Algmet-kompendium

Login - dugnadsarbeid

December 2023

Notes

- Big thanks to the contributors over at `https://github.com/bksuup/algmet`
  - Buy them a coffee if you find them in the lounge :)
- The bigger code-segments have been converted to english and refactored for improved readability
- These segments are functionally identical, and will produce the same results as Frodes examples.
- Untranslated code is ripped straight from examples

Contents

1 Useful Info

1.1 Table: Alphabet/Number

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

P	Q	R	S	T	U	V	W	X	Y	Z	Æ	Ø	Å
16	17	18	19	20	21	22	23	24	25	26	27	28	29

2 Prefix-, infix- and postfix-notation

2.1 Infix to Postfix

Infix expressions are written in the form:

```
(( 8 + 12 ) * ( 17 + 4 ))
```

Postfix expressions are written in the form:

```
8 12 + 17 4 + *
```

2.1.1 Algorithm

1. Push all operators '+' and '*' onto the stack.
2. Directly output digits/numbers.
3. Pop and output the operator when a ')' end parenthesis is found.
4. Ignore all '(' start parenthesis.

2.1.2 Code

```
while ((tegn = cin.get()) != '\n'){ // reads all numbers and signs until \n
    if(tegn == ')'){ // if it finds an end parenthesis
        cout << stakk.top(); // prints the top sign of the stack
        stakk.pop();         // removes the sign
    }

    // if it finds + or *, it is added to the stack
    else if (tegn == '+' || tegn == '*') stakk.push(tegn);

    while (tegn >= '0' && tegn <= '9'){
        cout << tegn;
        tegn = cin.get();
    }

    if (tegn != '(')
        cout << ' ';
}
```

2.2 Postfix to Answer

Calculating postfix expressions like: 8 12 + 17 4 + * = 420 using the stack.

2.2.1 Algorithm

1. When you find '+' or '*', pop two numbers/operands.
2. Their sum/product is calculated.
3. The answer is pushed onto the stack.
4. When digits are found, they possibly build a continuous number, which is then pushed onto the stack.

2.2.2 Code

```
while ((tegn = cin.get()) != '\n'){ // reads everything until \n
    tall = 0;                       // reset to zero
    while (tegn == ' ') tegn = cin.get(); // skips blanks

    // if we read '+' or '*', we take the top two numbers and do
    // the corresponding operation on them
    if (tegn == '+'){
        tall = stakk.top(); stakk.pop(); // tall = top number on the stack
        tall += stakk.top(); stakk.pop(); // adds the new number now on top
    } else if (tegn == '*'){
        tall = stakk.top(); stakk.pop();
        tall *= stakk.top(); stakk.pop();
    }

    // building a multi-digit number
    while (tegn >= '0' && tegn <= '9'){
        tall = (10 * tall) + (tegn - '0'); // converts from ascii to number
        tegn = cin.get();
    }

    stakk.push(tall); // pushes built number
}
```

3 Trees

3.1 Parse tree

Man kan bygge et parse-tre fra et postfix uttrykk. Ett parse-tre er et binært tre hvor bladnodene har tall, og alle interne noder har operatorerne '+' eller '' F.eks; 3 4 + 3 2 * + 2 + 5 3 * 4 2 + * + gir parse treet: [MISSING IMAGE]

3.1.1 Algoritme:

1. Leser ett og ett tegn (bokstav/siffer, '+' eller '')
2. Er det bokstav / siffer, pushes den til stakken
3. er det '+' eller '', pop'es av stakken det som blir høyre og venstre noder, så blir rotnoden pushet på stacken.

NB!

- postfix uttrykket må bestå av kun **EN** bokstav / **ETT** siffer, '+' og ''
- Uttrykket kan ikke avsluttes med en eller flere blanke.

3.1.2 Code:

```
struct Node{
        char ID;
        Node *left, *right;
        Node(char id){
                ID = id;
                left = right = nullptr;
        }
}

/* ... */

while ((tegn = cin.get()) != '\n'){
        while(tegn == ' ') {
        tegn = cin.get();
    }
        nyNode = new Node(tegn);
        if (tegn == '+' || tegn == '*'){
                nyNode->right = stakk.top(); stakk.pop();
                nyNode->left = stakk.top(); stakk.pop();
        }
        stakk.push(nyNode);
}
```

3.2 Tree Traversal

3.3 Node

```
struct Node {
    char ID;
    bool besokt; // 'besokt' brukes KUN ifm. postorder.
    Node *left, *right; // Initierende constructor:
    Node(char id) { ID = id; left = right = nullptr; besokt = false; }
};
```

3.4 Preorder

3.4.1 Algoritme

Besøker seg selv før den traverserer.

1. Besøk seg selv
2. Traverser Venstre
3. Traverser Høyre

3.4.2 Kode

```
void traverserPreorder(Node* node){
    if(node){
        gStakk.push(node);
        while (!gStakk.empty()){
            node = gStakk.top(); gStakk.pop();
            besok(node);
            if (node->right) gStakk.push(node->right);
            if (node->left) gStakk.push(node->left);
        }
    }
}
```

3.5 Inorder

3.5.1 Algoritme

Besøker seg selv 'mellom' traverseringen

1. Traverser Venstre
2. Besøk seg selv
3. Traverser Høyre

3.5.2 Kode

```
void traverserInorder(Node* node){
    while (node || !gStakk.empty()){
        if (node) {
            gStakk.push(node);
            node = node->left
        } else{
            node = gStakk.top(); gStakk.pop();
            besok(node);
            node = node->right;
        }
    }
}
```

3.6 Postorder

3.6.1 Algoritme

Besøker seg selv etter traversering

1. Traverser Venstre
2. Traverser Høyre
3. Besøk seg selv

```
void traverserPostorder (Node* node) {
    if (node) {
        gStakk.push(node);
        while (!gStakk.empty()){
            node = gStakk.top(); gStakk.pop();
            if (node->left->besokt = false) {
                gStakk.push(node->left);
            }
            if (node->right->besokt = flase) {
                gStakk.push(node->right);
            }
            besok(node);
        }
    }
}
```

3.7 Levelorder

3.7.1 Algoritme

1. Leser treet linjevis

3.7.2 Kode

```
void traverserLevelorder(Node* node){
    if (node){
        gKo.push(node);
        while (!gKo.empty()){
            node = gKo.front(); gKo.pop();
            besok(node);
            if(node->left) gKo.push(node->left);
            if(node->right) gKo.push(node->right);
        }
    }
}
```

Trees - searching

DFS - Depth First Search

Kode

Sjekker rekursivt om en nodes naboer er besøkt eller ikke, og markerer noden som besøkt når den besøkes av DFS funksjonen

```
/**
* @param nr - indeks (0 til ANTNODER-1) for noden som skal besøkes
*/
void DFS(const int nr) {
    gBesokt[nr] = ++gBesoktSomNr; // gir noden riktig "besøkt som" nr

    for (int j = 0;  j < ANTNODER;  j++) // sjekker hele "linjen" til noden i en nabo-matrise
        if (gNaboMatrise[nr][j]) // hvis forskjellig fra 0
            if (gBesokt[j] == USETT) DFS(j); // rekursivt besøk noden.
}
```

Framgangsmåte

1. Lag en stack og en besøkt-liste
2. Plasser startnoden på stacken
3. Begynn DFS loopen
    (a) Så lenge stacken ikke er tom
        i. Pop en node fra stakken (aktuell node)
        ii. Besøk den aktuelle noden
        iii. Få alle nabo-noder til den aktuelle noden
        iv. For hver nabo-node
            A. Marker som besøkt
            B. Push på stakken

3.8 Breadth First Search

Kode

Iterativ algoritme som gjør et bredde først søk.

```
/**
 *  @param   nr  - Indeks (0 til ANTNODER-1) for STARTNODEN i besøket
 */
 void BFS(int nr) {
    int j;                                  //  Indeks for aktuelle naboer.
    gBesokeSenere.push(nr);                 //  Legges BAKERST i besøkskø.
    while (!gBesokeSenere.empty()) {        //  Ennå noder å besøke igjen:
        nr = gBesokeSenere.front();         //  AVLES iden til første noden på køen
        gBesokeSenere.pop();                //  FJERNER/TAR UT fra køen.
        gBesokt[nr] = ++gBesoktSomNr;       //  Setter besøksnummeret.
        for (j = 0;  j < ANTNODER;  j++)    //  Nodens linje i matrisen:
            if (gNaboMatrise[nr][j])        //  Er nabo med nr.j,
                if (gBesokt[j] == USETT) {  //    og denne er ubesøkt:
                    gBesokeSenere.push(j);  //  Legger nabo BAKERST i køen.
                    gBesokt[j] = SENERE;    //  Setter at delvis besøkt!!!
                }
    }
}
```

Fremgangsmåte

1. Lag en kø og en liste over besøkte noder
   - Køen brukes for å holde orden på hvilken node som skal besøkes neste
   - Listen holder orden på hvilke noder som har blitt besøkt
2. Plasser startnoden på køen og marker den som besøkt
3. BFS Loop
   - Så lenge køen ikke er tom
     - Hent neste node fra starten på køen (aktuell node)
     - Besøk noden.
     - Se alle noder som er sammenkoblet med den aktuelle noden.
     - For hver node som er sammenkoblet med den aktuelle noden:
       * Marker den som besøkt.
       * Plasser den på køen.
4. Gjenta til køen er tom.
   - Hvis en nabonode er markert besøkt, ikke besøk noden på nytt igjen.

3.9 2-3-4 trees

3.9.1 Description

In computer science, a 2–3–4 tree (also called a 2–4 tree) is a self-balancing data structure that can be used to implement dictionaries. The numbers mean a tree where every node with children (internal node) has either two, three, or four child nodes:

- a 2-node has one data element, and if internal has two child nodes;
- a 3-node has two data elements, and if internal has three child nodes;
- a 4-node has three data elements, and if internal has four child nodes;

2–3–4 trees are B-trees of order 4; like B-trees in general, they can search, insert and delete in $O(\log n)$ time. One property of a 2–3–4 tree is that all external nodes are at the same depth.

2–3–4 trees are isomorphic to red–black trees, meaning that they are equivalent data structures. In other words, for every 2–3–4 tree, there exists at least one and at most one red–black tree with data elements in the same order. Moreover, insertion and deletion operations on 2–3–4 trees that cause node expansions, splits and merges are equivalent to the color-flipping and rotations in red–black trees. Introductions to red–black trees usually introduce 2–3–4 trees first, because they are conceptually simpler. 2–3–4 trees, however, can be difficult to implement in most programming languages because of the large number of special cases involved in operations on the tree. Red–black trees are simpler to implement, so tend to be used instead.

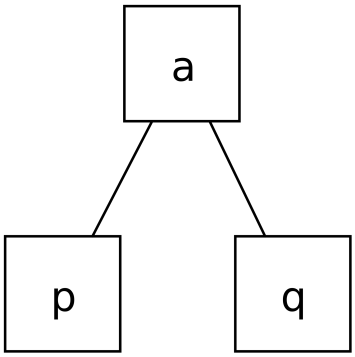

Figure 1: 2-node

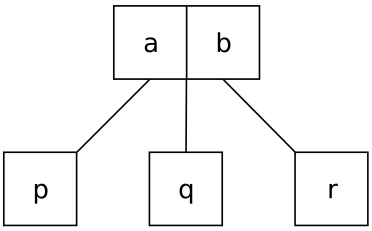

Figure 2: 3-node

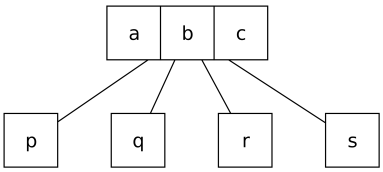

Figure 3: 4-node

3.9.2 Properties

- Every node (leaf or internal) is a 2-node, 3-node or a 4-node, and holds one, two, or three data elements, respectively.
- All leaves are at the same depth (the bottom level).
- All data is kept in sorted order.

3.9.3 Insertion

To insert a value, we start at the root of the 2-3-4 tree:

1. If the current node is a 4-node:
   - Remove and save the middle value to get a 3-node.
   - Split the remaining 3-node up into a pair of 2-nodes (the now missing middle value is handled in the next step).
   - If this is the root node (which thus has no parent):
     - the middle value becomes the new root 2-node and the tree height increases by 1. Ascend into the root.
   - Otherwise, push the middle value up into the parent node. Ascend into the parent node.
2. Find the child whose interval contains the value to be inserted.
3. If that child is a leaf, insert the value into the child node and finish.
   - Otherwise, descend into the child and repeat from step 1.

3.10 Red-Black Trees

3.10.1 Converting a 2-3-4 Tree to a Red-Black Tree

Because 2-3-4 trees and red-black trees are structurally similar, we can easily convert from one to the other. The key principle in this conversion is the representation of the 2-3-4 tree nodes in terms of red and black nodes of the red-black tree.

The basic rules for this conversion are as follows:

- A 2-node in a 2-3-4 tree is represented as a black node in a red-black tree.
- A 3-node in a 2-3-4 tree is represented as a black node with one red child in a red-black tree.
- A 4-node in a 2-3-4 tree is represented as a black node with two red children in a red-black tree.

This conversion maintains the essential properties of the red-black tree, such as the black-height property and the red-node property. By following these rules, one can transform any 2-3-4 tree into an equivalent red-black tree that represents the same set of data with the same ordering.

4 Recursion

Rekursjon: en funksjon som blant annet kaller/bruker seg selv, og har en stoppe-betingelse som stopper den fra å kalle seg selv.

- Brukes når et problem kan deles i mindre, enklere under-deler.
- Hvert underproblem kan løses ved å anvende samme teknikk.
- Hele problemet løses ved å kombinere løsningene på underproblemene.

4.1 Preorder Rekursiv

```
void traverserPreorder (Node* node) {
    if (node) {
        gNivaa++;
        besok(node);
        traverserPreorder(node->left);
        traverserPreorder(node->right);
        gNivaa--;
    }
}
```

4.2 Inorder Rekursiv

```
void traverserInorder (Node* node) {
    if (node) {
        gNivaa++;
        traverserInorder(node->left);
        besok(node);
        traverserInorder(node->right);
        gNivaa--;
    }
}
```

4.3 Postorder Rekursiv

```
void traverserPostorder (Node* node) {
    if (node) {
        gNivaa++;
        traverserPostorder(node->left);
        traverserPostorder(node->right);
        besok(node);
        gNivaa--;
    }
}
```

5 Sorting algorithms

5.1 Common functions

```
/**
 *  Swaps value of two variables passed by reference(!)
 *  Function uses a template value, IE it can be used with any
 *  variable types.
 *
 *  @param   a  -  Value to be swapped with b
 *  @param   b  -  Value to be swapped with a
**/
template <typename T>
void swap(char &a, char &b) {
    const char tmp = a;
    a = b;
    b = tmp;
}
```

5.2 Selection Sort

5.2.1 Algorithm

Fremgangsmåte:

1. Finn den minste verdien i det usorterte arrayet.
2. Bytt det med skuff[0]
3. Finn den minste verdien i resten av det usorterte arrayet.
4. gjenta til arrayet er sortert.

Algoritme:

1. Finner det minste elementet i resten av arrayen (f.o.m. nr 'i')
2. Legger det på plass nr 'i'

Orden:
(N*N)/2 NB:

- Er linær når verdiene som sammenlignes er små, og tilhørende data er store/mye.
- Ikke stabil: rekkefølgen på like elementer har ikke samme innbyrdes rekkefølge etter sortering.

5.2.2 Code

```
/**
 *  @param   a  -  value to be swapped with b
 *  @param   b  -  value to be swapped with a
**/
void swap(int &a, int &b) {
    const int tmp = a;
    a = b;
    b = tmp;
}

/**
 *  Sorts an input-array incrementally using the selection sort algorithm.
 *  @param   arr  -  Array to be sorted
 *  @param   n    -  Number of elements in 'arr'
 *  @see     swap(...)
**/
void selectionSort(int arr[], const int n) {
        int i, j, minIndex; // Miniumum Index - smallest value in interval

    for (i = 0;  i < n-1;  i++) {
        minIndex = i;
        for (j = i+1;  j < n;  j++) {
            if (arr[j] < arr[minIndex]){
                    minIndex = j;
            }
        }
        if (minIndex != i) {
                    swap(arr[minIndex], arr[i]);
        }
    }
}
```

5.3 Insertion Sort

5.3.1 Algorithm

Fremgangsmåte:

1. Start med element i indeks 1
2. Sammenlign element i forrige indeks
   (a) Hvis indeks[n+1] er mindre enn indeks[n] -> flytt den mindre verdien til venstre, helt til den er større enn verdien som ligger i skuffen den sammenlignes med.
   (b) Hvis indeks[n+1] er større enn indeks[n] -> la de være
3. Gå videre til neste element

Algoritme:

1. Start i indeks 1 og gå videre ut i arrayet
2. For hver iterasjon, hent verdien til arr[i]
3. Sammenlign arr[i] med den sorterte delen av arrayet
   (a) hvis et større element finnes, forskyv det en til plass til høyre
4. Sett inn nøkkelen i posisjonen (sorterte delen) hvor alle elementer til venstre er mindre eller lik, og alle elementer til høre er større eller lik
5. Repeat

Orden (N*N)/4
NB!

- Er nærmest linær for så godt som ferdig sorterte arrayer.
- Veldig kjapp når et stort sortert array får flere verdies som legges til bakpå, og skal sorteres inn i arrayet.

5.3.2 Code

```
/**
 *  Insertion sort algorithm
 *  IMPORTANT: This uses a sentinel key at arr[0]
 *  @param   arr  -  Array to be sorted
 *  @param   n    -  Number of elements in 'arr'
**/
void insertionSort(int arr[], int n) {
        int i, key, j;
        for (i = 1; i < n; i++) {
                key = arr[i];
                j = i - 1;

                /* Move elements of arr[0..i-1], that are
                        greater than key, to one position ahead
                        of their current position */
                while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                        j--;
                }
                arr[j + 1] = key;
        }
}
```

5.4 Shellsort

5.4.1 Algorithm:

NB!

- bruker sentinel key
- Ikke stabil algoritme

Fremgangsmåte:

1. Velg en størrelse 'gap'
   (a) vanligvis brukes n/2 hvor n er antall elementer i arrayet
2. Sorter arrayet for hver størrelse av 'gap'
   (a) For hver verdi av 'gap' -> gjør en insertion sort med elementet som er i indeks: akutuell indeks + gap
3. Reduser størrelsen på 'gap'
   (a) når du har sortert på en størrelse 'gap', reduser størrelsen
   (b) gjør sortering på arrayet med den nye 'gap' størrelsen
   (c) fortsett til 'gap' = 1
4. Insertion sort
   (a) Når du har en 'gap' på 1, kjør en standard insertion-sort på arrayet.

Algoritme:

1. Start med en predefinert størrelse 'gap'
2. Sammenlign element 'n', med element 'n+gap'
3. hvis element n > n+gap, så bytter elementene plass
4. Når alle elementene har blitt sammenlignet, halver størrelsen på 'gap'
5. Gjenta til gap = 1.
6. Kjør en vanlig insertion sort.

Orden Gjør aldri mer enn $N^(3/2)$*sammenligninger*.

5.4.2 Code:

```
/**
 *  Sorts an array incrementally using shellsort.
 *  IMPORTANT: This uses a sentinel key at arr[0]
 *  @param   arr  -  Array to be sorted
 *  @param   n    -  Number of elements in 'arr'
**/
void shellSort(char arr[], const int n) {
    int  i, j, h;
    char tmp;
    for (h = 1;  h <= n/9;  h = (3*h)+1){
            ; // Empty for loop, initialize h-variable to adequate gapsize
    }

    while (h > 0)  {
        for (i = h+1;  i < n;  i++) {
            tmp = arr[i];
            j  = i;

```

```
            while (j > h  &&  arr[j-h] > tmp) {
                arr[j] = arr[j-h];
                j -= h;
            }
            arr[j] = tmp;
        }
        // The H value will eventually reach 1.
        // When this happens, the algorithm is identical to insertion sort
        h /= 3;
    }
}
```

5.5 Quicksort

5.5.1 Algorithm:

Prinsipp: Splitt og hersk.

- Splitt i to, og sorter hver del

Fremgangsmåte:

1. Velg en 'pivot' (gjerne midten av arrayet -> n/2 hvor n er antall elementer)
2. Partisjoner arrayet
   (a) Plasser en peker på starten av arrayet, og en på slutten
      i. flytt pekerene mot hverandre, og sammenlign elementene de peker på
   (b) bytt verdiene slik at alle elementer mindre enn pivot -> venstre for pivot, og alle elementer større enn pivot -> høre for pivot.
   (c) når pekerene møtes, plasser elementet i 'pivot' i sin korrekte possisjon i det sorterte arrayet.
3. gjør steg 2 rekursivt til sub-arrayene
4. gjenta til arrayet er sortert.

Algoritme

1. Velg arr[hoyre] som et tilfeldig element å sortere ut fra. (partisjonselementet)
2. Lete fra venstre etter >= verdi, og fra høyre etter <= verdi og så bytte om på disse
3. Gjenta til letingen har passert hverandre.
4. Bytt partisjonselementet med den helt til høyre i venstre delarray (nå er partisjonselementet i arr[i] på sin endelige plass, og alle elementer til venstre er <= partisjonselementet, og alle elementer til høyre er >= partisjonselementet).
5. Gjenta rekursivt.

Orden Quicksort bruker i gjennomsnitt '2N ln N' sammenligninger, worst case '(N*N)/2'

5.5.2 Code:

```
/**
 *  Reorders an array according to the following principles:
 *      - elements are sorted in place
 *      - elements <= pivot element will be placed to the left of
 *          the pivot element in the array.
 *      - elements >= pivot element will be placed to the right of
 *          the pivot element in the array.
 *      - The resulting sub-arrays to the left and right of the
 *          pivot element are not necesserily sorted.
 *
 *  @param    arr       - Array to be partitioned
 *  @param    left      - Lower array index for partitioning
 *  @param    right     - Upper array index for partitioning
 *  @return             - The array index where the pivot element was placed
**/
int partition(char arr[], const int left, const int right) {
    if (right <= left) { return 0; }

    int i = left -1;
    int j = right;
    const char pivotElement = arr[right];

    while (true) {
        while (arr[++i] < pivotElement) { ; /* incrementing i */ }
```

```
25          while (arr[--j] > pivotElement) { ; /* decrementing j */ }
26          if (i >= j) {
27              break;
28          }
29          swap(arr[i], arr[j]);
30      }
31      swap(arr[i], arr[right]);
32      return i;
33  }
34
35  /**
36   *  Recursively sorts a char-array incrementally using quicksort
37   *
38   *  @param   arr      -  Arrayen som skal sorteres
39   *  @param   left     -  Nedre/venstre indeks for sorteringsintervall
40   *  @param   right    -  Øvre/høyre    indeks for sorteringsintervall
41   *  @see     partition(...)
42  */
43  void quickSort(char arr[], const int left, const int right)  {
44      if (right > left) {
45          const int index = partition(arr, left, right);
46          quickSort(arr, left, index-1);
47          quickSort(arr, index+1, right);
48      }
49  }
```

5.6 Heapsort

5.6.1 Algorithm:

Steps:

1. Tar en array som skal sorteres og starter halvveis uti, og går baklengs til dens start.
2. For hvert element utføres 'downHeap'.
3. Når man har kommet til første elementet, oppfyller hele den originale arrayen heap-betingelsen.
4. Bytt det aller første, og til enhver tid siste elementet, og utfører 'downHeap' for hver gang.
5. Teller stadig ned antall elementer i arrayen som er igjenn å sortere.

Orden bruker færre enn 2N lg N sammenligninger (selv i 'Worst Case').
NB!

- Bottom-up heap-konstruksjon er tidslinær.
- Ustabil

5.6.2 Code:

```
/**
 * Sorts an unsigned char-array incrementally with (bottom-up) heapsort.
 * Sorterer en unsigned char-array STIGENDE med (Bottom-Up) HEAPSORT.
 *
 *  @param   arr      -  Array to be sorted
 *  @param   n        -  number of elements in 'arr'
 *  @see     Heap.downHeap(...)
 *  @see     swap(...)
 *  @see     heap(...) - under datastructures
**/
void heapSort(unsigned char arr[], int n)  {
    for (int keyNum = n/2;  keyNum >= 1;  keyNum--) {
        gHeap.downHeap(arr, n, keyNum);
    }
    while (n > 1) {
        swap(arr[1], arr[n]);
        gHeap.downHeap(arr, --n, 1);
    }
}
```

6 Searching Algorithms

6.1 Binary search - array

```c
/**
*  Søker BINÆRT i en SORTERT array.
 *
 *  @param    arr    - The sorted array to be searched.
 *  @param    target - The value to be searched for in 'arr'.
 *  @param    n      - Amount of elements in 'arr'.
 *  @return   The array-index of 'arr' where 'target' was found.
 *            0/NULL if 'target' was not found.
 */
int binarySearch(const int arr[], const int verdi, const int n) {
    int left = 1,              // Left boundary of search interval.
        right = n,             // Right boundary of search interval.
        middle;                // Center/middle of the search boundary.

    while (left <= right) {
        middle = (left + right) / 2;
        if (target == arr[middle]) { return middle; }
        if (target < arr[middle])  { right = middle - 1; }
        else { left = middle + 1; }
    }
    return 0;
}
```

6.2 Sequential search - array

```c
/**
 *  Sequentially searches in an array.
 *  Can be used on UNSORTED arrays.
 *
 *  @param    arr    - The sorted array to be searched.
 *  @param    target - The value to be searched for in 'arr'.
 *  @param    n      - Amount of elements in 'arr'.
 *  @return   The array-index of 'arr' where 'target' was found.
 *            0/NULL if 'target' was not found.
 */
int sequentialSearch(const int arr[], const int target, const int n) {
    int index = n + 1;
    while (index > 0  &&  target != arr[--index])  { ; } //empty loop, for finding target
    return index;
}
```

7 Hashing

7.0.1 Separate Chaining

Det brukes en array/vector med Stacker eller LIFO-lister. Når en nøkkel hashes til en indeks i arrayen/vectoren, så settes den bare inn aller først i stacken/listen. Er det derfor N nøkler som skal hashes inn i en array/vector som er M lang, så vil det gjennomsnittlig være N/M elementer/nøkler i hver stack/liste. Greit å bruke denne metoden når N er så stor at det er lite hensiktsmessig i bruke en array/vector der det er plass til alle nøklene/elementene.

7.1 Linear Probing

Nøkkelen hashes til indeksen der den bør legges. Er det allerede opptatt der, forsøkes den lagt inn i første etterfølgende ledige indeks. Når man arrayens slutt, så startes det med leting forfra igjen. Er arrayens lengde satt stor nok, så er vi garantert å finne en ledig plass!

Fremgangsmåte:

1. Finn hash-verdien for et element
2. Hvis plassen elementet skal plasseres er ledig -> plasser elementet på den plassen
   (a) Hvis plassen ikke er ledig -> prøv plass+1 til du finner en ledig plass å plassere elementet

7.2 Double Hashing

Den store ulempen med Linear Probing er «clustering». Dvs. sammenklumping av nøkler som har blitt hashet til omtrent de samme indeksene. Dette kan forbedres ved at når en krasj oppstår, så letes det ikke bare i en og en fortløpende indeks etterpå. I stedet får de ulike nøklene litt forskjellige tilleggsverdier som det sjekkes om vedkommende indeks er ledig i stedet. F.eks. at en nøkkel sjekker hver andre indeks utover, mens en annen sjekker hver sjette. Nøklene får ulike tilleggsverdier ved å kjøre den også igjennom en annen hash- funksjon. Denne kan f.eks. være: 6 - (nøkkel % 6) - som altså blir et tall i intervallet 1 til 6

Fremgangsmåte:

1. Hash elementet / verdien på 2 måter
   (a) Den første (hash1) bestemmer hvor vi skal prøve å plassere elementet først
   (b) Den andre (hash2) bestemmer hvor stor 'gap' vi skal hoppe for å prøve å plassere elementet hvis skuffen er opptatt.
2. Sjekk om plassen til det hashede elementet er opptatt eller ikke
   (a) Hvis ledig -> plasser elementet i (hash1)
   (b) Hvis ikke ledig -> sjekk hash1 + hash2 verdi
      i. Gjenta til vi finner en ledig skuff

7.3 Merkle trees

Beskrivelse: Et merkle tre er et binært tre sammensatt av hash-verdier.
Struktur

- Blad-noder:
  - Inneholder en Hash av en datablokk (datablokken er ikke en del av merkle treet).
- Intermediate-noder:
  - Inneholder en hash av sine 2 barn-noder (hash-barn1+hash-barn2, hashet).
- Rot-node:
  - Representerer en hash av alle underliggende data, og endres for hver gang noen av de underliggende dataene endres.
    * Dette gjør den sensitiv til dataendring.

7.3.1 Code

```
/**
 *  Container-klassen Hasing.
 *
 *  Inneholder en char-array ('tabell'), max. antall elementer i
 *  arrayen ('lengde'), samt hvilken type tabellen er av ('hType').
 */
class  Hashing {
  private:
    char* array;        ///< Pointer to the character array used for hashing
    int   length;       ///< Maximum number of elements in the array
    HashType hType;     ///< Type of the hashing technique used

    // Methods
    int hash1(const int modValue, const int kValue);
    int hash2(const int hashValue, const int kValue);
    int kValue(char character);

  public:
    // Methods
    Hashing(const HashType hT, const int len);
    ~Hashing();
    void display() const;
    void insert(const int hashVerdi, const char data);
};

/**
 * Simple hash function.
 *
 * @param modValue The modulus value for the hash calculation.
 * @param kValue The key value to be hashed.
 * @return The hash value computed using modValue and kValue.
 */
int Hashing::hash1(const int modValue, const int kValue) {
    return (kValue % modValue);
}

/**
 * Secondary hash function.
 *
 * @param hashValue The initial hash value.
 * @param kValue The key value to be hashed.
 * @return The secondary hash value computed.
 */
int Hashing::hash2(const int hashValue, const int kValue) {
    return (hashValue - (kValue % hashValue));
}

/**
 * Computes the key value from a character.
 *
 * @param character The character to be converted into a key value.
 * @return The key value corresponding to the character.
 */
int Hashing::kValue(char character) {
    character = toupper(character);
    if (character >= 'A'  &&  character <= 'Z') {
        return (static_cast <int> (character - 'A') + 1);
    } else {
        return 0;
    }
}

/**
 * Constructor for the Hashing class.
 *
```

```
 * @param hT The type of hash to be used.
 * @param len The length of the hash table.
 */
Hashing::Hashing(const HashType hT, const int len) {
    length = len;
    hType = hT;
    array = new char[len];
    for (int i = 0; i < length; i++) {
        array[i] = '-';
    }
}

/**
 * Destructor for the Hashing class.
 */
Hashing::~Hashing() {
    delete []  array;
}

/**
 * Displays the contents of the hash table.
 */
void Hashing::display() const {
    for (int i = 0; i < length; i++) {
        cout << setw(3) << i << ':';
    }
    cout << '\n';

    for (int i = 0; i < length; i++) {
        cout << "  " << array[i] << ' ';
    }
    cout << "\n\n";
}

/**
 * Inserts a character into the hash table.
 *
 * @param hashValue The initial hash value to consider for insertion.
 * @param data The character data to be inserted.
 */
void Hashing::insert(const int hashValue, const char data) {
    int dataToKVal = kValue(data);
    int index   = hash1(length, dataToKVal);
    int addition  = hash2(hashValue, dataToKVal);

    while (array[index] != '-') {
        index = (hType == LinearProbing) ? (index+1) : (index+addition);
        index %= length;
    }
    array[index] = data;
    display();
}
```

8 Graphs

Most of the code will refer to the fringe class - see final entry in section for the fringe.h file.

8.1 Minimum Spanning Tree (MST) - Prim

8.1.1 Algoritme

1. Noden er enten
   (a) I det hittil oppbygde MST.
   (b) På Fringen.
   (c) Usett.
2. Finner man en node som allerede er på Fringen:
   (a) Sjekk om den skal oppdateres med enda lavere vekt.
3. $gTilknyttning[k]$ er noden som sørget for at noden k:
   (a) Ble flyttet fra Fringen til grafen.
   (b) Fikk sin minimale verdi (vekt) på Fringen hittil.
4. $gKantvekt[k]$ er vekten på kanten mellom k og $gTilknyttning[k]$.
5. Noder på Fringen er (i $gKantvekt$) markert med negativ vekt (USETT = -999).

8.1.2 Framgangsmåte

1. Lag en prioritets-kø hvor nodene er rangert med hensyn på vekten til kanten som forbinder den med det allerede bygde MST.
2. Start i en node (startnode).
3. Så lenge MST ikke inneholder alle noder:
   (a) Se på alle kanter som knytter en node til MST.
   (b) Velg den kanten med lavest vekt (evt første noden i prioritetskøen).
   (c) Legg til denne kanten & noden til MST.
4. Gjenta steg 3 til alle noder er en del av MST.

8.1.3 Orden

$(E + V) \cdot \log V$

8.1.4 Code:

Note: The whole example is included here to give enough context for the example code.

```
#include <iostream>          //  cout
#include <iomanip>           //  setw
#include "fringe.h"          //  Datatypen/-strukturen "Fringe"
using namespace std;
const int NUMNODES =     8;                  ///<  Antall noder i grafen (V).
const int UNSEEN    =  -999;                  ///<  IKKE sett - "stort" negativt tall.
Fringe gFringe;                               ///<  Lager Fringe.
int    gEdgeWeight[NUMNODES+1];               ///<  'gKantVekt[k]' er kantens
int    gConnection[NUMNODES+1];               ///<   vekt mellom 'k' og dens
                                              ///<   'gTilknytning[k]'.
void findMST(int num);
void print();

int  gNeighbourMatrix[NUMNODES+1][NUMNODES+1]  ///<  Nabomatrise for grafen:
   = { { 0, 0, 0, 0, 0, 0, 0, 0, 0 },       //    NB:  Bruker IKKE indeks 0 !!!
       { 0, 0, 3, 2, 0, 0, 0, 1, 0 },       // A = 1     B -------- F -------- H
       { 0, 3, 0, 0, 0, 0, 2, 0, 0 },       // B = 2     |     2    |   3    / |
       { 0, 2, 0, 0, 0, 1, 0, 3, 0 },       // C = 3    3|          2|       /  |
       { 0, 0, 0, 0, 0, 2, 0, 3, 1 },       // D = 4     |     1    |     2/   |
       { 0, 0, 0, 1, 2, 0, 0, 1, 0 },       // E = 5     A -------- G------    | 1
       { 0, 0, 2, 0, 0, 0, 0, 2, 3 },       // F = 6     |         / | \        |
       { 0, 1, 0, 3, 3, 1, 2, 0, 2 },       // G = 7    2|  ------  |  ------  |
       { 0, 0, 0, 0, 1, 0, 3, 2, 0 } };     // H = 8     | /  3    1|    3   \ |
//          A  B  C  D  E  F  G  H                   C -------- E -------- D
//                                                      1          2


/**
 *  Hovedprogrammet:
 */
int main() {
  for (int i = 1;  i <= NUMNODES;  i++) {
      gEdgeWeight[i] = UNSEEN;
  }
  findMST(1);
  cout << "\n\n";
  return 0;
}

/**
 *  Finner ETT minimums spenntre for en sammenhengende graf/komponent.
 *
 *  @param   num  -  Grafens startnode, som inni funksjonen brukes/oppdateres
 *                   til å være aktuell besøkt node
 */
void findMST(int num) {
    if (gFringe.update(num, -UNSEEN)) {
       gConnection[num] = 0;
    }

    while (!gFringe.empty())  {
        cout << "\n\nOPPSTART:";
        print();
        num = gFringe.remove();
        gEdgeWeight[num] = -gEdgeWeight[num];
        cout << "\nNr.1 (" << char ('A'+num-1) << ") fjernet:";
        print();

        if (gEdgeWeight[num] == -UNSEEN) {
            gEdgeWeight[num] = 0;
        }

        for (int j = 1;  j <= NUMNODES;  j++) {
            const int weight = gNeighbourMatrix[num][j];
```

```
65
66              if (weight > 0  && gEdgeWeight[j] < 0)  {
67
68                  if (gFringe.update(j, weight) ) {
69                      gEdgeWeight[j] = -weight;
70                      gConnection[j] = num;
71                      cout << "\nOppdatering:";
72                      print();
73                  }
74              }
75          }
76      }
77  }
78
79
80  /**
81   *  Skriver ut på skjermen fringen og alle globale variable (arrayer).
82   */
83  void print()  {
84      gFringe.display(Character);
85
86      cout << "\n\t\t\t";
87      for (int i = 1;  i <= NUMNODES;  i++) {
88          cout << setw(5) << char('A'+i-1) << ':';
89      }
90
91      cout << "\n\tgKantVekt:     ";
92      for (int i = 1;  i <= NUMNODES;  i++) {
93        cout << setw(6) << gEdgeWeight[i];
94      }
95
96      cout << "\n\tgTilknytning:  ";
97      for (int i = 1;  i <= NUMNODES;  i++) {
98          cout << setw(6) << ((gConnection[i] > 0) ? char('A'+gConnection[i]-1) : '0');
99      }
100 }
```

8.2 Shortest Path - Dijkstra

8.2.1 Framgangsmåte

1. Start med et sett med ubesøkte noder.
   (a) Initieelt vil dette inneholde alle nodene.
2. Så lenge det er ubesøkte noder:
   (a) Velg den noden fra ubesøkte noder med lavest distanse fra startnoden.
   (b) For denne noden, se alle nabo-noder.
   (c) Kalkuler distansen til nabo-nodene med distansen/vekten fra start til aktuell node pluss vekten på kanten fra aktuell node til nabo-node.
   (d) Hvis den nye distansen er mindre enn nabo-nodens tidligere kalkulerte distanse, oppdater distansen til denne noden, og hvor hvilken node den kom fra.
   (e) Etter alle nabo-noder til den aktuelle noden har blitt sett, marker den aktuelle noden som besøkt. Denne noden vil ikke bli sjekket igjen.
3. Gjenta til alle noder har blitt besøkt.
4. Start i "slutt" - noden og konstruer stien tilbake til starten ved å se på hvor hver node kommer fra. Vekten på stien/lengden fra start til slutt, er summen av vekten på alle kantene i stien.

8.2.2 Algoritme

Algoritme/virkemåte: "Identisk" til Prim's algoritme, bare at avstanden fra node nr. 'i' via node nr. 'j' til node nr. 'k' er avstanden fra nr. 'j' til nr. 'k' pluss minimumsavstanden fra nr. 'i' til nr. 'j'.

```
// ************************************************************
// **   CODE IS ALMOST IDENTICAL TO MST PRIMS ALGORITHM,     **
// **   JUST INSERT THE BELOW CODE BEFORE THE TWO FINAL IF   **
// **   BLOCKS IN THE FUNCTION (example below):              **
// **            weight += gEdgeWeight[num];                 **
// **                                                        **
// ************************************************************

/* ... */
    weight += gEdgeWeight[num]; // <-- ADD THIS LINE
    if (weight > 0  && gEdgeWeight[j] < 0)  {

        if (gFringe.update(j, weight) ) {
            gEdgeWeight[j] = -weight;
            gConnection[j] = num;
            cout << "\nOppdatering:";
            print();
        }
    }
/* ... */
```

8.3 A Star

Algoritme for å effektivt finne korteste/raskeste vei fra en startnode til en sluttnode. A* = Dijkstra + Heuristikk

8.3.1 Heuristikk

Omhandler å estimere veien/avstanden fra nåværende node/rute og til målet. Avstanden til målet (f) er derfor summen av reell avlagt avstand fra startnoden, pluss estimert avstand til målet.

8.3.2 Orden

Tidskompleksiteten avhenger av den heuristikke funksjonen.

- Beste fall: $b \cdot d$
- Gjennomsnitt/verste fall: b^d

8.3.3 Framgangsmåte

1. Initialiser
   - Lag to sett/mengder: 'Open Set' og 'Closed Set'.
   - Legg til startnoden til 'Open Set'.
2. Definer kostnad-funksjonene
   - $g(n)$: Kostnaden fra startnoden til node n.
   - $h(n)$: Den heuristiske distansen fra node n til målet.
   - $f(n)$: Estimert totalkostnaden til node n fra start til mål, $g(n) + h(n)$.
3. Hovedfunksjon
   - Så lenge 'Open Set' ikke er tomt:
   - Velg den noden i Open Set med lavest $f(n)$-verdi. Kall denne noden 'current'.
   - Hvis 'current' er målet, så er du ferdig.
   - Hvis ikke, flytt 'current' fra 'Open Set' til 'Closed Set'.
4. For hver nabo av 'current':
   - Hvis 'nabo' er i 'Closed Set', ignorer.
   - Kalkuler $g('nabo')$ med en sti gjennom 'current'.
   - Hvis 'nabo' ikke er i 'Open Set', legg den til.
   - Hvis 'nabo' er i 'Open Set', sjekk om $g('nabo')$ er større enn den nodens tidligere $g(n)$ verdi.
   - Hvis den er lavere, oppdater $g(n)$ for noden.
   - Hvis den er større, ignorer denne stien.
5. Sti rekonstruksjon:
   - Når du har kommet til mål, konstruer stien baklengs fra mål til starten ved å gå til hver node sin mor.

8.3.4 Code - A star:

Note: The whole example is included here to give enough context for the example code.

```
#include <iostream>
#include <cmath>
#include "fringe.h"
using namespace std;

// Variables
const int DIMENSION = 20;
const int ARRLEN = DIMENSION * 101;
const int UNSEEN = -999;
const int TARGET_SQUARE = 1814;
const int START_SQUARE = 208;
int gEdgeWeight[ARRLEN + 1];
int gAssociation[ARRLEN + 1];
Fringe gFringe(DIMENSION * DIMENSION);

// Functions
bool AStar();
int heuristics(int nr);
void createAndPrintRoute();

// Gameboard
char gGameBoard[DIMENSION + 2][DIMENSION + 3]
//      01234567890123456789001
   = { "XXXXXXXXXXXXXXXXXXXXX",
       "X        XXXXXX      X",
       "X             X      X",
       "X   X                X",
       "X   X           X    X",
       "X   X           X    X",
       "X   XXXXXXX     XXX  X",
       "X      X        X    X",
       "X     XX        X    X",
       "X     X  XXXXXXXX    X",
       "X                    X",
       "X   XX XXXXXXXXXX    X",
       "X   X         X      X",
       "X   X       XXX      X",
       "X             X      X",
       "X          XXXX      X",
       "XXXXXXXXXX           X",
       "X          XXXXXX    X",
       "X   XXXX        X    X",
       "X   XXXX        XXX  X",
       "X                    X",
       "XXXXXXXXXXXXXXXXXXXXX" };


/**
 *  The main program.
 */
int main() {
    for (int i = 101; i <= (ARRLEN); i++) {
        gEdgeWeight[i] = UNSEEN;
    }

    if (AStar()) {
        createAndPrintRoute();
    } else {
        cout << "\n\n\tUnable to find a path from " << START_SQUARE
             << " to " << TARGET_SQUARE << "\n\n";
    }
    return 0;
}

```

```
/**
 *  Tries efficiently and quickly to find a short path between two squares in a grid.
 *  Utilizes the A* pathfinding algorithm, combining Dijkstra's algorithm and a heuristic.
 *  The algorithm is often used in games and artificial intelligence for grid-based pathfinding.
 *
 *  @return   Whether a path was found from START_SQUARE to TARGET_SQUARE or not.
 *  @see      heuristics(...)
 */
bool AStar() {
   int i,
       nr = START_SQUARE,
       neighbor,
       x,
       y,
       weight,
       additional;

   gFringe.update(nr, -UNSEEN);
   gAssociation[nr] = 0;
   gEdgeWeight[nr] = 0;

   while (!gFringe.empty()) {
       nr = gFringe.remove();
       if (nr == TARGET_SQUARE) { return true; }

       gEdgeWeight[nr] = -gEdgeWeight[nr];

       // Check all 8 potential neighbors (including diagonals)
       for (i = 1; i <= 8; i++) {
            // Determine the neighbor based on the current index
            switch (i) {
                case 1: neighbor = nr-100; additional = 2; break;  // Up
                case 2: neighbor = nr-99;  additional = 3; break;  // Diagonal Up-Right
                case 3: neighbor = nr+1;   additional = 2; break;  // Right
                case 4: neighbor = nr+101; additional = 3; break;  // Diagonal Down-Right
                case 5: neighbor = nr+100; additional = 2; break;  // Down
                case 6: neighbor = nr+99;  additional = 3; break;  // Diagonal Down-Left
                case 7: neighbor = nr-1;   additional = 2; break;  // Left
                case 8: neighbor = nr-101; additional = 3; break;  // Diagonal Up-Left
            }

            // Convert the neighbor square's number to grid coordinates
            x = neighbor % 100;  y = neighbor / 100;

            // If the neighbor is not a wall ('X') and is unvisited
            if ((gGameBoard[y][x] != 'X') && (gEdgeWeight[neighbor] < 0)) {
                weight = gEdgeWeight[nr] + additional;

                    // If this is a new path or a better one, update the fringe
                    if (gFringe.update(neighbor, weight + heuristics(neighbor))) {
                        gEdgeWeight[neighbor] = -weight;
                        gAssociation[neighbor] = nr;
                    }
            }
       }
   }
   return false;
}

/**
 *  Calculates the "straight-line distance"
 *  (the hypotenuse in a triangle, i.e., Euclidean distance) between two squares.
 *
 *  @param    nr  -  Square number/ID
 *  @return   Calculated "straight-line" from square 'nr' to TARGET_SQUARE
 */
int heuristics(int nr) {
   int dx = ((nr % 100) - (TARGET_SQUARE % 100));
```

```
   int dy = ((nr / 100) - (TARGET_SQUARE / 100));
   float straightLine = sqrt((dx * dx) + (dy * dy));
   return (2 * straightLine);
}

/**
 *  Finds the path that has been taken, "draws" it on the grid, and prints this.
 */
void createAndPrintRoute() {
    int i, j, nr;
    cout << "\n\n";

    gGameBoard[START_SQUARE / 100][START_SQUARE % 100] = 'S';
    gGameBoard[TARGET_SQUARE / 100][TARGET_SQUARE % 100] = 'M';

    nr = gAssociation[TARGET_SQUARE];
    while (gAssociation[nr] != 0) {
        gGameBoard[nr / 100][nr % 100] = '.';
        nr = gAssociation[nr];
    }

    for (i = 0; i < DIMENSION + 2; i++) {
        for (j = 0; j < DIMENSION + 2; j++)
            cout << gGameBoard[i][j];
        cout << '\n';
    }
}
```

8.4 Union Find

8.4.1 Framgangsmåte

1. Initialize the Sets
   - Start with n elements (nodes).
   - Each element is in its own set, usually represented by a tree where each element is its own root.
   - You can visualize this as a list of elements, each pointing to themselves.
2. Union Operations
   - To "union" two elements means to connect their sets.
   - If they are already in the same set, do nothing. Otherwise, choose one of the sets and link its root to the root of the other set.
3. Find Operations
   - To "find" an element means to determine which set it belongs to. This is done by following the chain of parents until you reach the root.
   - The root uniquely identifies the set.

8.4.2 Code - Union find:

```
/**
 *   Programeksempel nr 36 - Union-Find.
 *
 *   Noen ganger er spørsmålet om node A er i samme komponent / (sub)graf /
 *   set / ekvivalensklasse som node B (path'en imellom er uinteressant).
 *   NB:  Det bygges IKKE en lignende graf, men et tre/flere trær av de som
 *        er i samme komponent/subgraf.
 *
 *   Funksjonen 'unionerOgFinn' setter noder til å være i samme komponent
 *   om 'unioner' er lik 'true', dvs. en unionering skal skje.
 *   Er 'unioner' lik 'false' er det interessant hva funksjonen returnerer,
 *   dvs. om nodene allerede befinner seg i samme komponent eller ei.
 *
 *
 *   Algoritme/virkemåte:
 *     gForeldre[i] > 0 (=x) når node nr.'i' har 'x' som foreldre/mor
 *     gForeldre[i] = 0      når node nr.'i' ennå ikke har noen foreldre,
 *                           eller ender opp som rot for et tre
 *     unioner = true      om noder skal knyttes sammen
 *             = false     om det skal finnes ut om noder er i samme komponent
 *
 *   @file     EKS_36_UnionFind.CPP
 *   @author   Frode Haug, NTNU
 */


#include <iostream>
#include <iomanip>                //  setw
using namespace std;
const int NUMNODES  = 10;         ///<  Nodene har 'ID' lik  'A'-'J'  (1-10).
const int NUMEDGES = 14;          ///<  Antall kanter i grafen.
int gParents[NUMNODES+1];         ///<  I "skuff" nr.i er foreldre til nr.i.
                                  ///<  Grafkantene:
char gEdges[NUMEDGES][3] = { "AB", "CG", "JI", "AJ", "BD", "HB", "DC",
                             "DE", "GE", "FE", "CH", "IH", "BJ", "BC" };

/*  Grafen ser slik ut:      A------------B---------D
                             |          / | \     /   \
                             |        /   |  \  /     \
                             |      /     |    C       E----F
                             |    /       |   / \     /
                             |  /         |  /   \   /
                             |/           | /     \ /
                             J------I-----H        G
*/

void skriv();
bool unionFind(int nr1, int nr2, const bool unioned);

/**
 *  Hovedprogrammet:
 */
int main() {
    int nr1 = 0, nr2 = 0;
    char tegn = ' ';

    for (int i = 0; i < NUMEDGES; i++) {
        nr1 = static_cast <int> (gEdges[i][0]-'A'+1);
        nr2 = static_cast <int> (gEdges[i][1]-'A'+1);
        unionFind(nr1, nr2, true);
        cout << '\n' << gEdges[i][0] << ' ' << gEdges[i][1] << ':';
        skriv();
        cout << "\t\t'D' og 'H' er " << (!unionFind(4, 8, false) ? "IKKE " : "")
             << "i samme komponent";
    }
```

```
66      cout << "\n\n";
67      return 0;
68  }
69
70
71  /**
72   *  Skriver 'gForeldres' innhold som bokstaver og evt. antall barn i subtre.
73   */
74  void skriv() {
75      cout << '\t';
76      for (int i = 1; i <= NUMNODES; i++) {
77          cout << "  " << static_cast <char> (i+'A'-1);
78      }
79
80      cout << "\n\t";
81      for (int i = 1; i <= NUMNODES; i++) {
82          if (gParents[i] > 0) {
83              cout << "  " << static_cast <char> (gParents[i]+'A'-1);
84          } else if (gParents[i] == 0) {
85              cout << "  -";
86          } else {
87              cout << setw(3) << (-gParents[i]);
88          }
89      }
90  }
91
92  /**
93   * If 'union' is 'true', node 'nr1' and 'nr2' will be set to be
94   * in the same component, or it will return if they already ARE in the same component.
95   *
96   * @param nodeIndex1 - Index for 1st node (becomes parent of or child to 'nr2')
97   * @param nodeIndex2 - Index for 2nd node (becomes parent of or child to 'nr1')
98   * @param unioned    - Should the nodes end up in the same component or not
99   * @return Are 'nr1' and 'nr2' in the same component or not?
100  */
101 bool unionFind(int nodeIndex1, int nodeIndex2, const bool unioned) {
102     int i = nodeIndex1,
103         j = nodeIndex2;
104     while (gParents[i] > 0)  { i = gParents[i]; }
105     while (gParents[j] > 0)  { j = gParents[j]; }
106     if (unioned && (i != j)) { gParents[j] = i; }
107     return (i == j);
108 }
```

8.4.3 Union Find with Weight Balancing and Path Compression

```
/**
 * If 'unioned' is 'true', nodes 'nodeIndex1' and 'nodeIndex2' will be set to be
 * in the same component, otherwise it returns whether they already are in the same component.
 *
 * Identical to 'unionFind(...)' in previous example except that
 * the code also includes Path Compression (PC) and Weight Balancing (WB).
 *
 * @param nodeIndex1 - Index for the 1st node (becomes parent of or child to 'nodeIndex2')
 * @param nodeIndex2 - Index for the 2nd node (becomes parent of or child to 'nodeIndex1')
 * @param unioned    - Should the nodes end up in the same component or not
 * @return Are 'nodeIndex1' and 'nodeIndex2' in the same component or not?
 */
bool unionFind(int nodeIndex1, int nodeIndex2, const bool unioned) {
    int i = nodeIndex1,
        j = nodeIndex2;
    while (gParents[i] > 0)  { i = gParents[i]; }
    while (gParents[j] > 0)  { j = gParents[j]; }

    // ********************************************************************************
    // NEW (down to star line) compared to regular union find. Path Compression (PC):
    int index = 0;
    while (gParents[nodeIndex1] > 0) {
        index = nodeIndex1;
        nodeIndex1 = gParents[nodeIndex1];
        gParents[index] = i;
    }

    while (gParents[nodeIndex2] > 0) {
        index = nodeIndex2;
        nodeIndex2 = gParents[nodeIndex2];
        gParents[index] = j;
    }

    // ********************************************************************************
    // NEW (down to star line) compared to regular union find. Weight Balancing (WB):
    if (unioned  &&  (i != j)) {
        if (gParents[j] < gParents[i])  {
             gParents[j] += gParents[i]-1;
             gParents[i] = j;
        } else  {
            gParents[i] += gParents[j]-1;
            gParents[j] = i;
        }
    }
    //  ********************************************************************************

    return (i == j);
}
```

8.5 fringe.h

```
#ifndef __FRINGE_H            //  For at evt. bare skal includes EN gang:
#define __FRINGE_H

/**
 * Enum to specify if the Fringe's key should be printed as a character or a number.
 */
enum PrintType { Character, Number};

/**
 * Class representing a Fringe, which contains the number of elements in each array
 * and two arrays for storing the key/ID/data of a node and the lowest weight
 * of the edge leading to this node found so far.
 */
class Fringe  {
  private:
    int* keys;            ///< 'keys[i]' is the nodes so far encountered.
    int* weights;         ///< 'weight[i]' is the weight for connecting the node at 'keys[i]'.
    int  indexAmount;     ///< Last index used in both 'keys' and 'weights'.
                          ///<     incremented for every node encountered
  public:
    Fringe(const int max = 200);
    ~Fringe();
    void display(const PrintType type) const;
    bool empty() const;
    bool update(const int key, const int weight);
    int remove();
};

/**
 * Constructor for Fringe class. Initializes arrays to store keys and weights.
 * @param max The maximum size of the arrays.
 */
Fringe::Fringe(const int max) {
    keys = new int[max];
    weights = new int[max];
    indexAmount = 0;
}

/**
 * Destructor for Fringe class. Deallocates memory used for keys and weights arrays.
 */
Fringe::~Fringe() {
    delete [] keys;
    delete [] weights;
}

/**
 * Displays the contents of the Fringe.
 * @param type The PrintType specifying how to display keys (as characters or numbers).
 */
void Fringe::display(const PrintType type) const  {
    std::cout << "\tFringe:\t";
    for (int i = 0;  i < indexAmount;  i++)  {
        if (type == Character) {
            std::cout << char(keys[i]+'A'-1);
        } else {
            std::cout <<  keys[i];
        }
        std::cout << ':' << weights[i] << "  ";
    }
}

/**
 * Checks if the Fringe is empty.
 * @return True if empty, otherwise false.
```

```
 */
bool Fringe::empty() const {
    return  (indexAmount == 0);
}

/**
 * Updates the Fringe with a new node and its weight. Adds the node if it's new,
 * or updates the weight if it's lower than the existing one.
 * IMPORTANT: The node is added before any existing nodes with the same weight.
 *
 * @param key     - The key/ID/data of the node to be added.
 * @param weight  - The weight of the edge to this node.
 * @return        - True if the node was added or updated, otherwise false.
 */
bool Fringe::update(const int key, const int weight) {
    int j;
    int i = 0;

    // Search for the node's key in the existing keys.
    // If found, check if the new weight is not lower than the existing weight.
    while ((keys[i] != key)  &&  (i < indexAmount)) {
        i++;
    }

    // If the node exists and the new weight is not lower, do not update.
    if ((i < indexAmount)  &&  (weight >= weights[i])) {
        return false;
    }

    // If the node exists and the new weight is lower, remove the old node.
    if (i < indexAmount)  {
        for (j = i;  j < indexAmount-1;  j++) {
            keys[j] = keys[j+1];
            weights[j] = weights[j+1];
        }
        --indexAmount;
    }

    // Find the position where the new or updated node should be inserted.
    i = 0;
    while ((weights[i] < weight)  &&  (i < indexAmount)) {
        i++;
    }

    // Increase the size of the Fringe to accommodate the new node.
    ++indexAmount;

    // Shift existing nodes to make space for the new node.
    for (j = indexAmount-1; j > i; j--) {
        keys[j] = keys[j-1];
        weights[j] = weights[j-1];
    }

    // Insert the new or updated node.
    keys[i] = key;
    weights[i] = weight;
    return true;
}

/**
 * Removes and returns the first element in the Fringe.
 * @return     - The key of the first element.
 */
int Fringe::remove()  {
    int key = keys[0];
    for (int i = 0;  i < indexAmount-1;  i++)  {
        keys[i] = keys[i+1];
        weights[i] = weights[i+1];
```

```
    }
    --indexAmount;
    return key;
}

#endif
```

Huffman

Framgangsmåte

1. Samle data og frekvens
   - Start med en streng, del denne opp i tegn og tell hvor ofte hvert tegn forekommer
2. Lag bladnoder
   - Lag en bladnode for hvert tegn
   - hver bladnode inneholder tegnet og frekvensen
3. Lag en prioritetskø
   - Plasser alle nodene i en prioritets-kø, hvor den noden med lavest frekvens er fremst i køen, og den med høyest frekvens er bakerst i køen.
4. Konstruer Huffman treet.
   - Så lenge det er mer enn en node i køen:
     - Fjern de to nodene med lavest frekvens fra køen
     - Lag en ny "intern" node med disse to nodene som barn, og med frekvensen lik summen av frekvensen til barna
     - Plasser den nye noden på prioritets-køen.
   - Den siste noden på prioritets-køen er roten til huffman treet
5. Generer Kode
   - Traverser treet fra Rot-noden til hver Blad-node og skriv ned "koden" til hvert tegn
     - for hver venstre-kant $\rightarrow$ legg til en 0
     - for hver høyre-kant $\rightarrow$ legg til en 1
6. Encode dataen
   - Oversett hvert tegn i strengen med den nye koden fra huffman-kodingen for å få den nye dataen.
7. Decode dataen
   - følg stiene i huffman treet for å finne strengen.
     - hver 0 $\rightarrow$ gå til venstre barn
     - hver 1 $\rightarrow$ gå til høyre barn
     - hver bladnode $\rightarrow$ skriv tegnet.

9 Datastructures

9.1 Heap

9.1.1 VIKTIG:

Hvis ingenting annet er nevnt, anta at det er snakk om en maxheap - IKKE en minheap!

9.1.2 Beskrivelse

En binary heap kan skrives som et komplett binært tre, hvor hver node sin verdi er mindre eller lik barna sine. Siden heapen skal være som et komplett binært tre, legger vi hele tiden på neste node på laveste nivå i treet, fra venstre mot høyre. Siden treet skal være komplett, kan vi implementere heapen som et array, og bruke følgende metoder for å finne barna og foreldre nodene til en gitt node:

```
leftChild:  2 * index +1
rightChild: 2 * index +2
parent:     (index - 1) / 2
    NB: Vi runder hele tiden ned for parent-node
    hvor index = indexen til en gitt node i arrayet
```

9.1.3 UpHeap

Når vi legger til et element i Heapen, legger vi det hele tiden på neste ledige plass (som beskrevet ovenfor), deretter sammenligner vi verdien til den nye noden, med verdien til foreldre noden. Hvis verdien er mindre enn foreldre-noden, bytter nodene plass. Gjenta til noden er på korrekt plass.

9.1.4 DownHeap

Når vi erstatter verdien i rot noden med en ny verdi, er det ikke sikker at den nye verdien / noden er på riktig plass, derfor kaller vi DownHeap. Downheap sjekker om verdien til noden, er større enn sine barn, hvis noden er større ett av sine barn, bytter vi de to nodene (hvis noden er større en begge sine barn, bytter vi den med den av sine barn som har lavest verdi). Gjenta dette til noden er på sin korrekte plass (den er ikke mindre enn noen av sine barn).

9.1.5 Eksempel

index	0	1	2	3	4	5	6	7	8	9
verdi	2	4	8	9	7	10	9	15	20	13

```
Barna til index 2 ('node' 8):
    2 * 2 + 1 = 5
    2 * 2 + 2 = 6

Forelder til index 2 ('node' 8):
    (2 - 1) / 2 = 1/2 -> index 0

Barna til index 4 ('node' 7):
    2 * 4 + 1 = 9
    2 * 4 + 2 = 10

Forelder til index 4 ('node' 7):
    (4 - 1) / 2 = 3/2 -> index 1
```

9.1.6 Code - Heap class:

```
/**
 *   Programeksempel nr 25 - Heap (prioritetskø) - selvlaget enkel klasse.
 *
 *   Eksemplet viser en selvlaget implementasjon av container-klassen Heap.
 *   Det er laget kode for følgende funksjoner:
 *
 *     - Heap(const int lengde = 200)
 *     - ~Heap
 *     - void change(const int keyNr, const T nyVerdi)      //  Oppgave nr.15
 *     - void display()
 *     - void downHeap(T arr[], const int ant, int keyNr)
 *     - void extract(const int keyNr)                      //  Oppgave nr.15
 *     - void insert(const T verdi)
 *     - T    remove()
 *     - T    replace(const T verdi)
 *     - void upHeap(int keyNr)
 *
 *   For mer forklaring av prinsipper og koden, se:  Heap.pdf
 *
 *   Orden ( O(...)):
 *        Alle operasjonene (insert, remove, replace, upHeap, downHeap,
 *        change og extract) krever færre enn  2 lg N  sammenligninger
 *        når utført på en heap med N elementer.
 *
 *   NB:  - Det er bare funksjonene 'insert' og 'remove' som er laget
 *          litt robuste. De andre ('change', 'extraxt' og 'replace')
 *          er IKKE det.
 *        - Heapen i denne koden fungerer ut fra at det er STØRSTE element
 *          som skal være i element nr.1. Men, koden kunne enkelt ha vært
 *          omskrevet slik at den i stedet fungerer for MINSTE element.
 *        - Er en .h-fil, da skal includes og brukes av EKS_26_HeapSort.CPP
 *
 *   @file     EKS_25_Heap.H
 *   @author   Frode Haug, NTNU
 */


#include <iostream>          //  cout
#include <limits>            //  numeric_limits::max


/**
 *  Container-klassen Heap.
 *
 *  Inneholder en array ('data') av typen 'T', to int'er som angir heapens
 *  max.lengde og nåværende antall elementer i arrayen, samt en sentinel key.
 */
template <typename T>
class Heap {
  private:
    int lengde,antall;
    T sentinelKey;
    T* data;
    void upHeap(int keyNr);

  public:
    Heap(const int len = 200); // constructor
    ~Heap();                   // destructor
    void change(const int keyNr, const T nyVerdi);
    void display();
    void downHeap(T arr[], const int ant, int keyNr);
    void extract(const int keyNr);
    void insert(const T verdi);
    T remove();
    T replace(const T verdi);
```

```
};

/**
 * @brief Constructs a heap.
 *
 * Allocates memory for the heap's array with a default length of 200.
 * Initializes the length of the array, the number of elements, and
 * sets the sentinel key to the maximum value for type T.
 *
 * @param len The initial size of the heap. Defaults to 200.
 */
template <typename T>
Heap(const int len = 200) {
    data = new T[len];  lengde = len;  antall = 0;
    sentinelKey = std::numeric_limits<T>::max();
}

/**
 * @brief Destructs the heap.
 *
 * Deallocates the memory used for the heap's array.
 */
template <typename T>
~Heap() {
    delete [] data;
}

/**
 * @brief Adjusts the position of a value in the heap to maintain heap property.
 *
 * This is a private helper method used during insertion and change operations.
 * It moves the value at the given index upwards in the heap until the heap
 * property is restored.
 *
 * @param keyNr The index of the value to move up.
 */
template <typename T>
void Heap<T>::upHeap(int keyNr) {
    T verdi = data[keyNr];
    data[0] = sentinelKey;
    while (data[keyNr/2] < verdi) {
        data[keyNr] = data[keyNr/2];
        keyNr = keyNr/2;
    }
    data[keyNr] = verdi;
}

/**
 * @brief Changes the value of an element in the heap and re-heapifies.
 *
 * @param keyNr The index of the element to change.
 * @param nyVerdi The new value to assign to the element.
 */
template <typename T>
void Heap<T>::change(const int keyNr, const T nyVerdi) {
    // LAG INNMATEN ifm. Oppgave nr.15
}

/**
 * @brief Displays the contents of the heap.
 *
 * Prints all elements of the heap in their current order to standard output.
 */
template <typename T>
void Heap<T>::display() const {
    for (int i = 1; i <= antall; i++)  std::cout << ' ' << data[i];
}

```

```cpp
/**
 * @brief Adjusts the position of a value in the heap to maintain heap property.
 *
 * This method is used during deletion operations. It moves the value at the given
 * index downwards in the heap until the heap property is restored.
 *
 * @param arr The heap array.
 * @param ant The number of elements in the heap.
 * @param keyNr The index of the value to move down.
 */
template <typename T>
void Heap<T>::downHeap(T arr[], const int ant, int keyNr) {
    int j;
    T verdi = arr[keyNr];
    while (keyNr <= ant/2) {
        j = 2 * keyNr;
        if (j < ant  &&  arr[j] < arr[j+1]) { j++; }
        if (verdi >= arr[j]) { break; }
        arr[keyNr] = arr[j];
        keyNr = j;
    }
    arr[keyNr] = verdi;
}

/**
 * @brief Extracts a value from the heap.
 *
 * The method for extracting values from the heap. Intended to be
 * implemented in relation to specific task requirements.
 *
 * @param keyNr The index of the value to extract.
 */
template <typename T>
void Heap<T>::extract(const int keyNr) {
    //  LAG IMNMATEN  ifm.  Oppgave nr.15
}

/**
 * @brief Inserts a new value into the heap.
 *
 * Adds a new value to the heap and then re-heapifies to maintain the heap property.
 * If the heap is full, displays an error message.
 *
 * @param verdi The value to insert.
 */
template <typename T>
void Heap<T>::insert(const T verdi) {
    if (antall < lengde - 1) {
        data[++antall] = verdi;
        upHeap(antall);
    } else {
        std::cout << "\nHeapen er full med " << lengde << " elementer (inkl. sentinel key)!\n\n";
    }
}

/**
 * @brief Removes and returns the top element of the heap.
 *
 * Removes the top element (maximum or minimum based on heap type), re-heapifies, and returns
 * the removed element. If the heap is empty, displays an error message and returns the sentinel key.
 *
 * @return The removed top element of the heap.
 */
template <typename T>
T Heap<T>::remove() {
    if (antall > 0) {
        T verdi = data[1];
        data[1] = data[antall--];
```

```
        downHeap(data, antall, 1);
        return verdi;
    }  else {
        std::cout << "\nHeapen er helt tom - ingenting i 'remove'!\n\n";
        return sentinelKey;
    }
}

/**
 * @brief Replaces the top element of the heap with a new value.
 *
 * Replaces the top element of the heap with a given value and then re-heapifies.
 * This operation combines removal and insertion into a single step.
 *
 * @param verdi The value to replace the top element with.
 * @return The replaced top element of the heap.
 */
template <typename T>
T Heap<T>::replace(const T verdi) {
    data[0] = verdi;
    downHeap(data, antall, 0);
    return data[0];
}
```

9.2 Binary Search Tree

9.2.1 Code:

```
/**
 *  Container class Binary Search Tree (BST)
 *
 *  Contains a binary search tree consisting of 'Node' structs,
 *  and has a "dummy" 'head' node which has 'data' (ID/key) smaller than everything else
 *  in the tree. 'head->right' points to the actual root of the tree !!!
 *
 */
template <typename Key, typename Value>
class BST {
  private:

      /**
       * @brief Nested struct representing a single node in the BST.
       */
      struct Node {
          Key data;
          Value value;
          Node* left, * right;

          /**
           * @brief Constructor for Node.
           * @param k Key of the node.
           * @param v Value of the node.
           */
          Node(const Key k, const Value v) {
             data = k;
             value = v;
             left = right = nullptr;
          }
      };

      Node* head;
      void traverseInOrder(Node* node) const;

  public:
```

```
37      BST() {
38        head = new Node(Key(), Value());
39      }
40      ~BST()  {    /*  delete the whole tree  */    }
41
42      void display() const;
43      void insert(const Key key, const Value value);
44      bool remove(const Key key);
45      Value search(const Key key) const;
46  };
47
48  /**
49   * @brief In-order traversal of the BST.
50   * @param node The starting node for traversal.
51   */
52  template <typename Key, typename Value>
53  void BST<Key, Value>::traverseInOrder(Node* node) const {
54      if (node) {
55          traverseInOrder(node->left);
56          cout << '\t' << node->data;
57          if (node->left) {
58              cout << "    Left child: " << node->left->data;
59          }
60          if (node->right) {
61              cout << "    Right child: " << node->right->data;
62          }
63          cout << '\n';
64          traverseInOrder(node->right);
65      }
66  }
67
68  /**
69  * @brief Displays the BST in-order.
70  */
71  template <typename Key, typename Value>
72  void BST<Key, Value>::display() const {
73      traverseInOrder(head->right);
74  }
75
76  /**
77   * @brief Inserts a new node in the BST.
78   * @param key Key of the new node.
79   * @param value Value of the new node.
80   */
81  template <typename Key, typename Value>
82  void BST<Key, Value>::insert(const Key key, const Value value) {
83      Node * parent = head,
84           * current = head->right;
85
86      if (current) {
87          while (current) {
88              parent = current;
89              current = (key < current->data) ? current->left : current->right;
90          }
91          current = new Node(key, value);
92          if (key < parent->data) {
93              parent->left = current;
94          } else {
95              parent->right = current;
96          }
97      } else {
98          head->right = new Node(key, value);
99      }
100 }
101
102 /**
103  * @brief Removes a node from the BST.
104  * @param key Key of the node to be removed.
```

```
 * @return True if removal is successful, false otherwise.
 */
template <typename Key, typename Value>
bool BST<Key, Value>::remove(const Key key) {
    Node *parentToRemove,
         *toRemove,
         *parentSuccessor,
         *successor;

    parentToRemove = head;
    toRemove = head->right;

    while (toRemove  &&  key != toRemove->data) {
        parentToRemove = toRemove;
        toRemove = (key < toRemove->data) ? toRemove->left : toRemove->right;
    }

    if (!toRemove) { return false; }
    successor = toRemove;

    if (!toRemove->right) {
        successor = successor->left;
    } else if (!toRemove->right->left) {
        successor = successor->right;
        successor->left = toRemove->left;
    } else {
        parentSuccessor = successor->right;

        while (parentSuccessor->left->left) {
            parentSuccessor = parentSuccessor->left;
        }

        successor = parentSuccessor->left;
        parentSuccessor->left = successor->right;
        successor->left = toRemove->left;
        successor->right = toRemove->right;
    }

    delete toRemove;

    if (key < parentToRemove->data) {
        parentToRemove->left = successor;
    } else {
        parentToRemove->right = successor;
    }

    return true;
}

/**
 * @brief Searches for a node by its key.
 * @param key Key of the node to be searched.
 * @return Value of the found node, or default value if not found.
 */
template <typename Key, typename Value>
Value BST<Key, Value>::search(const Key key) const {
    Node* current = head->right;
    while (current  &&  current->data != key) {
        current = (key < current->data) ? current->left : current->right;
    }

    if (current) {
        return current->value;
    }
    else {
        return head->value;
    }
}
```