

Search & Sort

Juho Jansson

Tietorakenteet ja algoritmit

Huhtikuu 2022

Tietotekniikka

Ohjelmisto- ja tietoverkkotekniikka

SISÄLLYS

[1 Ajankäyttö 3](#_Toc102253744)

[2 Käyttöohjeet 4](#_Toc102253745)

[3 Toiminta 5](#_Toc102253746)

[4 Ratkaisut 6](#_Toc102253747)

[4.1 Funktiot 10](#_Toc102253748)

[4.2 Search -funktiot 11](#_Toc102253749)

[4.2.1 Sequential search 11](#_Toc102253750)

[4.2.2 Binary search 11](#_Toc102253751)

[4.3 n2 sort -funktiot 11](#_Toc102253752)

[4.3.1 Insert sort 11](#_Toc102253753)

[4.3.2 Select sort 12](#_Toc102253754)

[4.4 n log n -sort 13](#_Toc102253755)

[4.4.1 Merge sort 13](#_Toc102253756)

[4.4.2 Tree sort 14](#_Toc102253757)

[4.5 Performance 17](#_Toc102253758)

[5 Ajo 18](#_Toc102253759)

# Ajankäyttö

10.4 – 2 h Ykkösvaiheen tekeminen, eli alkuhommien kasaaminen

18.4. – 6 h Kakkos ja kolmosvaiheiden, eli sort-algoritmien tekeminen

23.4 – 8 h viimeistely kakkos ja kolmosvaiheille. Yritystä saada double linked list templateluokka -toimimaan siten, että olisi ollut Record, jolla on Key -aliohjelma, jonka kautta olisi ajettu, mutta muuttui niin hanalaksi, että päätin tämän hylätä konsaan.

27.4. – 6 h nelosvaiheen kirjoittelemista ja debuggausta

28.4. – 8 h nelosvaiheen loppuunsaattaminen, 5 vaiheen tekeminen,

30.4. – 10 h – vitosvaiheen loppuunsaattaminen, tämän kirjoittaiminen illalla. Meni kauan vitosvaiheessa lähinnä, koska tree\_sort algoritmia en meinannut saada toimimaan kunnolla. Vitosvaiheen ns. performance testaus tulosteet ja ominaisuudet aika ala-arvoisia, koska jos niitä olisin alkanut säätämään olisi palautus myöhästynyt tämän raportin kirjoittamisen vuoksi. Myös jotkin vaaditut tulosteet eivät skulaa, kun tajusin ne liian myöhään tehtävänannosta. Algoritmit kuitenkin toimivat.

Tavoitepistemäärä 5

# Käyttöohjeet

Kokoomalla normaalisti g++ \*.cpp -o a , saa rakennettua ohjelman kansioon nimellä a. Saattaa herjata joitain varoituksia lähinnä siksi, että yksi muuttuja alustetaan NULLiksi.

Kohdat ovat aika simppelisti siten, että haluttu algoritmi valitaan valikosta 1 – 6, 5:n pisteen performance tehtävä on 7:n ja pois pääsee valitsemalla 8.

Graphical user interface

Description automatically generated with low confidence

Valikot pistejärjestyksessä menevät:

* 1 = 1 pst
* 2 = 2 pst
* 3 = 3 pst
* 5 = 5 pst
* 4, 6 & 7 – 5 pst (4 ja 6 suoraan yhden algoritmin ajo, 7 performance)

# Toiminta

Ohjelma toimii siten, että valittaessa jokin ajo kysytään käyttäjältä kuinka monella numerolla halutaan täyttää lista, tämän tulee olla positiivinen kokonaisluku. Search-funktiot täyttävät listan ennalta määrätysti siten, -menetelmällä, puolestaan Sort-funktioissa randomi kyselee sen jälkeen käyttäjältä satunnaislukugeneraattoria varten raja-arvot järjestyksessä minimi ja maksimi.

Tämän jälkeen ohjelma tulostaa konsolille generoidun listan, ajaa funktion, kertoo kuinka monta vertailua tehtiin ja montako millisekuntia aikaa kului, paitsi search-funktioiden kohdalla tulostetaan nanosekunnit. Lopuksi ohjelma tulostaa järjestellyn listan käyttäjälle ja päästää tämän valitsemaan seuraavan ajettavan kohdan tai vaihtoehtoisesti

Performancen listan koko riippuu kierroksesta ja on siis ja satunnaislukugeneraattori arpoo listalle luvut väliltä 0 - 1’000’000.

Satunnaislukugeneraattori käyttää valmiiksi aiemmin jostain löytämääni Mersenne-Twister -generaattoria, joka on notaatiossa ja lyhyydessään hyödyllinen:

int random\_gen(int min, int max) {

    std::random\_device rd;     // only used once to initialise (seed) engine

    std::mt19937 rng(rd());    // random-number engine used (Mersenne-Twister in this case)

    std::uniform\_int\_distribution<int> uni(min, max); // guaranteed unbiased

    return uni(rng);

}

# Ratkaisut

Koodin pääohjelma toimii switch-case muodolla, joka puolestaan on do-while loopissa. Lisäksi on kosolti apufunktioita, joiden tarkoitus on joko arpoa numeroita, tyhjätä lista performance-kierroksen välissä tai ottaa käyttäjäsyöte. Ohessa funktioiden määrittelyt.

**functions.h**

#pragma once

#include "header.h"

int initialize();

int ask\_size();

void print\_result(const Error\_code& error, const int& pos, const int& comp, const int64\_t time);

int get\_int();

int random\_gen(int, int);

void fill\_list\_integers\_det(List& List);

void fill\_list\_integers\_rand(List& list, bool performance, int round);

void performance(List& list);

void reset(List& list, int round);

// --- Search & Sort ---

void sequential\_search(List& sList, const Key& key);

void binary\_search(List& sList, const Key& key);

// n^2

void insertion\_sort(List& sList);

void selection\_sort(List& sList);

// n log n

void merge\_sort(List& sList);

void tree\_sort(List& sList);

// Merge\_sort helpers:

Node<Key>\* merge\_sort\_nodes(Node<Key>\* headNode);

Node<Key>\* split(Node<Key>\* source);

Node<Key>\* merge\_sorted\_nodes(Node<Key>\* first, Node<Key>\* second);

// Tree\_sort helpers:

Node<Key>\* tree\_sort\_nodes(Node<Key>\* head);

Node<Key>\* build\_tree(Node<Key>\* root, Node<Key>\* branch);

Node<Key>\* order\_tree(Node<Key>\* root);

Ratkaisuna käytin double linked -list tyyppistä listaa, joskin ehkä olisi ollut järkevämpi käyttää pelkkiä nodeja listan sijaan. Ymppäsin kaikki ominaisuuden yhteen listaluokkaan, josta poistin lopulta jopa templaten, koska tarkoituksena oli lähinnä sortata kokonaislukuja. Käytän kuitenkin kaksoislinkattua Node-template structia, jossa on ohjelmassa Key-luokka sisällytettynä. Listan funktiot ovat päällisinpuolin samoja, kuin kurssilla käytetyt, muutamia printtausfunktioita tein. Itse search- ja sort funktiot käyn tuonnempana läpi. Double-linked listin käyttö aiheutti hieman hankaluuksia joidenkin implementointien kanssa, mutta niistä selvittiin.

**list.h**

#pragma once

#include "header.h"

class List {

public:

    List();

    List(const List &copy);

    ~List();

    void operator = (const List &copy);

    int size() const;

    bool empty() const;

    bool full() const;

    void clear();

    Error\_code insert( int position, const Key &x );

    Error\_code replace( int position, const Key &x );

    Error\_code retrieve( int position, Key &x );

    Error\_code remove( int position, Key &x );

    void print\_current(bool one) const;

    void print\_prev\_next(bool which) const;

    void print\_list();

    void traverse( void (\*visit)(Key &) );

    void set\_position(int position);

    Error\_code sequential\_search(List &the\_list, const Key &target, int &dest);

    Error\_code binary\_search(List &l, const Key &target, int &dest);

    void insert\_sort(List &l);

    void select\_sort(List &l);

    void merge\_sort(List &l);

    void tree\_sort(List &l);

    int getComparisons() const;

protected:

//  Data members for the doubly-linked list implementation follow:

   int count;

   mutable int current\_position;

   mutable Node<Key> \*current;

//  The auxiliary function to locate list positions fozllows:

};

Node.h

#pragma once

#include "header.h"

template <typename Node\_entry>

struct Node {

//  data members

    Node\_entry entry;

    Node<Node\_entry> \*back;

    Node<Node\_entry> \*next;

//  constructors

   Node();

   Node(Node\_entry, Node<Node\_entry> \*link\_back = NULL,

                    Node<Node\_entry> \*link\_next = NULL);

};

template<class Node\_entry>

Node<Node\_entry>::Node()

{

    entry = NULL;

    next = nullptr;

    back = nullptr;

}

template<class Node\_entry>

Node<Node\_entry>::Node(Node\_entry item, Node<Node\_entry> \*link\_back, Node<Node\_entry> \*link\_next)

{

    entry = item;

    next = link\_next;

    back = link\_back;

}

key.h

#pragma once

#include "header.h"

class Key {

private:

   int key;

public:

   static int comparisons;

   Key();

   Key(int x);

   int get\_key() const;

   void set\_key(int x);

};

key.cpp

#include "header.h"

int Key::comparisons = 0;

Key::Key(){}

Key::Key(int x) { set\_key(x); }

int Key::get\_key() const {

    return key;

}

void Key::set\_key(int x){

    key = x;

}

operator.h

#pragma once

#include "header.h"

//  Declare overloaded comparison operators for keys.

  bool operator ==(const Key &x, const Key &y);

  bool operator > (const Key &x, const Key &y);

  bool operator < (const Key &x, const Key &y);

  bool operator >=(const Key &x, const Key &y);

  bool operator <=(const Key &x, const Key &y);

  bool operator !=(const Key &x, const Key &y);

operator.cpp

#include "operator.h"

bool operator == (const Key &x, const Key &y)

{

    Key::comparisons++;

    return x.get\_key() == y.get\_key();

}

bool operator > (const Key &x, const Key &y)

{

    Key::comparisons++;

    return x.get\_key() > y.get\_key();

}

bool operator < (const Key &x, const Key &y)

{

    Key::comparisons++;

    return x.get\_key() < y.get\_key();

}

bool operator >= (const Key &x, const Key &y)

{

    Key::comparisons++;

    return x.get\_key() >= y.get\_key();

}

bool operator <=(const Key &x, const Key &y)

{

    Key::comparisons++;

    return x.get\_key() <= y.get\_key();

}

bool operator != (const Key &x, const Key &y)

{

    Key::comparisons++;

    return x.get\_key() != y.get\_key();

}

## Funktiot

Pääohjelma kutsuu funktioita, joiden päätarkoitus on ajaa listan sisältä kyseinen funktio sekä ottaa dataa ylös kuten kulunut aika sekä vaikka vertailujen määrä. Yleisesti ne ovat kaikki lähes samannäköisiä:

void sequential\_search(List& sList, const Key& key) {

    Error\_code check;

    int pos;

    auto clock\_start = std::chrono::steady\_clock::now();

    check = sList.sequential\_search(sList, key, pos);

    auto clock\_end = std::chrono::steady\_clock::now();

    int64\_t time\_taken = std::chrono::duration\_cast<std::chrono::nanoseconds>(clock\_end - clock\_start).count();

    print\_result(check, pos, sList.getComparisons(), time\_taken);

}

## Search -funktiot

### Sequential search

List-luokan metodi sequential\_search etsii haetun muuttujan yksinkertaisella tavalla. Tämän oli valmisfunktio materiaaleissa, eikä vaatine kovinkaan paljon avaamista.

### Binary search

List-luokan metodi hakee pyydetyn muuttujan jakamalla haettavan alueen kahtia, kunnes data joko löydetään tai ei.

Error\_code List::binary\_search(List &l, const Key &target, int &dest)

{

    int start = 0,

        end = l.size();

    Key data;

        while(true){

           dest = (start+end)/2; //start from middle position

           l.retrieve(dest, data);

            if( operator == (data, target) ) return success;

            else if ( operator > ( data, target ) ) end = dest+1;

            else start = dest;

            if (start == end-1 || start == end) return not\_present;

        }

        return not\_present;

}

## n2 sort -funktiot

### Insert sort

Insert sort on ensimmäinen järjestelyfunktio. Käytännössä ajetaan listaa eteenpäin kunnes löytyy arvo, joka on suurempi kuin edellinen ja tämä laitetaan oikealle kohdalle.

void List::insert\_sort(List &l)

{

    Key temp,

        current\_val;

    int unsorted = 1,

        position = 0;

    for ( unsorted; unsorted < l.size(); unsorted++ ){

        position=unsorted-1;

        l.retrieve(position, temp);

        l.retrieve(unsorted, current\_val);

        if( operator < (current\_val, temp) ){

            l.remove(unsorted, current\_val);

            while(position >= 0 && operator < (current\_val, temp ) ) {

                position--;

                l.retrieve(position, temp);

            }

            l.insert(position+1, current\_val);

        }

    }

}

### Select sort

Funktiota ajetaan siten, että koko lista käydään läpi silmukassa, jonka sisällä oleva silmukka vertaa kaikki seuraavat arvot tämän hetken arvoon. Kun löytyy arvo, joka on pienempi kuin nyt löytynyt, laitetaan vallitseva arvo tämän tilalle. Kun kaikki listan arvot on käyty läpi, sijoitetaan pienin löydetty arvo listan vallitsevaan kohtaan ennen kuin siirrytään seuraavaan.

void List::select\_sort(List& l) {

    int position = 0, run\_through = 0;

    set\_position(position);

    Key min, temp;

    for (position; position < size(); position++) {

        retrieve(position, min);

        temp = min;

        for (run\_through = position+1; run\_through < size(); run\_through++) {

            retrieve(run\_through, temp);

            if (operator < (temp, min)) {

                replace(run\_through, min);

                min = temp;

            }

        }

        replace(position, min);

    }

}

## n log n -sort

Näiden algoritmien rakentelussa tuli hieman enemmän ongelmia johtuen double linked list -rakenteesta, mutta lopputulos on kohtuullisen hyvä. Näihin otin mukaan ulkopuolisia apufunktioita, ja itse listan sisäiset metodit kutsuvat vain aloittavaa apufunktiota ja sijoittavat position 0:n arvon node-ketjun päähän. Ratkaisu vaikutti kaikkein helpoimmalta toteuttaa lopulta.

### Merge sort

Kutsutaan funktiota merge\_sort\_nodes(), joka palauttaa aloitusnoden pään. Kyseessä on rekursiivinen funktio, joka ajaa itse itseään puolittaen aina sen listan, kunnes päästään tilanteeseen, että käsiteltävänä on enää vain yksi funktio.

Node<Key>\* merge\_sort\_nodes(Node<Key>\* first) {

    //stop doing recursive once there is nullptr in the node

    if (first == nullptr || first->next == nullptr) return first;

    Node<Key>\* second = split(first);

    //recursive, continue until both halves are sorted

    first = merge\_sort\_nodes(first);

    second = merge\_sort\_nodes(second);

    //merge double linked nodes, finally returns node with lowest key value

    return merge\_sorted\_nodes(first, second);

}

Halkaisu tapahtuu ns. Aisopoksen sadusta tutulla kilpikonna- ja jänis -metodilla split() -funktiossa, jossa jänis ottaa kaksi askelta, kilpikonnan ottaessa yhden. Kun jänis pääsee maaliin, on kilpikonna puolessa välissä. Lopussa node-ketju katkaistaan puolivälissä ns. asettamalla kilpikonnan seuraava node null pointeriksi ja palautetaan temp, joka on ennen katkaisu olevat toisen puoliskon pää. Edellisessä funktiossa tämä temp on toinen puolikas ensimmäisen ollessa lähtötilanteen pää. Metodi toimii rekursiivisessa halkaisussa erittäin hyvin.

Node<Key>\* split(Node<Key>\* source) {

    //Turtle-rabbit method to find middle point. Due to recursive splitting.

    Node<Key> \*rabbit = source,

            \*turtle = source,

            \*temp;

    while (rabbit->next != nullptr && rabbit->next->next != nullptr) {

        rabbit = rabbit->next->next;

        turtle = turtle->next;

    }

    temp = turtle->next;

    turtle->next = nullptr;

    return temp;

}

Kun ollaan päästy tilanteeseen, jossa kaikki noded on halkaistu, ajetaan puolikkaan sisään jälleen rekursiivisesti merge\_sorted\_nodes -funktiolla, joka palauttaa lopulta järjestetyn node-ketjun pään. Kyseessä on niin ikään rekursiivinen funktio, joka päättyy, kun toinen syötetty node on nullptr. Muuten tämä linkkailee toisiaan edes-takaisin sen mukaan kumman avain on suurempi.

Node<Key>\* merge\_sorted\_nodes(Node<Key>\* first, Node<Key>\* second) {

    //recursive merging, continue til either input is nullptr

    if (first == nullptr) return second;

    if (second == nullptr) return first;

    if (operator < (first->entry, second->entry)) {

        first->next = merge\_sorted\_nodes(first->next, second);

        first->next->back = first; //recursion, link backnode

        first->back = nullptr;

        return first;

    }

    else {

        second->next = merge\_sorted\_nodes(first, second->next);

        second->next->back = second;

        second->back = nullptr;

        return second;

    }

}

### Tree sort

Kutsutaan funktiota tree\_sort\_nodes(), tässä funktiossa luodaan kokonaan uusi node ketju ja tuhotaan vanha. Palautetaan node, joka järjestelyn jälkeen ensimmäinen. Aluksi tehdään uusi muistivaraus Nodelle, johon sijoitetaan funktioon tulleen listan aloitus noden pään arvo. Tämän jälkeen käydään listaa läpi siten, että jokaisen noden arvo sijoitetaan uuteen branch nodeen. Näiden perusteella rakennetaan binäärinen puu. Muistetaan tuhota vanha node aina ennen seuraavaa kierrosta ja lopuksi viimeinenkin node viemästä muistitilaa. Lopuksi ajetaan palautunut, järjestetty puu alkuun ja palautetaan alkupiste.

Node<Key>\* tree\_sort\_nodes(Node<Key>\* headNode) {

    Node<Key>\* root = new Node<Key>;

    root->entry = headNode->entry;

    //Run through whole list and build a tree based on entries

    //Tree is completely new list, where lowest node is returned

    while (headNode->next != nullptr) {

        headNode = headNode->next;

        Node<Key>\* branch = new Node<Key>;

        branch->entry = headNode->entry;

        root = build\_tree(root, branch);

        //delete old node since we are building completely new tree

        delete headNode->back;

    }

    delete headNode; //delete last node once tree is built;

    order\_tree(root); //order built tree

    while (root->back != nullptr)

        root = root->back; //Run back to first node to return it as a new head.

    return root;

}

Puunrakennus tapahtuu build\_tree() -funktiolla. Funktio ajetaan puun kasvaessa rekursiivisesti siten, että juuriarvoa pienemmät arvot sijoitetaan aina vasemmalle, tässä tapauksessa taakse, ja suuremmat puolestaan oikealle, eteen. Kun päästään puun loppuun ja arvo on null, sijoitetaan syötetty oksa tämän tilalle.

Node<Key>\* build\_tree(Node<Key>\* root, Node<Key>\* branch) {

    //back is left, next is right

    //recursive

    if (root == nullptr) {

        root = branch;

        return root;

    }

    else if (operator > (root->entry, branch->entry)) {

        root->back = build\_tree(root->back, branch);

    }

    else {

        root->next = build\_tree(root->next, branch);

    }

    return root;

}

Kun puu on rakennettu, se pitää vielä järjsetää order\_tree() -funktiolla. Tässä lähdetään käymään ns. juuresta (eli jos ajatellaan kolmiona, kärjestä) ensin vasemmalle (pienemmät) ja siten oikealle (suuremmat) järjestyksessä, kunnes saavutetaan sen ketjun pienin tai suurin arvo (riippuen kumpaan suuntaan kuljetaan). Kyseessä on jälleen rekursiivinen funktio. Linkkaaminen tapahtuu siten, että kun päästään tilanteeseen, jossa sekä vasen, että oikea haara on käyty läpi, ajetaan link\_branch while-loopissa ensin vasemman (pienemmät) haaran oikeampaan laitaan (joka on sen haaran isoin) ja päin vastoin. Nämä linkataan sitten juuri nodeen, jolloin saadaan yhtenäinen ketju. Viimeinen paluuarvo on alkuperäinen juurinode, josta on nyt yhtenäinen kaksisuuntainen linkitetty nodeketju listan pienimmästä suurimpaan.

Node<Key>\* order\_tree(Node<Key>\* root) {

    if (root == nullptr) {

        return root;

    }

    else{

        root->back = order\_tree(root->back);

        root->next = order\_tree(root->next);

        Node<Key> \*link\_branch = new Node<Key>;

        if (root->back != nullptr) {

            link\_branch = root->back;

            //Run to the biggest node to the left (lower than root)

            while (link\_branch->next != nullptr) link\_branch = link\_branch->next;

            //link greatest of the lower than nodes as a next of the root

            link\_branch->next = root;

            root->back = link\_branch;

        }

        if (root->next != nullptr) {

            link\_branch = root->next;

            //Run to the lowest node to the right (greater than root)

            while (link\_branch->back != nullptr) link\_branch = link\_branch->back;

            //link lowest of the greater than nodes as a next of the root

            link\_branch->back = root;

            root->next = link\_branch;

        }

    }

## Performance

Tämä funktio ei oikeastaan tee muuta kuin aja 10 kierrosta kutakin sort-funktiota tulostaen testin arvot (jotka on mukana tulosteessa). Lista resetoidaan (tyhjätään ja täytetään uudestaan random-numeroilla) jokaisen funktioajon jälkeen sekä vertailujen määrä nollataan.

void performance(List &list){

    int round = 1;

    std::cout << "running 10 rounds of performance on search functions: \n";

    for(round; round<=10; round++){

        std::cout << "\n\nRound#"<< round << ", list size: " << round\*1000 << ":\n";

        reset(list, round);

        std::cout << "\nInsertion sort: \n";

        insertion\_sort(list);

        reset(list, round);

        std::cout << "\nSelection sort: \n";

        selection\_sort(list);

        reset(list, round);

        std::cout << "\nMerge sort: \n";

        merge\_sort(list);

        reset(list, round);

        std::cout << "\nTree sort: \n";

        tree\_sort(list);

    }

    list.clear();

    std::cout << "\n\nFINISH\n";

}

# Ajo

Ajetaan valikon kaikki läpi kertaalleen:

Sequential search:

A screenshot of a computer

Description automatically generated with low confidence

Binary search:

A screenshot of a computer

Description automatically generated with low confidence

Jo näin pienellä tutkimisella lie selvää, että binary search on pääasiassa nopeampi. Toisaalta, jos hettu luku on listan alkupäässä, on sequential search nopeampi.

Insertion sort:

A cityscape at night

Description automatically generated with low confidence

Selection sort:

A picture containing text, night sky

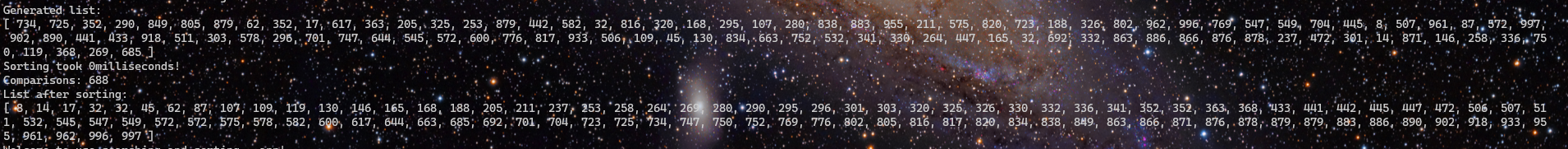
Description automatically generated

Merge sort:

A cityscape at night

Description automatically generated with low confidence

Tree sort:



100:lla muuttujalla näissä ei vielä liiemmin eroa ole, joten ajetaan performance.

Performance:

A picture containing text

Description automatically generated

Ero alkaa olla aika huomattava n^2 ja n log n -funktioiden välillä niin ajassa kuin vertailujen määrässäkin.