  x^(1<<k)
  ```

  Still to do what is on page 22

The following bloc prints the bit representation of an integer:

```
for (int k = 31; k >= 0; k--) {
    if (x&(1<<k)) cout << "1"
    else cout << "0"
}
```

1.2 Quick sort

Part II

Data Structures

Chapter 2

Disjoint Sets

2.1 Implementing disjoint set

2.1.1 2 Way Merge

It's about merging two sorted list.

```
vector<int> merge2Ways(const vector<int>& lst1,
                       const vector<int> lst2){
    int n1(lst1.size());
    int n2(lst2.size());

    vector<int> result;

    int i1(0);
    int i2(0);

    while (i1 < n1 && i2 < n2) {
        if (lst1[i1] == lst2[i2] ) {
            result.push_back(lst1[i1]);
                ++i1; ++i2;
        } else if (lst1[i1] < lst2[i2] ) {
            result.push_back(lst1[i1]);
            ++i1;
        } else {
            result.push_back(lst2[i2]);
            ++i2;
        }

    }
    if (i1 < n1) {
        while (i1 < n1) {
            result.push_back(lst1[i1]);
            ++i1;
```

```
        }
    } else {
        while (i2 < n2) {
            result.push_back(lst2[i2]);
            ++i2;
        }
    }
    return move(result);
}
```

2.2 References

Chapter 3

Strings

3.1 Tries

3.1.1 Simple non compressed trie

It's about merging two sorted list.

```
class TrieNode {
    public:
        TrieNode* chld[26];
        bool isTerminal = false;
        int nw = 0;
        TrieNode() {
        for (int i=0; i<26; ++i) chld[i] = nullptr;
}
};

TrieNode* root = new TrieNode;

// add word to trie
void AddWordToList(const std::string& wordToAdd) {
    unsigned long n = wordToAdd.size() ;
    TrieNode* current = root;
    for (int i=0; i<n; ++i) {
        int ind = wordToAdd[i] - 'a';
        cout << ind << endl;
        if (current->chld[ind] == nullptr) {
            current->chld[ind] = new TrieNode();
        }
        current  = current->chld[ind];
        ++(current->nw);
    }
    current->isTerminal = true;
```

```cpp
}

// returns number of words having a given prefix
int getNbOfWords(const std::string& wordToAdd) {
    unsigned long n = wordToAdd.size() ;
    TrieNode* current = root;
    for (int i=0; i<n; ++i) {
        int ind = wordToAdd[i] - 'a';
        if (current->chld[ind] == nullptr) {
            return 0;
        }
        current  = current->chld[ind];
    }
return current->nw;
}
```

3.2 References

Chapter 4

Sorting

4.1 Merge sort

It is an example of **divide and conquer** strategy.

4.2 Quick sort

The quicksort algorithm has a worst-case running time of $\Theta(n^2)$ on an input array of n numbers.

Applies **divide and conquer** strategy.

4.3 Heap sort

Chapter 5

Trees

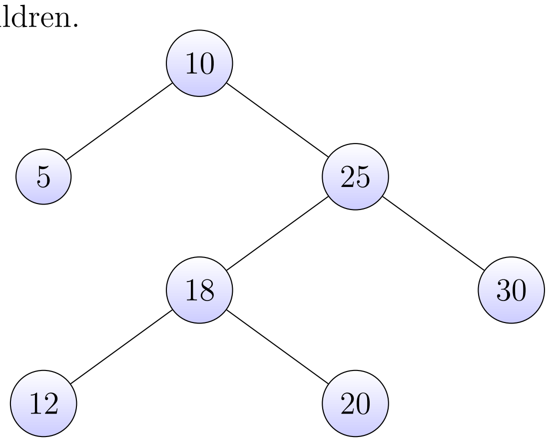
5.1 General Trees

A **tree** is an abstract data type that organize elements hierarchically.

A tree is **ordered** if there is a meaningful order among the children of each node.

5.2 Binary Trees

A **binary tree** is an ordered tree in which any node could have at most 2 children.



5.2.1 Implementation

```
class BinaryNode {
    public:
    BinaryNode* left;
```

```
    BinaryNode* right;
    BinaryNode* parent;
    int key;
};
```

5.2.2 Tree traversals

Algorithm 1 Inorder recursive traversal

procedure InorderRecursive($node$)
  if $node! = NULL$ **then**
    InorderRecursive($node.left$)
    Visit($node$)
    InorderRecursive($node.right$)

Algorithm 2 Inorder iterative traversal

procedure InorderIterative($node$)
  $current \leftarrow node$
  $stack \leftarrow emptystack$
  while NOT ($stack$ is empty) OR $current! = NULL$ **do**
    if $current! = NULL$ **then**
      stack.push($current$)
      $current \leftarrow current.left$
    else
      $current \leftarrow$ stack.pop()
      Visit($current$)
      $current \leftarrow current.right$

5.3 Binary Search Tree

As the number of children of one node is at most 2 we can keep direct links to them.

For some algorithms the navigation up in the tree is required so we can keep a link to the parent node.

5.4 Tries (Prefix trees)

A **trie** is a variant of a n-ary tree in which characters are stored at each node.

Goossens, Mittelbach, and Samarin 1993

Part III

Graph Algorithms

Chapter 6

Graph algorithms

6.1 Breadth-first search (BFS)

The algorithm use a **Queue** to manage the gray/visiting vertices.

6.2 Depth-first search(DFS)

The algorithm use a **Stack** to manage the vising vertices.

It could be implemented simple as recursive algorithm.

6.3 Topological Sorting

A **Topological Sorting** of a DAG is an linear ordering of its vertices such that for each vertice (u,v) in DAG the u appears before v in the ordering.

There are two known/classical algorithms for the Topological Sorting.

6.3.1 TS with DFS

Topological-Sorting(G) 1. Call DFS(G) to compute finishing time for each vertex. 2. When a vertex is finished inserted into a front of a linked list 3. Return the linked list of vertices.

6.3.2 TS taking the vertex without inbound edge first

1. Add all vertices without inbound edges to the ordered list 2. Remove from the graph these vertices's and the corresponding edges. 3. Repete steps 1 and 2 while there are vertices in the graph. If no vertices without inbount edges there is no solution.

Part IV

Algorithm Techniques

Chapter 7

Backtracking

7.1 Problem's specification

Pattern:

For a given problem we search all the sequences $\boldsymbol{x_1 x_2 ... x_n}$ for which some property holds $\boldsymbol{P_n(x_1, x_2, ..., x_n)}$

where: $\boldsymbol{x_k} \in \boldsymbol{D_k}$ (some given domain of integers)
The backtrack method consists in designing "cutoff"/"bounding" properties $\boldsymbol{P_l(x_1, x_2, ..., x_l)}$ for $\boldsymbol{1 \leq l < n}$ such that:

- $\boldsymbol{P_l(x_1, x_2, ..., x_l)}$ is true whenever $\boldsymbol{P_{l+1}(x_1, x_2, ..., x_{l+1})}$ is true;
- $\boldsymbol{P_l(x_1, x_2, ..., x_l)}$ is simple to test, if $\boldsymbol{P_{l-1}(x_1, x_2, ..., x_{l-1})}$ holds.

7.2 References

Bactracking from Knuth 2019

Supplementary Information

.1 Tables

.2 Figures

References

Bibliography

[GMS93] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. 3rd ed. Reading, Massachusetts: Addison-Wesley, 1993.

[Knu19] Donald E. Knuth. *The Art of Computer Programming - prefascicle 5B*. Addison-Wesley, 2019.

[Laa17] Antti Laaksonen. *Guide to Comptetitive Programming*. Springer, 2017.