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);
        }
    }
}
```

3.8 Trees - searching

3.8.1 DFS - Depth First Search

3.8.2 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.
}
```

3.8.3 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

3.9 Breadth First Search

3.9.1 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!!!
                }
    }
}
```

3.9.2 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.

3.10 2-3-4 trees

3.10.1 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.

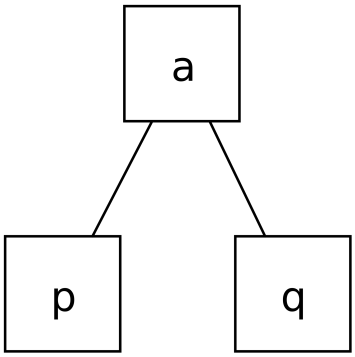


Figure 1: 2-node

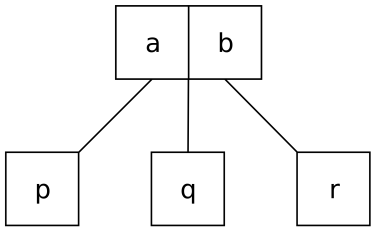


Figure 2: 3-node

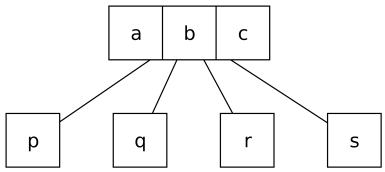


Figure 3: 4-node

3.10.2 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.

3.10.3 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.

3.11 Red-Black Trees

3.11.1 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);
}
```

8.4.4 Code results:

The following is the results of running the above example WITH weight balancing and path compression

```
A B:          A  B  C  D  E  F  G  H  I  J
            1 A  -  -  -  -  -  -  -  -                 'D' og 'H' er IKKE i samme komponent

C G:          A  B  C  D  E  F  G  H  I  J
            1 A  1  -  -  -  C  -  -  -                 'D' og 'H' er IKKE i samme komponent

J I:          A  B  C  D  E  F  G  H  I  J
            1 A  1  -  -  -  C  -  J  1                 'D' og 'H' er IKKE i samme komponent

A J:          A  B  C  D  E  F  G  H  I  J
            3 A  1  -  -  -  C  -  J  A                 'D' og 'H' er IKKE i samme komponent

B D:          A  B  C  D  E  F  G  H  I  J
            4 A  1  A  -  -  C  -  J  A                 'D' og 'H' er IKKE i samme komponent

H B:          A  B  C  D  E  F  G  H  I  J
            5 A  1  A  -  -  C  A  J  A                 'D' og 'H' er i samme komponent

D C:          A  B  C  D  E  F  G  H  I  J
            7 A  A  A  -  -  C  A  J  A                 'D' og 'H' er i samme komponent

D E:          A  B  C  D  E  F  G  H  I  J
            8 A  A  A  A  -  C  A  J  A                 'D' og 'H' er i samme komponent

G E:          A  B  C  D  E  F  G  H  I  J
            8 A  A  A  A  -  A  A  J  A                 'D' og 'H' er i samme komponent

F E:          A  B  C  D  E  F  G  H  I  J
            9 A  A  A  A  A  A  A  J  A                 'D' og 'H' er i samme komponent

C H:          A  B  C  D  E  F  G  H  I  J
            9 A  A  A  A  A  A  A  J  A                 'D' og 'H' er i samme komponent

I H:          A  B  C  D  E  F  G  H  I  J
            9 A  A  A  A  A  A  A  A  A                 'D' og 'H' er i samme komponent

B J:          A  B  C  D  E  F  G  H  I  J
            9 A  A  A  A  A  A  A  A  A                 'D' og 'H' er i samme komponent

B C:          A  B  C  D  E  F  G  H  I  J
            9 A  A  A  A  A  A  A  A  A                 'D' og 'H' er i samme komponent



Process finished with exit code 0
```

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];
```

```
134         }
135         --indexAmount;
136         return key;
137     }
138 
139     #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);
```

```
66  };
67
68  /**
69   * @brief Constructs a heap.
70   *
71   * Allocates memory for the heap's array with a default length of 200.
72   * Initializes the length of the array, the number of elements, and
73   * sets the sentinel key to the maximum value for type T.
74   *
75   * @param len The initial size of the heap. Defaults to 200.
76   */
77  template <typename T>
78  Heap(const int len = 200) {
79      data = new T[len];  lengde = len;  antall = 0;
80      sentinelKey = std::numeric_limits<T>::max();
81  }
82
83  /**
84   * @brief Destructs the heap.
85   *
86   * Deallocates the memory used for the heap's array.
87   */
88  template <typename T>
89  ~Heap() {
90      delete [] data;
91  }
92
93  /**
94   * @brief Adjusts the position of a value in the heap to maintain heap property.
95   *
96   * This is a private helper method used during insertion and change operations.
97   * It moves the value at the given index upwards in the heap until the heap
98   * property is restored.
99   *
100  * @param keyNr The index of the value to move up.
101  */
102 template <typename T>
103 void Heap<T>::upHeap(int keyNr) {
104     T verdi = data[keyNr];
105     data[0] = sentinelKey;
106     while (data[keyNr/2] < verdi) {
107         data[keyNr] = data[keyNr/2];
108         keyNr = keyNr/2;
109     }
110     data[keyNr] = verdi;
111 }
112
113 /**
114  * @brief Changes the value of an element in the heap and re-heapifies.
115  *
116  * @param keyNr The index of the element to change.
117  * @param nyVerdi The new value to assign to the element.
118  */
119 template <typename T>
120 void Heap<T>::change(const int keyNr, const T nyVerdi) {
121     // LAG INNMATEN ifm. Oppgave nr.15
122 }
123
124 /**
125  * @brief Displays the contents of the heap.
126  *
127  * Prints all elements of the heap in their current order to standard output.
128  */
129 template <typename T>
130 void Heap<T>::display() const {
131     for (int i = 1; i <= antall; i++)  std::cout << ' ' << data[i];
132 }
133
```

```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:
```

```
     BST() {
       head = new Node(Key(), Value());
     }
     ~BST()  {    /*  delete the whole tree  */    }

     void display() const;
     void insert(const Key key, const Value value);
     bool remove(const Key key);
     Value search(const Key key) const;
};

/**
 * @brief In-order traversal of the BST.
 * @param node The starting node for traversal.
 */
template <typename Key, typename Value>
void BST<Key, Value>::traverseInOrder(Node* node) const {
    if (node) {
        traverseInOrder(node->left);
        cout << '\t' << node->data;
        if (node->left) {
            cout << "    Left child: " << node->left->data;
        }
        if (node->right) {
            cout << "    Right child: " << node->right->data;
        }
        cout << '\n';
        traverseInOrder(node->right);
    }
}

/**
* @brief Displays the BST in-order.
*/
template <typename Key, typename Value>
void BST<Key, Value>::display() const {
    traverseInOrder(head->right);
}

/**
 * @brief Inserts a new node in the BST.
 * @param key Key of the new node.
 * @param value Value of the new node.
 */
template <typename Key, typename Value>
void BST<Key, Value>::insert(const Key key, const Value value) {
    Node * parent = head,
         * current = head->right;

    if (current) {
        while (current) {
            parent = current;
            current = (key < current->data) ? current->left : current->right;
        }
        current = new Node(key, value);
        if (key < parent->data) {
            parent->left = current;
        } else {
            parent->right = current;
        }
    } else {
        head->right = new Node(key, value);
    }
}

/**
 * @brief Removes a node from the BST.
 * @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;
    }
}
```

10 Exams

10.1 V23

NTNU
Kunnskap for en bedre verden

Institutt for datateknologi og informatikk

Ekstra eksamensoppgave i IDATG2102 – Algoritmiske metoder

Faglig kontakt under eksamen:	**Frode Haug**
Tlf:	**950 55 636**
Eksamensdato:	**29.mars 2023**
Eksamenstid (fra-til):	**09:00-13:00 (4 timer)**
Hjelpemiddelkode/Tillatte hjelpemidler:	**F - Alle trykte og skrevne.** (kalkulator er *ikke* tillatt)
Annen informasjon:	
Målform/språk:	**Bokmål**
Antall sider (inkl. forside):	**4**

Informasjon om trykking av eksamensoppgaven

Originalen er:

1-sidig X **2-sidig** ☐

sort/hvit X **farger** ☐

Skal ha flervalgskjema ☐

Kontrollert av:

Dato Sign

Oppgave 1 (teori, 25%)

Denne oppgaven inneholder tre totalt uavhengige oppgaver fra pensum.

a) 8 Shellsort skal utføres på bokstavene «SMITTEVERN». For hver gang indre for-løkke i eksemplet med Shellsort er ferdig (dvs. rett etter: a[j] = verdi;) :
Skriv/tegn opp arrayen og skriv verdiene til 'h' (4 og 1) og 'i' underveis i sorteringen. Marker spesielt de key'ene som har vært involvert i sorteringen.

b) 12 I de følgende deloppgaver er det key'ene "S M I T T E V E R N"
(i denne rekkefølge fra venstre mot, og blanke regnes *ikke* med) som du skal bruke.
For alle deloppgavene gjelder det at den initielle heap/tre er *tom* før første innlegging ("Insert") utføres. **Skriv/tegn den *resulterende* datastruktur når key'ene legges inn i:**

1) **en heap**
2) **et binært søketre**
3) **et 2-3-4 tre**
4) **et Red-Black tre**

c) 5 **Skriv/tegn opp Merkle treet som er basert på 7 blokker.**

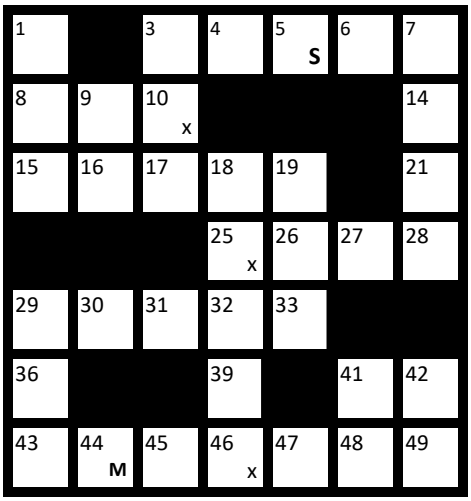
Oppgave 2 (teori, 25%)

Denne oppgaven inneholder tre totalt uavhengige oppgaver fra pensum.

a) I forbindelse med dobbelt-hashing har vi teksten «SMITTEVERNET» og de to hash-funksjonene `hash1(k) = k mod 17` og `hash2(k) = 6 - (k % 6)` der `k` står for bokstavens nummer i alfabetet (1-29). Vi har også en array med indeksene 0 til 16.
Skriv hver enkelt bokstav sin `k`-verdi og returverdi fra både `hash1` og `hash2`.
Skriv også opp arrayen hver gang en bokstav hashes inn i den.

b) Følgende kanter i en (ikke-retted, ikke-vekted) graf er gitt:
FA DB DC FC BE CA
Utfør Union-Find *m/weight balancing (WB) og path compression (PC)* på denne grafen.
Skriv/tegn opp innholdet i `gForeldre` etter hvert som `unionerOgFinn2` kjøres/utføres. Bemerk hvor WB og PC er brukt.
Skriv/tegn også opp den resulterende union-find skogen.

c) Vi har rutenettet med **S**(tart)- og **M**(ål)-ruter:



Hva vil den minste f-verdien i rutene (x) 10, 25, 46 og M kunne være
når det *kun* kan gås opp/ned/høyre/venstre (*ikke* på skrå) med en vekt på 1 (en), og som heuristikk brukes Manhatten distanse (summen av antall ruter horisontalt og vertikalt til og inkludert målet).

Oppgave 3 (koding, 26%)

Vi har et binært tre (*ikke* nødvendigvis søketre) bestående av:

```
struct Node  {
  int ID;                 //   Nodens ID/key/nøkkel/navn (et tall).
  Node *left, *right;     //   Referanse til begge subtrærne (evt. 'nullptr/NULL').
  Node(int id) { ID = id; left = right = nullptr;  }
};
```

Vi har *kun* den globale variabelen:

```
Node* gRoot = nullptr;   //   Rot-peker (har altså ikke at head->right er rota).
```

Det skal her lages/kodes to *helt* uavhengige funksjoner.
Begge funksjonene kalles initielt fra `main` med `gRoot` som parameteren `t`.
Hvordan et tre har blitt bygd/satt opp, og hvordan pekere ellers er satt, trenger du ikke å tenke på.

NB: I *hele* oppgave 3 skal det *ikke* innføres flere globale data eller `struct`-medlemmer enn angitt ovenfor. Det skal heller *ikke* brukes andre hjelpestrukturer - som f.eks. array, stakk, kø eller liste.

a) **Lag den *rekursive* funksjonen**

```
bool erSosken(const Node* t, const Node* s1, const Node* s2)
```

Funksjonen skal sjekke og returnere (`true`/`false`) om nodene tilpekt/referert av `s1` og `s2` er søsken under den samme moren/foreldret (`t`) eller ei. Hvem av dem som evt. peker til den venstre eller høyre noden er ikke definer/klart.

b) **Lag den *rekursive* funksjonen**

```
void kappTreNedentil(Node* t, const int verdi)
```

Funksjonen skal kutte forbindelsen fra `t` til alle barn som har en `ID` *større eller lik* `verdi`.
Dvs. sette aktuelle pekere i `t` til `nullptr`.
NB: At det videre nedover i de utkuttede subtrærne evt. er `ID`er som er mindre enn `verdi` tar vi ikke hensyn til. At det i C++ på denne måten egentlig oppstår en memory-lekkasje, ved at det ikke sies `delete` om alle nodene i subtrærne, tar vi heller ikke hensyn til.
Vi forutsetter at rota ikke har en verdi som gjør at den skal kuttes vekk/ut.

Oppgave 4 (koding, 24%)

a) **Lag den *ikke-rekursive* funksjonen**

```
char forsteSingleBokstav(const char tekst[], const int n)
```

som finner og returnerer den *første* bokstaven i `tekst` som *forekommer bare en gang*.
`n` er antall tegn i teksten. Det er her i oppgave 4a) *ikke* lov til å bruke noen hjelpestrukturer - som f.eks. array, stakk, kø eller liste. Funksjonen blir derfor sikkert av O (n^2).
`tekst` inneholder *kun* bokstavene A-Z (*kun* store bokstaver, og *ingen* blanke/space).
Finnes det *ingen* bokstav som forekommer *bare en gang*, returneres ‘ ‘ (blank/space).

b) **Lag en ny versjon av funksjonen i oppgave 4a)** som returnerer det samme.
Men denne gangen er det lov til å bruke *en* hjelpearray. Funksjonen bør bli av O ($^3/_2\, n$).

NB: I *hele* dette oppgavesettet skal du *ikke* bruke kode fra (standard-)biblioteker (slik som bl.a. STL). Men de vanligste includer/import du brukte i 1.klasse er tilgjengelig.
Koden kan skrives valgfritt i C++ eller Java.

Løkke tæll!
FrodeH

10.1.1 OPG 1

```
/**
 *   Løsning til ekstraeksamen i AlgMet, vår 2023, oppgave 1.
 *
 *   @file     EX_V23_1.TXT
 *   @author   Frode Haug, NTNU
 */



OPPGAVE A:
==========

       h:4  i: 5    SmitTevern                    (ingenting skjer)
       h:4  i: 6    sEittMvern                'E' vandrer forbi 'M'
       h:4  i: 7    seIttmVern                    (ingenting skjer)
       h:4  i: 8    seiEtmvTrn                'E' vandrer forbi 'T'
       h:4  i: 9    ReieSmvtTn                'R' vandrer forbi 'T' og 'S'
       h:4  i:10    reiesMvttN                    (ingenting skjer)
       h:1  i: 2    ERiesmvttn                'E' vandrer forbi 'R'
       h:1  i: 3    EIResmvttn                'I' vandrer forbi 'R'
       h:1  i: 4    EEIRsmvttn                'E' vandrer forbi 'R' og 'I'
       h:1  i: 5    eeiRSmvttn                    (ingenting skjer)
       h:1  i: 6    eeIMRSvttn                'M' vandrer forbi 'S' og 'R'
       h:1  i: 7    eeimrSVttn                    (ingenting skjer)
       h:1  i: 8    eeimrSTVtn                'T' vandrer forbi 'V'
       h:1  i: 9    eeimrsTTVn                'T' vandrer forbi 'V'
       h:1  i:10    eeiMNRSTTV                'N' vandrer forbi "RSTTV"




OPPGAVE B:
==========

       "SMITTEVERN" satt inn i:

       1)  Heap:      V T T R S E I E M N


       2)  Binært søketre:        S
                                 /   \
                                M     T
                               /   \     \
                              I     R     T
                             /     /       \
                            E     N         V
                             \
                              E


       3)  2-3-4 tre:           M  T
                               /   |   \
                              EEI  NRS   TV


       4)  Red-Black tre:         M
                                 /    \\
                                E       T          (eller MT og TV rotert andre veien)
                               // \\    /  \
                              E     I  R    T
                                      // \\   \\
                                     N    S    V

```

```
OPPGAVE C:
==========

        Merkle-tre med 7 blokker:

                        H(1234+567)
                      /              \
                 H(12+34)            H(56+7)
                /       \           /       \
           H(1+2)    H(3+4)    H(5+6)    H(7)
           /    \    /    \    /    \    /
        H(1) H(2) H(3) H(4) H(5) H(6) H(7)
        ---------------------------------
         B1   B2   B3   B4   B5   B6   B7
```

10.1.2 OPG 2

```
/**
 *   Løsning til ekstraeksamen i AlgMet, vår 2023, oppgave 2.
 *
 *   @file     EX_V23_2.TXT
 *   @author   Frode Haug, NTNU
 */



OPPGAVE A:
==========

      Keyene:           S   M   I   T   T   E   V   E   R   N   E   T
      k (alfabetnr):   19  13   9  20  20   5  22   5  18  14   5  20

      Hash1 (M = 17):   2  13   9   3   3   5   5   5   1  14   5   3
      Hash2:            5   5   3   4   4   1   2   1   6   4   1   4


      Indeks:   0   1   2   3   4   5   6   7   8   9  10   11  12  13  14  15  16
                -   -   S   -   -   -   -   -   -   -   -   -   -   -   -   -   -
                -   -   S   -   -   -   -   -   -   -   -   -   -   M   -   -   -
                -   -   S   -   -   -   -   -   -   I   -   -   -   M   -   -   -
                -   -   S   T   -   -   -   -   -   I   -   -   -   M   -   -   -
                -   -   S   T   -   -   -   T*  -   I   -   -   -   M   -   -   -
                -   -   S   T   -   E   -   T*  -   I   -   -   -   M   -   -   -
                -   -   S   T   -   E   -   T*  -   I   -   V*  -   M   -   -   -
                -   -   S   T   -   E   E*  T*  -   I   -   V*  -   M   -   -   -
                -   R   S   T   -   E   E*  T*  -   I   -   V*  -   M   -   -   -
                -   R   S   T   -   E   E*  T*  -   I   -   V*  -   M   N   -   -
                -   R   S   T   -   E   E*  T*  E*  I   -   V*  -   M   N   -   -
                -   R   S   T   -   E   E*  T*  E*  I   -   V*  -   M   N   T*  -

      (* = bokstaver som hashes på plass ved bruk av hash2 også.)




OPPGAVE B:
==========

      "gForeldre"-arrayen etterhvert:

                A  B  C  D  E  F
          F A:  F  -  -  -  -  1
          D B:  F  D  -  1  -  1
          D C:  F  D  D  2  -  1
          F C:  F  D  D  4  -  D     Weight Balancing
          B E:  F  D  D  5  D  D
          C A:  D  D  D  5  D  D     Path Compression


      Resulterende skog:      D
                             / / | \ \
                            A B  C  E F




OPPGAVE C:
==========

    10:   f =  9      ( der:  g =   3     h = 6 )
```

```
25:   f = 11      ( der:  g =   6     h = 5 )
46:   f = 11      ( der:  g =   9     h = 2 )
 M:   f = 11      ( der:  g =  11     h = 0 )
```

10.1.3 OPG 3

```
/**
 *   Løsningsforlag til ekstraeksamen i AlgMet, vår 2023, oppgave 3.
 *
 *   @file     EX_V23_3.CPP
 *   @author   Frode Haug, NTNU
 */


#include <iostream>
using namespace std;


/**
 *  Node (med ID/key, "avstand" vertikalt fra rota, og venstre/høyre subtre).
 */
struct Node {
    int ID;                     //  Nodens ID/key/nøkkel/navn (et tall).
    Node *left, *right;         //  Peker til begge subtrærne (evt. 'nullptr').
    Node(int id) { ID = id;  left = right = nullptr;  }
};


Node* gRoot = nullptr;          ///<  Peker til HELE treets rot.


/**
 *  EKSTRA - Traverserer treet under 't' rekursivt inorder.
 *
 *  @param   t  -  Noden å besøke/behandle
 */
void traverseInorder(Node* t) {
  if (t) {
      traverseInorder(t->left);
      cout << "  " << t->ID;
      traverseInorder(t->right);
  }
}


/**
 *  OPPGAVE 3A - Sjekker rekursivt om to noder er søsken (samme mor/forelder).
 *
 *  @param    t   -  Mor-/forelder-noden
 *  @param    s1  -  Node(/søsken?) nr.1
 *  @param    s2  -  Node(/søsken?) nr.2
 *  @return   Om nodene tilpekt av 's1' og 's2' er søsken under 't' eller ei
 */
bool erSosken(const Node* t, const Node* s1, const Node* s2) {
    if (t)                                          // Mor/forelder finnes:
       return((t->left == s1  &&  t->right == s2) ||  // Er søsken på den ene
              (t->left == s2  &&  t->right == s1) ||  //   eller andre siden:
              erSosken(t->left, s1, s2) ||           // Er evt. søsken lengre
              erSosken(t->right, s1, s2));           //   nede i treet:
    else                                            //  't' er lik nullptr:
       return false;
}


/**
 *  OPPGAVE 3B - Kutter rekursivt du noder i treet (og deres subtrær)
 *               større enn eller lik en gitt verdi.
 *
 *  NB: At det videre nedover i de utkuttede subtrærne evt. er IDer som er
 *      mindre enn  verdi  tar vi ikke hensyn til. At det i C++ på denne måten
 *      egentlig oppstår en memory-lekkasje, ved at det ikke sies 'delete' om
```

```
 *      alle nodene i subtrærne, tar vi heller ikke hensyn til.
 *      Vi forutsetter at rota ikke har en verdi som gjør at den skal
 *      kuttes vekk/ut.
 *
 *  @param   t      -  Noden å evt. kutte vekk subtrær under
 *  @param   verdi  -  Noder med ID >= 'verdi' kuttes vekk (og subtrærne)
 */
void kappTreNedentil(Node* t, const int verdi) {
    if (t)  {                                    //  Node finnes:
       if (t->left  &&  t->left->ID >= verdi)    //  Venstre barn finnes, og
          t->left = nullptr;                     //    skal kuttes vekk.
       else                                      //  Ellers besøkes venstre
          kappTreNedentil(t->left, verdi);       //    subtre.

       if (t->right  &&  t->right->ID >= verdi)  //  Samme for høyre barn:
          t->right = nullptr;
       else
          kappTreNedentil(t->right, verdi);
    }
}


/**
 *  Hovedprogrammet:
 */
int main() {

    Node* p[20];
    for (int i = 1; i <= 19; i++)  p[i] = new Node(i);
    gRoot = p[1];                              //  Bygger treet:
    p[1]->left = p[2];  p[1]->right = p[3];    //           1             //
    p[2]->left = p[4];  p[2]->right = p[5];    //       /       \         //
    p[3]->left = p[6];  p[3]->right = p[7];    //      2         3        //
    p[4]->left = p[14]; p[4]->right = p[15];   //    /   \     /   \      //
    p[5]->left = p[8];                         //   4     5   6     7     //
    p[6]->right = p[9];                        //  / \    /     \   / \   //
    p[7]->left = p[10]; p[7]->right = p[12];   // 14 15  8       9 10 12  //
    p[8]->left = p[16]; p[8]->right = p[17];   //       / \     /   \     //
    p[9]->left = p[11];                        //     16  17   11    13   //
    p[10]->right = p[13];                      //         /     \         //
    p[17]->left = p[18];                       //        18     19        //
    p[11]->right = p[19];



    //  Tester 3A:
    Node* node1, *node2;
    node1 = p[4];   node2 = p[5];
    cout << "\nNode 4 og 5 er "
         << (!erSosken(gRoot, node1, node2) ? "IKKE " : "") << "søsken!\n";
    node1 = p[9];   node2 = p[10];
    cout << "\nNode 9 og 10 er "
         << (!erSosken(gRoot, node1, node2) ? "IKKE " : "") << "søsken!\n";
    node1 = p[16];   node2 = p[17];
    cout << "\nNode 16 og 17 er "
         << (!erSosken(gRoot, node1, node2) ? "IKKE " : "") << "søsken!\n";
    node1 = p[18];   node2 = p[19];
    cout << "\nNode 18 og 19 er "
         << (!erSosken(gRoot, node1, node2) ? "IKKE " : "") << "søsken!\n";
    node1 = p[2];   node2 = p[3];
    cout << "\nNode 2 og 3 er "
         << (!erSosken(gRoot, node1, node2) ? "IKKE " : "") << "søsken!\n";
    node1 = p[6];   node2 = p[6];
    cout << "\nNode 6 og 6 er "
         << (!erSosken(gRoot, node1, node2) ? "IKKE " : "") << "søsken!\n";
    node1 = p[1];   node2 = p[1];
    cout << "\nNode 1 og 1 er "
         << (!erSosken(gRoot, node1, node2) ? "IKKE " : "") << "søsken!\n";
```

```
134     node1 = p[10];   node2 = nullptr;
135     cout << "\nNode 10 og 'nullptr' er "
136          << (!erSosken(gRoot, node1, node2) ? "IKKE " : "") << "søsken!\n\n";
137
138
139
140     //  Tester 3B:
141     cout << "\nNodene når de større enn 11 og deres subtrær er fjernet:\n\t";
142     kappTreNedentil(gRoot, 12);
143     traverseInorder(gRoot);
144
145     cout << "\n\n\n";
146     return 0;
147 }
```

10.1.4 OPG 4

```cpp
/**
 *   Løsningsforlag til ekstraeksamen i AlgMet, vår 2023, oppgave 4.
 *
 *   @file     EX_V23_4.CPP
 *   @author   Frode Haug, NTNU
 */


#include <iostream>             //  cout
#include <cstring>              //  strlen
using namespace std;

char tekst1[] = "ABCDEFGABCDEFGHIJKLKLMNQPAHGIJKLMNPWQERSTTRETSUVVUX";  //  W
char tekst2[] = "KTWRQOPDENSKTWRQOPDENSQOPDENSKTWRQOPDENSKTWRQOPDENS";  // ' '


/**
 *  OPPGAVE 4A - Finner og returnerer den første bokstaven i en tekst som
 *               forekommer bare EN gang, UTEN å unnalagre hjelpedata.
 *
 *  Orden:  N*N
 *
 *  @param    tekst  -  Teksten som skal gjennomgås/sjekkes
 *  @param    n      -  Lengden til/antall tegn i 'tekst'
 *  @return   FØRSTE bokstaven med bare EN forekomst, evt. ' ' (blank)
 */
char forsteSingleBokstav1(const char tekst[], const int n) {
   int i, j;                            //  Løkkevariable.
   bool duplikat;                       //  Funnet duplikat eller ei.

   for(i = 0;  i < n;  i++) {           //  For hver bokstav i teksten:
      duplikat = false;                 //  Antar ingen duplikat.
                                        //  Sjekker HELE teksten for duplikat:
      for(j = 0;  j < n;  j++)          //  IKKE korrekt bare at: 'j=i+1' !!!
                                        //  IKKE samme indeks, men like:
        if (i != j  &&  tekst[i] == tekst[j])  {
           duplikat = true;             //  Duplikat funnet!
//           break;                     //  Trenger ikke sammenligne videre.
        }
        if (!duplikat) return (tekst[i]); //  Ingen duplikater!
   }                                    //  Returnerer aktuell bokstav.
   return (' ');                        //  Ingen single/unike funnet
}                                       //    - returnerer ' ' (blank).


/**
 *  OPPGAVE 4B - Finner og returnerer den første bokstaven i en tekst som
 *               forekommer bare EN gang, VED å unnalagre hjelpedata.
 *
 *  Orden:  N    ( strengt tatt:  3/2 N )
 *
 *  @param    tekst  -  Teksten som skal gjennomgås/sjekkes
 *  @param    n      -  Lengden til/antall tegn i 'tekst'
 *  @return   FØRSTE bokstaven med bare EN forekomst, evt. ' ' (blank)
 */
char forsteSingleBokstav2(const char tekst[], const int n) {
   int i;                               //  Løkkevariabel.
   int antall[26];                      //  Telling av bokstavforekomster.

   for (i = 0;  i < 26;  i++)  antall[i] = 0;  //  Nullstiller hjelpearray.

   for (i = 0;  i < n;  i++)            //  Teller opp forekomsten av ALLE
       antall[tekst[i]-'A']++;          //    bokstavene i teksten.
                                        //  Finner og returnerer den FØRSTE i
   for (i = 0;  i < n;  i++)            //    teksten med bare EN forekomst:
```

```
66          if (antall[tekst[i]-'A'] == 1)  return (tekst[i]);
67
68      return (' ');                          //  Ingen single/unike funnet
69  }                                          //    - returnerer ' ' (blank).
70
71
72  /**
73   *  Hovedprogrammet:
74   */
75  int main() {
76
77      //  Tester 4A:
78      cout << "\nFørste unike forekomst i 'tekst1' er bokstaven:  '"
79           << forsteSingleBokstav1(tekst1, strlen(tekst1)) << "'\n";
80      cout << "\nFørste unike forekomst i 'tekst2' er bokstaven:  '"
81           << forsteSingleBokstav1(tekst2, strlen(tekst2)) << "'\n";
82      cout << "\nFørste unike forekomst i '' (tom tekst) er bokstaven:  '"
83           << forsteSingleBokstav1("", 0) << "'\n\n\n";
84
85      //  Tester 4B:
86      cout << "\nFørste unike forekomst i 'tekst1' er bokstaven:  '"
87           << forsteSingleBokstav2(tekst1, strlen(tekst1)) << "'\n";
88      cout << "\nFørste unike forekomst i 'tekst2' er bokstaven:  '"
89           << forsteSingleBokstav2(tekst2, strlen(tekst2)) << "'\n";
90      cout << "\nFørste unike forekomst i '' (tom tekst) er bokstaven:  '"
91           << forsteSingleBokstav2("", 0) << "'\n\n\n";
92
93      return 0;
94  }
```

10.2 H22

NTNU
Kunnskap for en bedre verden

Institutt for datateknologi og informatikk

Eksamensoppgave i IDATG2102 – Algoritmiske metoder

Faglig kontakt under eksamen:	**Frode Haug**
Tlf:	**950 55 636**
Eksamensdato:	**5.desember 2022**
Eksamenstid (fra-til):	**13:00-17:00 (4 timer)**
Hjelpemiddelkode/Tillatte hjelpemidler:	**F - Alle trykte og skrevne.** (kalkulator er *ikke* tillatt)
Annen informasjon:	
Målform/språk:	**Bokmål**
Antall sider (inkl. forside):	**4**

Informasjon om trykking av eksamensoppgaven
Originalen er:
1-sidig X **2-sidig** ☐
sort/hvit X **farger** ☐
Skal ha flervalgskjema ☐

Kontrollert av:

Dato Sign

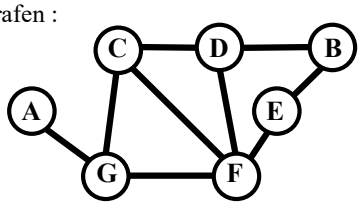
Oppgave 1 (teori, 25%)

Denne oppgaven inneholder tre totalt uavhengige oppgaver fra pensum.

a) 8 **Skriv/tegn det resulterende 2-3-4 treet når bokstavene: NEUSCHWANSTEIN settes inn i det. Gjør også om sluttresultatet til et Red-Black tre.**

b) 9 Quicksort skal utføres på bokstavene/keyene: OBERSTDORF
Lag en oversikt/tabell der du for hver rekursive sortering skriver de involverte bokstavene og markerer/uthever hva som er partisjonselementet.

c) 8 Vi har grafen :

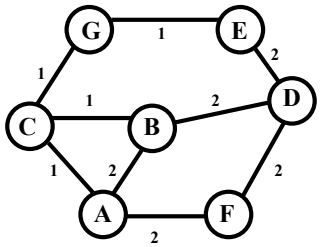


Skriv opp nabomatrisen.
Skriv/tegn dybde-først-søketreet, ved bruk av nabomatrisen og når koden `DFS(…)` i EKS_30_DFS_BFS.cpp brukes/kjører, og *vi starter i node D*.

Oppgave 2 (teori, 25%)

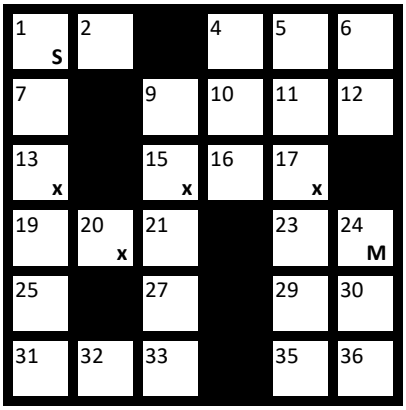
Denne oppgaven inneholder tre totalt uavhengige oppgaver fra pensum.

a) 8 Følgende vektede (ikke-rettede) graf er gitt:



Vi bruker nabomatrise, og starter i node D. **Skriv/tegn opp minimums spenntreet (MST)** for denne grafen, etter at koden i EKS_31_MST.cpp er utført/kjørt.
Skriv også opp innholdet i/på fringen etterhvert som konstruksjonen av MST pågår.
NB: *Husk at ved lik vekt så vil noden sist oppdatert (nyinnlagt eller endret) havne først på fringen ift. andre med den samme vekten.*

b) 8 Vi har rutenettet med **S**(tart)- og **M**(ål)-ruter:



Hva vil den minste f-verdien i rutene (x) 13, 20, 15 og 17 kunne være
når det *kun* kan gås opp/ned/høyre/venstre (*ikke* på skrå) med en vekt på 1 (en), og som heuristikk brukes Manhatten distanse (summen av antall ruter horisontalt og vertikalt til og inkludert målet).

c) 9 Vi har følgende bokstaver og `forelder`-array ifm. Huffman-koding (`frekvens`-arrayen er ukjent/irrelevant):

char(k)	' '	A	B	E	G	I	L	N	S	U
k	0	1	2	5	7	9	12	14	19	21
forelder[k]	32	29	-28	31	28	-27	-34	-30	27	33

k	27	28	29	30	31	32	33	34	35
forelder[k]	30	-29	-31	-32	-33	34	35	-35	0

Skriv/tegn opp Huffmans kodingstreet/-trien. **Skriv bokstavenes bitmønster**.

Vi har også følgende bitstrøm, som er kodet etter denne trien:
10100101011110011001010010001111011001110100101110100 00
Hva står i denne meldingen?

Oppgave 3 (koding, 30%)

Vi har et binært tre bestående av nodene:

```
struct Node  {
  int   ID;             // Nodens ID/key/nøkkel/navn (et tall).
  Node *left, *right;   // Referanse til begge subtrærne (evt. nullptr/NULL).
  Node* parent;         // Peker oppover igjen til forelder/mor,
};                      //    evt. nullptr/NULL  om noden er rota.
```

Legg merke til `parent`, som *alltid* peker til nodens mor/forelder (rotas `parent` er `nullptr/NULL`) . Ifm. de to funksjonene du skal lage nedenfor, kan du forutsette at parameteren `n` *ikke* peker til `nullptr/NULL`. **Hint:** Tegn opp et litt større tilfeldig binært (søke)tre, så er det lettere å studere/ tenke på hvordan de ulike funksjonene skal operere.

NB: I *hele* oppgave 3 skal det *ikke* innføres globale data eller flere `struct`-medlemmer enn angitt ovenfor. Det skal heller *ikke* brukes andre hjelpestrukturer - som f.eks. array, stakk, kø eller liste.

a) **Lag den *ikke-rekursive* funksjonen** `Node* nestePreorder(const Node* n)`

Funksjonen mottar pekeren `n` som parameter. Denne peker til en helt vilkårlig node ett eller annet sted i treet. Funksjonen skal returnere en peker til den *neste* noden i *preorder rekkefølge*. Er `n` selv den siste noden i treet, så returneres `nullptr/NULL`.

Hint: Har en node venstre og/eller høyre barn, så er dette rimelig enkelt. Har den *ikke* det (altså `n` er selv en bladnode), må det letes oppover i treet igjen etter nærmeste høyre-node/-subtre som er ubesøkt.

b) **Lag den *ikke-rekursive* funksjonen** `Node* forrigePreorder(const Node* n)`

Funksjonen mottar pekeren `n` som parameter. Denne peker til en helt vilkårlig node ett eller annet sted i treet. Funksjonen skal returnere en peker til den *forrige* noden i *preorder rekkefølge*. Er `n` selv rota, så returneres `nullptr/NULL`.

Hint: Er `n` rota, et venstre barn eller mor har ingen venstre, så er dette rimelig enkelt. Er ikke noe av dette tilfelle, så er forrige i preorder rekkefølge en bladnode!

Oppgave 4 (koding, 20%)

Vi skal se på tall **`n`**, der summen av alle heltallelige divisorer til **`n`** er *større enn* **`2n`**.

Tre eksempler:
12 sine divisorer er 1, 2, 3, 4, 6 og 12, der 1+2+3+4+6+12 = 28 28 > 24 (= 2*12)
70 sine divisorer er 1, 2, 5, 7, 10, 14, 35 og 70, der 1+2+5+7+10+14+35+70 = 144 144 > 140 (= 2*70)
168 sine divisorer er 1,2, 3, 4, 6, 7, 8, 12, 14, 21, 24, 28, 42, 56, 84 og 168
der 1+2+3+4+6+7+8+12+14+21+24+28+42+56+84+168 = 480 480 > 336 (= 2*168)

Skriv et program (`main`) som finner og skriver ut *antall* slike tall under 1.000.000 (1 million) der *differansen mellom summen av alle tallets divisorer og `2n` er heltallelig kvadratisk*.

(For de to første eksemplene over er differansen på summen og to ganger tallet lik 4, som er 2^2
I det tredje eksemplet er differansen 144, som er 12^2)

Litt hjelp: Biblioteksfilen `cmath` i C++ inneholder bl.a. funksjonen `sqrt(y)` for å beregne kvadratroten av `y`. Svaret er en `float`, som altså må finnes ut om er det samme som heltall også.

NB: I *hele* dette oppgavesettet skal du *ikke* bruke kode fra (standard-)biblioteker (slik som bl.a. STL og Java-biblioteket). Men de vanligste includer/import du brukte i 1.klasse er tilgjengelig. Koden kan skrives valgfritt i C++ eller Java.

Løkke tæll!
FrodeH

10.2.1 OPG 1

```
/**
 *   Løsning til eksamen i AlgMet, desember 2022, oppgave 1.
 *
 *   @file     EX_H22_1.TXT
 *   @author   Frode Haug, NTNU
 */



OPPGAVE A:
==========

    2-3-4 tre:                N
                            /         \
                          E             SU
                        /   \         /  |  \
                      AC    EHI      NN  ST   W


                                                 Alle 3-noder kan vippes motsatt vei.
    Red-Black tre:            N                Her er alle konsekvent høyre-rotert:
                            /         \
                          E             S
                        /   \         /  \\
                       /     \       /     U
                      A       H     N     / \
                       \\  // \\     \\  S   W
                         C  E   I      N   \\
                                           T




OPPGAVE B:
==========

    "OBERSTDORF" sorteres vha. Quicksort.
    Oversikten/tabellen for hver rekursive sortering blir da:
    (NB: Partisjonselementet er skrevet med STOR bokstav,
         mens resten er skrevet med små bokstaver.)


                1  2  3  4  5  6  7  8  9 10
    Initielt:   O  B  E  R  S  T  D  O  R  F

                d  b  e  F  s  t  o  o  r  r
                d  b  E
                B  d
                            r  o  o  R  s  t
                            o  O  r
                                        s  T




OPPGAVE C:
==========

    Nabomatrisen:    | A  B  C  D  E  F  G
                   --|--------------------
                   A | 0  0  0  0  0  0  1
                   B | 0  0  0  1  1  0  0
```

```
   C | 0  0  0  1  0  1  1
   D | 0  1  1  0  0  1  0
   E | 0  1  0  0  0  1  0
   F | 0  0  1  1  1  0  1
   G | 1  0  1  0  0  1  0      (evt. 1 på diagonalen også)


Dybde-først-søketreet når starter i D:

                .............
                :  .........D
                :  :       /
                :  :      B
                :  :     /
                :  :    E
                :  :   /
                :  :..F..
                :    /  :
                :...C   :
                   /    :
                  G.....:
                 /
                A
```

10.2.2 OPG 2

```
/**
 *   Løsning til eksamen i AlgMet, desember 2022, oppgave 2.
 *
 *   @file     EX_H22_2.TXT
 *   @author   Frode Haug, NTNU
 */



OPPGAVE A:
==========

     Fringen etterhvert (NVd = Nodenavn, Vekt, dad):

                        F2d   A2f   C1a   G1c
                        E2d   E2d   E2d   B1c   E1g
                  D*    B2d   B2d   B2d   E2d   B1c  B1c


           Minimums spenntreet:     G -------- E
                                   /
                                  C ---- B       D
                                   \            /
                                    A -------- F




OPPGAVE B:
==========

     13:     g =  2        h =  6           f =  8
     20:     g =  4        h =  4           f =  8
     15:     g =  6        h =  4           f = 10
     17:     g =  8        h =  2     f = 10



OPPGAVE C:
==========
                                    (Laget ut fra:  SULLBINGE ULL ULLA ULLEN)

     Huffmans kodingstreet/trien:          35
                                    /              \
                                33                    34
                             /      \              /       \
                            U        31           32        L
                                    /   \        /    \
                                   E     29    ' '     30
                                        /  \          /  \
                                       A    28      27    N
                                            / \    / \
                                           G   B  S   I



     Vi har følgende bitmønster for bokstavene:
          ' '    A     B     E     G      I     L    N     S     U
         100  0110  01111  010  01110  10101  11  1011  10100  00


```

```
66
67      Bitstrømmen er derfor følgende tekst/melding:  "SIL LUS BAG NSU"
```

10.2.3 OPG 3

```
/**
 *   Løsningsforlag til eksamen i AlgMet, desember 2022, oppgave 3.
 *
 *   @file     EX_H22_3.CPP
 *   @author   Frode Haug, NTNU
 */


#include <iostream>
#include <iomanip>
using namespace std;


/**
 *  Node (med ID/key, venstre/høyre subtre og peker til foreldre/mor).
 */
struct Node  {
  int   ID;                       //  Nodens ID/key/nøkkel/navn (et tall).
  Node *left, *right;  //  Referanse til begge subtrærne (evt.  nullptr/NULL).
  Node* parent;                   //  Peker oppover igjen til forelder/mor,
                                                //          evt. nullptr/NULL om er rota.
  Node (int id, Node* l, Node* r, Node* p)  //  Constructor:
       { ID = id; left = l; right = r; parent = p; }
};


Node* gRoot = nullptr;        ///<  Rot-peker  (head->right er altså IKKE rota).


/**
 *   Enum for å skrive ut på pre-, in- og postorder måte.
 */
enum Type { Pre, In, Post };


/**
 *   Skriver treet ut på rekursiv måte som pre-, in- eller postorder.
 *
 *   @param   ty  -  Order-måten (pre, in, post)
 *   @param   n   -  Noden som skal skrives ut
 */
void traverse(const Type ty, const Node* n)  {
  if (n != nullptr)  {         //  Reell node i treet:
     if (ty == Pre)  cout << ' ' << n->ID;
     traverse(ty, n->left);    //  Går til venstre subtre.
     if (ty == In)   cout << ' ' << n->ID;
     traverse(ty, n->right);   //  Går til høyre subtre.
     if (ty == Post) cout << ' ' << n->ID;
  }
}


/**
 *  OPPGAVE 3A - Returnerer NESTE node i preorder rekkefølgen ift 'n'.
 *
 *  @param   n   -  Aktuell node å finne preorder-ETTERFØLGER til
 *  @return  Peker til noden som er preorder-etterfølger (evt. 'nullptr/NULL')
 */
Node* nestePreorder(const Node* n)  {   //  'n' IKKE peker til nullptr!
                    //  'n' har venstre - denne er neste i preorder:
  if (n->left != nullptr)  return n->left;
                    //  'n' har KUN høyre, denne er neste i preorder:
  else if (n->right != nullptr)  return n->right;
  else  {           //  'n' er bladnode (da har hverken venstre eller høyre):
     Node* p = n->parent;     //  Starter hos mora/forelder.
```

```
66                      //  Så lenge ikke bladd helt opp og forbi rota og
67                      //    så lenge kommer fra høyre side eller at
68                      //    mora/forelder IKKE har et høyre barn/subtre:
69      while (p != nullptr  &&  (n == p->right ||  p->right == nullptr))  {
70         n = p;  p = n->parent;  //  Blar ett hakk opp (til mora/forelder).
71      }               //  Om stanset inni treet, med en reell høyre node -
72                      //    returneres denne høyre noden, ellers nullptr.
73      return ((p != nullptr) ? p->right : nullptr);
74    }
75  }
76
77
78  /**
79   *  OPPGAVE 3B - Returnerer FORRIGE node i preorder rekkefølgen ift 'n'.
80   *
81   *  @param    n   -  Aktuell node å finne preorder-FORGJENGER til
82   *  @return   Peker til noden som er preorder-forgjenger (evt. 'nullptr/NULL')
83   */
84  Node* forrigePreorder(const Node* n)  {   //  'n' IKKE peker til nullptr!
85                                //  'n' er rotnode, derfor ingen forrige/før:
86    if (n->parent == nullptr)  return nullptr;
87    Node* p = n->parent->left;   //  Starter øverst i mor's venstre subtre.
88                                 //  Er selv dette venstre barnet ELLER mor har
89                                 //    ingen venstre - forrige er da mora selv:
90    if (n == p  ||  p == nullptr) return n->parent;
91                         //  Looper "evig" til har funnet forrige i preorder -
92                         //     som ALLTID(!!!) er en bladnode:
93    while (true)         //  Primært blas det nedover mot høyre:
94      if (p->right != nullptr) p = p->right;
95                         //  Sekundært blas det nedover mot venstre:
96      else if (p->left != nullptr) p = p->left;
97      else return p;     //  Kommet ned til bladnode - og dette var den forrige!
98  }
99
100
101 /**
102  *   Hovedprogrammet.
103  */
104 int main()  {
105   Node *n6, *n8, *n11, *n12, *n13, *n17, *n28, *n31, *n33, *n34, *n35, *n39;
106                 // Bygger treet:          17              //
107                 //                      /        \        //
108                 //                    11          33      //
109                 //                  /    \      /    \    //
110                 //                6      13 28     35    //
111                 //                 \    /    \     / \   //
112                 //                  8  12     31  34 39   //
113   n8  = new Node(8,  nullptr,  nullptr, nullptr);
114   n12 = new Node(12, nullptr,  nullptr, nullptr);
115   n31 = new Node(31, nullptr,  nullptr, nullptr);
116   n34 = new Node(34, nullptr,  nullptr, nullptr);
117   n39 = new Node(39, nullptr,  nullptr, nullptr);
118   n6  = new Node(6,  nullptr,  n8,      nullptr);    n8->parent  = n6;
119   n13 = new Node(13, n12,      nullptr, nullptr);    n12->parent = n13;
120   n28 = new Node(28, nullptr,  n31,     nullptr);    n31->parent = n28;
121   n35 = new Node(35, n34, n39, nullptr); n34->parent = n35; n39->parent = n35;
122   n11 = new Node(11, n6,  n13, nullptr); n6->parent  = n11; n13->parent = n11;
123   n33 = new Node(33, n28, n35, nullptr); n28->parent = n33; n35->parent = n33;
124   n17 = new Node(17, n11, n33, nullptr); n11->parent = n17; n33->parent = n17;
125   gRoot = n17;
126
127
128   cout << "\n\nTreet traversert:\n\tPreorder:   ";
129   traverse(Pre,  gRoot);   cout << "\n\tInorder:    ";
130   traverse(In,   gRoot);   cout << "\n\tPostorder:  ";
131   traverse(Post, gRoot);
132
133   Node* svar;
```

```
134
135     cout << "\n\n\nNESTE PREORDER ift:\n";
136     svar = nestePreorder(n17);
137     cout << "\t17: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  11
138     svar = nestePreorder(n11);
139     cout << "\t11: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //   6
140     svar = nestePreorder(n6);
141     cout << "\t 6: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //   8
142     svar = nestePreorder(n13);
143     cout << "\t13: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  12
144     svar = nestePreorder(n8);
145     cout << "\t 8: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  13
146     svar = nestePreorder(n12);
147     cout << "\t12: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  33
148     svar = nestePreorder(n31);
149     cout << "\t31: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  35
150     svar = nestePreorder(n34);
151     cout << "\t34: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  39
152     svar = nestePreorder(n39);
153     cout << "\t39: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //   0
154
155     cout << "\n\nFORRRIGE PREORDER ift:\n";
156     svar = forrigePreorder(n17);
157     cout << "\t17: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //   0
158     svar = forrigePreorder(n11);
159     cout << "\t11: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  17
160     svar = forrigePreorder(n6);
161     cout << "\t 6: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  11
162     svar = forrigePreorder(n13);
163     cout << "\t13: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //   8
164     svar = forrigePreorder(n8);
165     cout << "\t 8: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //   6
166     svar = forrigePreorder(n12);
167     cout << "\t12: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  13
168     svar = forrigePreorder(n31);
169     cout << "\t31: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  28
170     svar = forrigePreorder(n34);
171     cout << "\t34: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  35
172     svar = forrigePreorder(n39);
173     cout << "\t39: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  34
174     svar = forrigePreorder(n33);
175     cout << "\t33: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  12
176     svar = forrigePreorder(n35);
177     cout << "\t35: " << setw(4) << ((svar) ? svar->ID : 0) << '\n';    //  31
178
179     return 0;
180 }
```

10.2.4 OPG 4

```
/**
 *   Løsningsforlag til eksamen i AlgMet, desember 2022, oppgave 4.
 *
 *   @file     EX_H22_4.CPP
 *   @author   Frode Haug, NTNU
 */


#include <iostream>      //  cout
#include <cmath>         //  sqrt
using namespace std;


/**
 *   Hovedprogram.
 */
int main()  {
  int   i, j,           //  Løkkevariable, der i = n (fra oppgaveteksten).
        sum,            //  Summen av alle heltallelige divisorer for 'i'.
        diff,           //  Differansen mellom 'sum' og '2*i'.
        kvad1;          //  HELTALLS kvadratrot av 'diff'.
  float kvad2;          //  FLYTTALLS kvadratrot av 'diff'.
  int   antall = 0;     //  Totalt antall aktuelle tall funnet.


  for (i = 1;  i < 1000000;  i++) {   //  Går gjennom alle aktuelle tall:
      sum = i;                        //  Initierer 'sum' til tallet selv.
                                      if (i % 50000 == 0)  cout << i << '\n';
      for  (j = 1;  j <= i/2;  j++)   //  Går max. opp til halve tallet:
         if (i % j == 0)              //  Heltallelig dividerbar:
            sum += j;                 //  Oppdaterer totalsummen.

      if (sum > 2*i)  {               //  Totalsummen større enn '2*i':
         diff = sum - (2*i);          //  Finner differansen dem imellom.
         kvad2 = sqrt(diff);          //  Beregner dets kvadratrot (float).
         kvad1 = kvad2;               //  Gjør om til int!
         if (kvad1 == kvad2)  {       //  int og float er like:
            antall++;                 //  Teller opp antallet funnet.
//            cout << "i: " << i << "   2i: " << 2*i << "   sum: " << sum
//                 << "   Diff: " << diff << "     kvad1: " << kvad1
//                 << "   kvad2: " << kvad2 <<  "     Antall: " << antall << '\n';
//            getchar();
         }
      }
  }

  printf("\n\nAntallet:  %i\n\n", antall);  //  Skriver ut oppgavens svar
                                            //    (som er 705).
  return 0;
}
```

10.3 S22

NTNU
Kunnskap for en bedre verden

Institutt for datateknologi og informatikk

Kontinuasjonseksamensoppgave i IDATG2102 – Algoritmiske metoder

Faglig kontakt under eksamen:	**Frode Haug**
Tlf:	**950 55 636**
Eksamensdato:	**8.august 2022**
Eksamenstid (fra-til):	**09:00-13:00 (4 timer)**
Hjelpemiddelkode/Tillatte hjelpemidler:	**F - Alle trykte og skrevne.** (kalkulator er *ikke* tillatt)
Annen informasjon:	
Målform/språk:	**Bokmål**
Antall sider (inkl. forside):	**4**

Informasjon om trykking av eksamensoppgaven

Originalen er:

1-sidig X **2-sidig** ☐

sort/hvit X **farger** ☐

Skal ha flervalgskjema ☐

Kontrollert av:

Dato Sign

Oppgave 1 (teori, 25%)

Denne oppgaven inneholder tre totalt uavhengige oppgaver fra pensum.

a) 8 Et av eksemplene i pensum leser og omgjør et infix-uttrykk til et postfix-uttrykk.
Vi har infix-uttrykket: (((5 + 7) * ((4 * 3) * (8 + 2))) * (3 + 6))
Hva blir dette skrevet på en postfix måte?
Skriv/tegn stakkens innhold etter hvert som koden leser tegnene i infix-uttrykket.

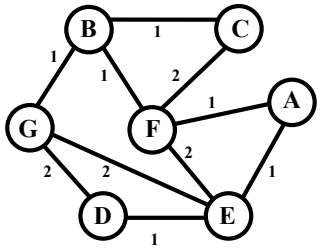
b) 8 Quicksort skal utføres på bokstavene/keyene "B A N A N B I L D E" (blanke regnes *ikke* med).
Lag en oversikt/tabell der du for hver rekursive sortering skriver de involverte bokstavene og markerer/uthever hva som er partisjonselementet.

c) 9 Heapsort (vha. bottom-up heap konstruksjon) skal utføres på bokstavene/keyene "B A N A N B I L D E" (blanke regnes *ikke* med).
Skriv/tegn opp heapens innhold etterhvert som heapen konstrueres og deretter sorteres.

Oppgave 2 (teori, 25%)

Denne oppgaven inneholder tre totalt uavhengige oppgaver fra pensum.

a) 9



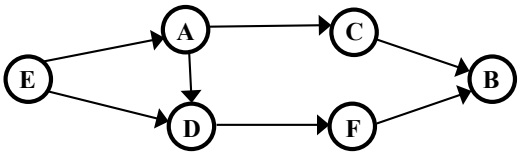
Vi bruker nabomatrise, og starter i node E. **Skriv/tegn opp minimums spenntreet (MST)** for denne grafen, etter at koden i EKS_31_MST.cpp er utført/kjørt.
Skriv også opp innholdet i/på fringen etterhvert som konstruksjonen av MST pågår.
NB: *Husk at ved lik vekt så vil noden sist oppdatert (nyinnlagt eller endret) havne først på fringen ift. andre med den samme vekten.*

b) 8 Vi har rutenettet med **S**(tart)- og **M**(ål)-ruter:

1	2 S	3	4 x	5	6
7	■	■	10	11	12
13 x	14	15 x	16	■	18
19	■	21	■	23	24
25 x	26	27	■	29	■
■	32	33 x	34	35 M	36

Hva vil den minste f-verdien i rutene 4, 13, 15, 25 og 33 (x) kunne være når det *kun* kan gås opp/ned/høyre/venstre (*ikke* på skrå) med en vekt på 1 (en), og som heuristikk brukes Manhatten distanse (summen av antall ruter horisontalt og vertikalt til og inkludert målet)

c) 8 Angi ***alle* mulige topologiske sorteringssekvenser** av nodene for den rettede (ikke-vektede) asykliske grafen («dag»):



Oppgave 3 (koding, 36%)

Vi har et binært tre bestående av:

```
struct Node  {
  int ID;                 // Nodens ID/key/nøkkel/navn (et tall).
  Node *left, *right;     // Referanse til begge subtrærne (evt. nullptr/NULL).
  Node(int id) {  ID = id; left = right = nullptr;  }
};
```

Vi har den globale variabelen:

```
Node* gRoot = nullptr;   // Rot-peker (har altså ikke at head->right er rota).
```

NB: I *hele* oppgave 3 skal det *ikke* innføres flere globale data eller `struct`-medlemmer enn angitt ovenfor. Det skal heller *ikke* brukes andre hjelpestrukturer - som f.eks. array, stakk, kø eller liste.

Det skal her lages/kodes to *helt* uavhengige funksjoner:

a) **Lag den *rekursive* funksjonen**

```
bool skrivNivaa(const Node* t, const int n, const int nivaa)
```

Funksjonen sørger (om mulig) rekursivt for at *alle* noder på nivå `nivaa` blir skrevet ut (på skjermen) fra venstre mot høyre. Dette gjøres om `n` er lik `nivaa` (og da returneres det `true`). Ellers prøver den rekursivt å få dette gjort for enda et høyere nivå (da `n` foreløpig er mindre enn `nivaa`), ved rekursivt å tilkalle seg selv med `n` *en* høyere, og for hver av barna/subtrærne (ved å sende med disse som parametre). `nivaa` forblir hele tiden den samme. Det returneres om det ønskede nivået blir funnet/oppnådd (hos *minst* ett av barna/subtrærne). Husk å håndtere at `t` også kan være `nullptr/NULL`. Rota er på nivå nr.1.

Funksjonen kalles (flere ganger) fra `main` vha. koden:

```
int aktueltNivaa = 1;
while (skrivNivaa(gRoot, 1, aktueltNivaa)) aktueltNivaa++;
```

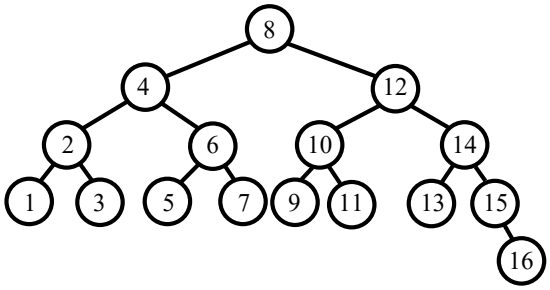
Totalt blir treet altså skrevet ut level order. Husk **NB**-punktet ovenfor.
Hvordan et tre har blitt bygd/satt opp, og `gRoot` satt til å peke på dets rot, trenger du ikke å tenke på.

b) **Lag den *rekursive* funksjonen** `Node* byggBalansertBST(const int arr[], const int start, const int slutt)`

Vi har en sortert array med `int`'er. Det skal rekursivt bygges (med `Node`'r) et mest mulig balansert binært søketre (BST) ut fra dets elementer. Input til funksjonen er hele tiden den samme arrayen `arr`, samt grensene/indeksene (`start` og `slutt`) for intervallet det skal bygges ut fra. Funksjonen kalles/brukes i `main` vha. koden:

```
int IDer[] {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
gRoot = byggBalansertBST(IDer, 0, 15);
```

Treet (under `gRoot`) vil da bli seende slik ut:



Oppgave 4 (koding, 14%)

Vi har `int tallene[1000000];` som inneholder 1 million tilfeldige *positive* heltall.
Skriv et *komplett* program som går igjennom denne arrayen, og summer sammen alle tallene som er *større eller lik det største tallet hittil funnet/registrert* tidligere i arrayen.
Dvs. vi oppsummerer *kun* tall som vil ha utgjort en *stigende (eller lik) sortert sekvens* av tall.
Skriv til slutt ut totalsummen, og antall tall som er summert sammen.

Eksempel (med bare ti tall):

5	4	3	6	7	5	6	7	5	1	4	10

Totalsum: 35 (da 5 + 6 + 7 + 7 + 10 = 35, altså 5 tall summert)
Antall summert: 5

For full score vektlegges også kort og effektiv kode.

NB: I *hele* dette oppgavesettet skal du *ikke* bruke kode fra (standard-)biblioteker (slik som bl.a. STL og Java-biblioteket). Men de vanligste includer/import du brukte i 1.klasse er tilgjengelig. Koden kan skrives valgfritt i C++ eller Java.

Løkke tæll!
FrodeH

10.3.1 OPG 1

```
/**
 *   Løsning til eksamen i AlgMet, august 2022, oppgave 1.
 *
 *   @file     EX_S22_1.TXT
 *   @author   Frode Haug, NTNU
 */



OPPGAVE A:
==========

    Infix-uttrykket:       ((( 5 + 7 ) * (( 4 * 3 ) * ( 8 + 2 ))) * ( 3 + 6 ))

    skrevet POSTFIX blir:   5  7  +  4  3  *  8  2  +  *  *  3  6  +  *

                                                +
                                          *     *  *  *           +
    Stakken underveis:     _  +  _  *  *  *  *  *  *  *  _  *  *  *  _

                           ('_' betyr at stakken er tom)




OPPGAVE B:
==========

    "BANANBILDE" sorteres vha. Quicksort.
    Oversikten/tabellen for hver rekursive sortering blir da:
    (NB: Partisjonselementet er skrevet med STOR bokstav,
         mens resten er skrevet med små bokstaver.)


                1  2  3  4  5  6  7  8  9 10
    Initielt:   B  A  N  A  N  B  I  L  D  E

                b  a  d  a  b  E  i  l  n  n
                a  a  B  b  d
                A  a
                         b  D
                                  i  l  N  n
                                  i  L




OPPGAVE C:
==========

    Bottom Up Heap-konstruksjon:
          1  2  3  4  5  6  7  8  9 10
          B  A  N  A  N  B  I  L  D  E
                      n              e
                   L           A  d
                n        b  i
             N     l  E              A
          N  L  n  D  e        a  B

    Dvs:  n  l  n  d  e  b  i  a  b  a


    Heapsort:
```

```
         1  2  3  4  5  6  7  8  9 10
         N  l  I  d  e  b  A  a  b  N
         L  E  i  d  B  b  a  a  N  n
         I  e  B  d  b  A  a  L  n  n
         E  D  b  A  b  a  I  l  n  n
         D  B  b  a  A  E  i  l  n  n
         B  A  b  a  D  e  i  l  n  n
         B  a  A  B  d  e  i  l  n  n
         A  a  B  b  d  e  i  l  n  n
         A  A  b  b  d  e  i  l  n  n

   Svar: A  A  B  B  D  E  I  L  N  N

```

10.3.2 OPG 2

```
/**
 *   Løsning til eksamen i AlgMet, august 2022, oppgave 2.
 *
 *   @file     EX_S22_2.TXT
 *   @author   Frode Haug, NTNU
 */



OPPGAVE A:
==========

    Fringen etterhvert (NVd = Nodenavn, Vekt, dad):

             D1e
             A1e   A1e         B1f
             G2e   G2e   F1a   C2f   G1b
        E*   F2e   F2e   G2e   G2e   C1b   C1b


                                  B------C
                                 / \
                                /   \
    Minimums spenntreet:       G     F------A
                                            /
                                           /
                                  D------E




OPPGAVE B:
==========

     4:          g =  2        h =  6               f =  8
    13:          g =  3        h =  7                f = 10
    15:          g =  5        h =  5               f = 10
    25:          g =  5        h =  5               f = 10
    33:          g =  8        h =  2               f = 10



OPPGAVE C:
==========

    De TRE ulike topologiske sorteringssekvensene er:
        E A C D F B
        E A D F C B
        E A D C F B
```

10.3.3 OPG 3

```
/**
 *   Løsningsforlag til eksamen i AlgMet, august 2022, oppgave 3.
 *
 *   @file     EX_S22_3.CPP
 *   @author   Frode Haug, NTNU
 */


#include <iostream>
using namespace std;


/**
 *  Node (med ID/key og venstre/høyre subtre).
 */
struct Node {
  int ID;                      //  Nodens ID/key/nøkkel/navn (et tall).
  Node *left, *right;          //  Peker til begge subtrærne (evt. 'nullptr').
  Node(int id) {  ID = id;  left = right = nullptr;  }
};


Node* gRoot = nullptr;           ///<  Peker til HELE treets rot.


/**
 *  OPPGAVE 3A - Sørger (om mulig) rekursivt for at ALLE noder på nivå 'nivaa'
 *               blir skrevet ut fra venstre mot høyre (og ut på skjermen).
 *
 *  @param    t       -  Noden å besøke/behandle
 *  @param    n       -  Aktuelt nivå man er på
 *  @param    nivaa   -  Nivået man skal frem/ned til
 *  @return   Om man har kommet frem til aktuelt 'nivaa' eller ei
 */
bool skrivNivaa(const Node* t, const int n, const int nivaa) {

  if (t == nullptr)  return false;   //  Ingen node her på dette nivået.

  if (n == nivaa)  {                 //  Noden er på aktuelt/ønsket nivå:
     cout << t->ID << ' ';           //  Skriver dens ID/navn.
     return true;                    //  Returnerer at node er funnet.
  }

  bool left  = skrivNivaa(t->left,  n+1, nivaa);  //  Søker ETT NIVÅ HØYERE
  bool right = skrivNivaa(t->right, n+1, nivaa);  //   til venstre og høyre.

  return (left || right);            //  Returnerer om node(r) ble funnet
}                                    //    eller ei på høyere nivå(er).


/**
 *  OPPGAVE 3B - Bygger rekursivt et mest mulig balansert tre ut fra arrayen
 *               'arr' og dets midtpunkt (midten mellom 'start' og 'slutt').
 *
 *  @param    arr     -  Sortert array som skal gjøres om til balansert BST
 *  @param    start   -  Nederste indeks for intervallet å bygge tre ut fra
 *  @param    slutt   -  Øvertste indeks for intervallet å bygge tre ut fra
 *  @return   Peker til rota for bygget balansert (sub)tre
 */
Node* byggBalansertBST(const int arr[], const int start, const int slutt) {
  Node* nyNode;
  int   midt = (start+slutt) / 2;

  if (start <= slutt)  {
     nyNode = new Node(arr[midt]);
```

```
66      nyNode->left  = byggBalansertBST(arr, start,  midt-1);
67      nyNode->right = byggBalansertBST(arr, midt+1, slutt);
68      return nyNode;
69    } else
70      return nullptr;
71  }
72
73
74  /**
75   *  EKSTRA - Traverserer treet under 't' rekursivt preorder.
76   *
77   *  @param   t  -  Noden å besøke/behandle
78   */
79  void traversePreorder(const Node* t) {
80    if (t) {
81        cout << t->ID << "  ("
82             << (t->left  ? t->left->ID  : -1) << "  "
83             << (t->right ? t->right->ID : -1) << ")\n";
84        traversePreorder(t->left);
85        traversePreorder(t->right);
86    }
87  }
88
89
90  /**
91   *  Hovedprogrammet:
92   */
93  int main() {
94
95      Node* p[20];
96      for (int i = 1; i <= 19; i++)  p[i] = new Node(i);
97      gRoot = p[1];                              //  Bygger treet:
98      p[1]->left = p[2];  p[1]->right = p[3];    //           1             //
99      p[2]->left = p[4];  p[2]->right = p[5];    //       /       \         //
100     p[3]->left = p[6];  p[3]->right = p[7];    //      2         3        //
101     p[4]->left = p[14]; p[4]->right = p[15];   //    /   \     /   \      //
102     p[5]->left = p[8];                         //   4     5   6     7     //
103     p[6]->right = p[9];                        //  / \    /     \   / \   //
104     p[7]->left = p[10]; p[7]->right = p[12];   // 14 15  8       9 10 12  //
105     p[8]->left = p[16]; p[8]->right = p[17];   //       / \     /   \     //
106     p[9]->left = p[11];                        //     16  17   11    13   //
107     p[10]->right = p[13];                      //         /     \          //
108     p[17]->left = p[18];                       //        18     19        //
109     p[11]->right = p[19];
110
111
112     //  Tester 3A:
113     cout << "\n\nTreet skrevet ut level order:\n\t";
114     int aktueltNivaa = 1;
115     while (skrivNivaa(gRoot, 1, aktueltNivaa))  aktueltNivaa++;
116
117
118     //  Tester 3B:
119     cout << "\n\n\n\nBalansert BST, traversert preorder (nodenes ID og dets barn):\n";
120     int IDer[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };
121     gRoot = byggBalansertBST(IDer, 0, 15);
122     traversePreorder(gRoot);
123
124
125     cout << "\n\n";
126     return 0;
127 }
```

10.3.4 OPG 4

```cpp
/**
 *   Løsningsforlag til eksamen i AlgMet, august 2022, oppgave 4.
 *
 *   @file     EX_S22_4.CPP
 *   @author   Frode Haug, NTNU
 */


#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;


int tallene[1000000];                    ///<   Alle de 1 million tallene.


/**
 *  Hovedprogrammet:
 */
int main()  {
   float  sum = 0;                       //  TOTALsummen av de økende tallene.
   int    i = 0,                         //  Indeks-variabel.
          tall,                         //  Aktuelt tall innlest/å behandle.
          storste = -1,                 //  Største tall registrert hittil.
          antall = 0;                   //  Antall tall i totalsummen.


//  IKKE EN DEL AV OPPGAVEN  (frem til stjernelinje):

   ifstream innfil("ex_s22_4.dta");     //  Åpner aktuell vil med
                                        //    1 million tall.
   if (innfil)  {                       //  Filen finnes.
      innfil >> tall;                   //  Leser (om mulig) ETT tall.
      while (!innfil.eof())  {          //  Ennå ikke nådd filslutt:
        tallene[i++] = tall;            //  Lagrer unna innlest tall.
        innfil >> tall;                 //  Leser (om mulig) NESTE tall.
      }
   } else                               //  Filen lot seg ikke finne/åpne:
     cout << "\n\nFant ikke filen 'EX_S22_4.DTA'!\n\n";

// ***************************************************************************

//  HOVEDDELEN AV OPPGAVEN:

      for (i = 0;  i < 1000000;  i++)  {  //  Går igjennom de 1 mill.tallene:
          tall = tallene[i];              //  Henter aktuelt tall.
          if (tall >= storste)  {         //  Større eller lik et tidligere:
             antall++;                    //  Antallet telles opp.
//                                 cout << tall << '\n';
             sum += tall;                 //  Totalsummen oppdateres.
             storste = tall;              //  Tar vare på det hittil største
          }                               //    registrerte tallet.
      }

      cout << fixed <<  setprecision(0);   //  Skriver ur resultatene:
      cout << "\n\nSummen av tallene er: " << sum;
      cout << "\n\nAntall tall summert: " << antall << "\n\n";

  return 0;
}
```

10.4 H21

NTNU
Kunnskap for en bedre verden

Institutt for datateknologi og informatikk

Eksamensoppgave i IDATG2102 – Algoritmiske metoder

Faglig kontakt under eksamen:	**Frode Haug**
Tlf:	**950 55 636**
Eksamensdato:	**8.desember 2021**
Eksamenstid (fra-til):	**09:00-13:00 (4 timer)**
Hjelpemiddelkode/Tillatte hjelpemidler:	**F - Alle trykte og skrevne.** (kalkulator er *ikke* tillatt)
Annen informasjon:	
Målform/språk:	**Bokmål**
Antall sider (inkl. forside):	**4**

Informasjon om trykking av eksamensoppgaven
Originalen er:
1-sidig X **2-sidig** ☐
sort/hvit X **farger** ☐
Skal ha flervalgskjema ☐

Kontrollert av:

Dato Sign

Oppgave 1 (teori, 25%)

Denne oppgaven inneholder tre totalt uavhengige oppgaver fra pensum.

a) 8 Et av eksemplene i pensum leser et postfix-uttrykk og regner ut svaret.
Vi har postfix-uttrykket: 5 2 4 + 4 2 * 3 2 4 * + + * +
Hva blir svaret ? Skriv/tegn opp stakkens innhold etter hver gang den er endret.

b) 9 Shellsort skal utføres på bokstavene «BRYLLUPENE». For hver gang indre for-løkke i eksemplet med Shellsort er ferdig (dvs. rett etter: a[j] = verdi;) :
Skriv/tegn opp arrayen og skriv verdiene til 'h' (4 og 1) og 'i' underveis i sorteringen. Marker spesielt de key'ene som har vært involvert i sorteringen.

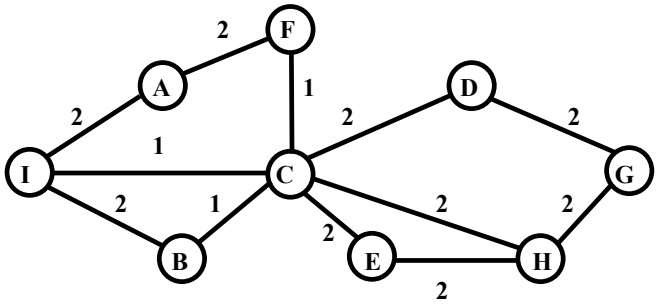
c) 8 I forbindelse med dobbelt-hashing har vi teksten «BRYLLUPSPLAN» og de to hash-funksjonene `hash1(k) = k mod 13` og `hash2(k) = 5 - (k % 5)` der `k` står for bokstavens nummer i alfabetet (1-29). Vi har også en array med indeksene 0 til 12.
Skriv hver enkelt bokstav sin `k`-verdi og returverdi fra både `hash1` og `hash2`.
Skriv også opp arrayen hver gang en bokstav hashes inn i den.

Oppgave 2 (teori, 25%)

Denne oppgaven inneholder tre totalt uavhengige oppgaver fra pensum.

a) 5 **Skriv/tegn opp Merkle treet som er basert på 11 blokker.**

b) 10 Følgende vektede (ikke-rettede) graf er gitt:



Vi bruker nabomatrise. Aktuell kode i et at pensumets eksempler utføres/kjøres på denne grafen.
Hvilke kanter er involvert i korteste-sti spenntreet fra noden A til *alle* de andre nodene?
Skriv også opp innholdet i/på fringen etterhvert som koden utføres.
NB: *Husk at ved lik vekt så vil noden sist oppdatert (nyinnlagt eller endret) havne først på fringen ift. andre med den samme vekten.*

c) 10 Følgende kanter i en (ikke-retted, ikke-vekted) graf er gitt:
DA FE CA FB GA FC BE
Utfør Union-Find *m/weight balancing (WB) og path compression (PC)* på denne grafen.
Skriv/tegn opp innholdet i `gForeldre` etter hvert som `unionerOgFinn2` kjøres/utføres. Bemerk hvor WB og PC er brukt.
Skriv/tegn også opp den resulterende union-find skogen.

Oppgave 3 (koding, 32%)

Vi har et binært tre bestående av:

```
struct Node  {
  int ID;                 // Nodens ID/key/nøkkel/navn (et tall).
  Node *left, *right;     // Referanse til begge subtrærne (evt. nullptr/NULL).
  Node(int id) {  ID = id; left = right = nullptr;  }
};
```

Vi har de to globale variablene:

```
Node* gRoot = nullptr;    // Rot-peker (har altså ikke at head->right er rota).
const int MAX = 999;      // Max.nodehøyde (høyere enn reelt. Brukes kun i 3a).
```

Det skal her lages/kodes to *helt* uavhengige funksjoner.
Begge funksjonene kalles initielt fra `main` med bl.a. `gRoot` som parameter.
Hvordan et tre har blitt bygd/satt opp, trenger du ikke å tenke på.

NB: I *hele* oppgave 3 skal det *ikke* innføres flere globale data eller `struct`-medlemmer enn angitt ovenfor. Det skal heller *ikke* brukes andre hjelpestrukturer - som f.eks. array, stakk, kø eller liste.

a)

Lag den *rekursive* funksjonen `int minimumHoyde(const Node* t)`

Funksjonen skal finne *minimumshøyden* i (sub)treet tilpekt av `t`. Altså *nivået for bladnoden på det laveste/minste nivået under* `t`. Rota er på nivå nr.1, dens barn er på nivå 2, rotas barnebarn er på nivå 3, osv. Et tomt tre (`gRoot == nullptr`) har minimumshøyde lik 0 (null). Funksjonen skal altså rekursivt finne og returnere minimumshøyden for node `t` ift. det laveste subtreet. Bladnoder har høyde lik 1 (ift. sine ikke-eksisterende barn). Er `t` lik `nullptr`, er dets høyde uklart/udefinert. Til dette bruker (og returnerer) vi `MAX`. Husk å spesialbehandle at treet evt. er helt tomt. **NB:** Husk at det *ikke* skal innføres flere globale variable.

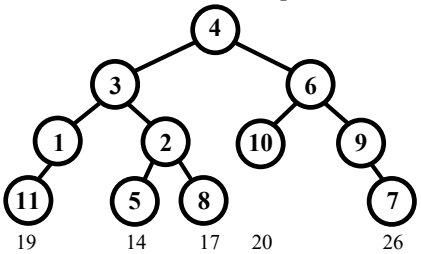
b)

Lag den *rekursive* funksjonen

```
int slettNoderPaaStiMedMinSum(Node* & t,
                              const int sum, const int min)
```

Funksjonen skal rekursivt slette (`delete`) *alle* noder som er på en sti (fra rot til bladnode), der *totalsummen* av `ID`'ene for *alle* nodene på stien *kun* er *mindre enn* `min`.

For eksempel:



Kallet `slettNoderPaaStiMedMinSum(gRoot, 0, 18)` medfører at nodene 2, 5 og 8 slettes, da disse ligger på stier med *totalsummer* mindre enn 18. Men nodene 3 og 4 slettes *ikke*, da de også ligger på en sti med totalsum høyere eller lik enn 18. Hadde `min`-parameteren i stedet vært 24, ville *alt* på venstre side av noden 4 ha blitt slettet, samt noden 10.

NB1: Funksjonen *skal* returnerer *totalsummen på lengste sti* som går gjennom `t`.
NB2: `sum` er totalsummen *hittil* på stien der `t` ligger.
NB3: `t` er referanseoverført! Dermed kan det den peker til evt. slettes (`delete`), og at den (dvs. «mor sin» `left` eller `right`) kan evt. bli satt til `nullptr/NULL`.

NB4: I Java er det ingen slik måte å direkte oppdatere selve original-parameter-referansen. Men skriver du kode i Java, så *later vi som* at dette er mulig ved å bruke `Node & t`. Dermed er det *ikke* en *lokal kopi-referanse* inni funksjonen som oppdateres, men selve den *originale medsendte referansen* som kan settes til NULL (og urefererte noder slettes jo automatisk i Java).

Oppgave 4 (koding, 18%)

Lag den *ikke-rekursive* funksjonen

```
void flettToSorterteArrayer(int a[],
                            const int b[],
                            const int aLen, const int bLen)
```

Funksjonen skal *flette sammen* den *sorterte* arrayen `a` med den *sorterte* arrayen `b` inni igjen i en fortsatt sortert array `a`, *uten å bruke ekstra memory/hjelpearray*. Begge arrayene inneholder altså heltall, og disse *kan også være negative*. Verdien 0 (null) i `a` betyr at den aktuelle «skuffen» er ledig. Array `a` er så lang at det er *eksakt* plass til elementene fra `b` i den. Dvs. det er like mange nuller i `a` som antall elementer i `b`. `aLen` og `bLen` er antall elementer i de to arrayene.
Disse ligger f.o.m. indeks nr.0 (null) t.o.m indeks nr. `aLen-1/bLen-1`.
For eksempel:

array `a`:	-8	-4	0	0	-1	3	0	7	0	9	11	0	0	15	19	(15 elementer, derav 6 nuller)
array `b`:	-10	-3	-1	4	9	24										(6 elementer)
`a` etterpå:	-10	-8	-4	-3	-1	-1	3	4	7	9	9	11	15	19	24	(15 samsorterte elementer)

NB: I *hele* dette oppgavesettet skal du *ikke* bruke kode fra (standard-)biblioteker (slik som bl.a. STL og Java-biblioteket). Men de vanligste includer/import du brukte i 1.klasse er tilgjengelig. Koden kan skrives valgfritt i C++ eller Java.

Løkke tæll!
FrodeH

10.4.1 OPG 1

```
/**
 *   Løsning til eksamen i AlgMet, desember 2021, oppgave 1.
 *
 *   @file     EX_H21_1.TXT
 *   @author   Frode Haug, NTNU
 */



OPPGAVE A:
==========

       Postfix-uttrykket: 5  2  4  +  4  2  *  3  2  4  *  +  +  *  +

                          ( skrevet Infix: (5 + (2 + 4) * (4 * 2 + (3 + 2 * 4))) )

       har svaret: 119
                                                    4
                                                 2  2  8
                                       2     3  3  3  3  11
                                 4     4  4  8  8  8  8  8   19
                              2  2  6  6  6  6  6  6  6  6   6   114
       Stakken underveis:  -  5 5  5  5  5  5  5  5  5  5  5   5     5  119  -



OPPGAVE B:
==========

       H: 4   I:  5   bryllupene               (ingenting skjer)
       H: 4   I:  6   bryllupene               (ingenting skjer)
       H: 4   I:  7   brPlluYene        'P' vandrer forbi 'Y'
       H: 4   I:  8   brpEluyLne        'E' vandrer forbi 'L'
       H: 4   I:  9   brpeluylne              (ingenting skjer)
       H: 4   I: 10   bEpelRylnU        'E' vandrer forbi 'R' og 'U'

       H: 1   I:  2   bepelrylnu              (ingenting skjer)
       H: 1   I:  3   bepelrylnu              (ingenting skjer)
       H: 1   I:  4   beEPlrylnu        'E' vandrer forbi 'P'
       H: 1   I:  5   beeLPrylnu        'L' vandrer forbi 'P'
       H: 1   I:  6   beelprylnu              (ingenting skjer)
       H: 1   I:  7   beelprylnu              (ingenting skjer)
       H: 1   I:  8   beelLPRYnu        'L' vandrer forbi 'P', 'R' og 'Y'
       H: 1   I:  9   beellNPRYu        'N' vandrer forbi 'P', 'R' og 'Y'
       H: 1   I: 10   beellnprUY        'U' vandrer forbi 'Y'



OPPGAVE C:
==========

     Keyene:            B   R   Y   L   L   U   P   S   P   L   A   N
       k (alfabetnr):   2  18  25  12  12  21  16  19  16  12   1  14

       Hash1 (M = 13):  2   5  12  12  12   8   3   6   3  12   1   1
       Hash2:           3   2   5   3   3   4   4   1   4   3   4   1


     Indeks:           0   1   2   3   4   5   6   7   8   9  10   11  12
                       -   -   B   -   -   -   -   -   -   -   -   -   -
                       -   -   B   -   -   R   -   -   -   -   -   -   -
```

```
                        -   -   B   -   -   R   -   -   -   -   -   -   Y
                        -   -   B   -   -   R   -   -   L*  -   -   -   Y
                        -   -   B   -   -   R   -   -   L*  -   -   L*  Y
                        -   -   B   U*  -   R   -   -   L*  -   -   L*  Y
                        -   -   B   U*  -   R   -   P*  L*  -   -   L*  Y
                        -   -   B   U*  -   R   S   P*  L*  -   -   L*  Y
                        -   -   B   U*  -   R   S   P*  L*  -   P*  L*  Y
                        -   L*  B   U*  -   R   S   P*  L*  -   P*  L*  Y
                        -   L*  B   U*  -   R   S   P*  L*  A*  P*  L*  Y
                        -   L*  B   U*  N*  R   S   P*  L*  A*  P*  L*  Y

                      (* = bokstaver som hashes på plass ved bruk av hash2 også.)

```

10.4.2 OPG 2

```
/**
 *   Løsning til eksamen i AlgMet, desember 2021, oppgave 2.
 *
 *   @file     EX_H21_2.TXT
 *   @author   Frode Haug, NTNU
 */



OPPGAVE A:
==========

                                H(12345678+91011)
                               /                 \
                         H(1234+5678)              H(91011)
                        /             \              |
                   H(12+34)           H(56+78)       H(910+11)
                  /       \          /       \      /       \
              H(1+2)    H(3+4)    H(5+6)    H(7+8)  H(9+10)  H(11)
             /    \    /    \    /    \    /    \  /    \     |
           H(1) H(2) H(3) H(4) H(5) H(6) H(7) H(8) H(9) H(10) H(11)
           ---------------------------------------------------------
            B1   B2   B3   B4   B5   B6   B7   B8   B9   B10   B11




OPPGAVE B:
==========

  Korteste sti fra "A" til ALLE de andre nodene vil involvere kantene:

         AF   AI   BI   CD   CE   CH   CI   GH


   Prioritetskøen etterhvert:
                                  B-4i
                      F-2a        H-5c   H-5c   E-5c
              I-2a    C-3i   C-3i   E-5c   E-5c   D-5c   D-5c
         A*   F-2a    B-4i   B-4i   D-5c   D-5c   G-7h   G-7h   G-7h




OPPGAVE C:
==========

   "gForeldre"-arrayen etterhvert:

            A  B  C  D  E  F  G
      D A:  D  -  -  1  -  -  -
      F E:  D  -  -  1  F  1  -
      C A:  D  -  D  2  F  1  -     Weight Balancing
      F B:  D  F  D  2  F  2  -
      G A:  D  F  D  3  F  2  D     Weight Balancing
      F C:  D  F  D  6  F  D  D     Weight Balancing
      B E:  D  D  D  6  D  D  D     Path Compression


   Resulterende skog:      D
                          / / / \ \ \
                         A B C   E F G
```

```
66
```

10.4.3 OPG 3

```
/**
 *   Løsningsforlag til eksamen i AlgMet, desember 2021, oppgave 3.
 *
 *   @file     EX_H21_3.CPP
 *   @author   Frode Haug, NTNU
 */


#include <iostream>
using namespace std;


/**
 *  Node (med ID/key og venstre/høyre subtre).
 */
struct Node {
  int ID;                     //  Nodens ID/key/nøkkel/navn (et tall).
  Node *left, *right;         //  Peker til begge subtrærne (evt. 'nullptr').
  Node(int id) {  ID = id;  left = right = nullptr;  }
};


Node* gRoot = nullptr;          ///<  Peker til HELE treets rot.
const int MAX = 999;            ///<  Max.nodehøyde (LANGT høyere enn reelt).


/**
 *  EKSTRA - Traverserer treet under 't' rekursivt inorder.
 *
 *  @param   t  -  Noden å besøke/behandle
 */
void traverseInorder(const Node* t) {
  if (t) {
      traverseInorder(t->left);
      cout << "  " << t->ID;
      traverseInorder(t->right);
  }
}


/**
 *  OPPGAVE 3A - Finner rekursivt treets LAVESTE høyde/dybde.
 *
 *  @param    t  -  Noden å besøke/behandle
 *  @return   Noden 't' sin høyde ift. NÆRMESTE/LAVESTE bladnode
 */
int minimumHoyde(const Node* t) {
  int vHoyde, hHoyde,         //  Laveste høyde på venstrehøyre subtre.
      minst;                  //  Den minste høyden av de to subtrærne.

  if (!gRoot)  return 0;      //  Spesialbehandler rota - tomt tre har
                              //        høyde lik 0 (null).
  if (t)  {                   //  Reell/eksisterende node:
     if (!t->left  &&  !t->right)  return 1;  //  Er en BLADNODE. Har høyde
                                              //    lik 1 (ift. "subtrærne).
     vHoyde = minimumHoyde(t->left);          //  Finner MINIMUMshøyden på
     hHoyde = minimumHoyde(t->right);         //    de to subtrærne.

     minst = (vHoyde < hHoyde) ?  vHoyde : hHoyde;  //  Finner den MINSTE/
                                                    //    LAVESTE av dem.
     return (minst + 1);      //  Returnerer minste høyde PLUSS en selv (+1).

  } else                      //  = nullptr - og derfor UKJENT høyde !!!
    return MAX;
}
```

```
66
67
68  /**
69   *  OPPGAVE 3B - Sletter rekursivt alle noder som ligger på en sti der
70   *               TOTALSUMMEN av ALLE sti-nodenes ID er mindre enn 'min'.
71   *
72   *  @param    t    -  Noden å besøke/behandle - ref.overført - KAN bli nullptr
73   *  @param    sum  -  Totalsum HITTIL på stien der 't' ligger
74   *  @param    min  -  Minimumsverdi på stien for ikke å slette nodene på den
75   *  @return   Lengste sti som går igjennom noden 't'
76   */
77  int slettNoderPaaStiMedMinSum(Node* & t, const int sum, const int min) {
78    int vSum, hSum,             //  STØRSTE nodesum i venstre/høyre subtre.
79        storst;                 //  Den største av de to summene i subtrærne.
80
81    if (t)  {                   //  Reell/eksisterende node:
82                                //  Får tak i STØRSTE stisum i subtrærne:
83      vSum = slettNoderPaaStiMedMinSum(t->left,   sum + t->ID,  min);
84      hSum = slettNoderPaaStiMedMinSum(t->right,  sum + t->ID,  min);
85
86      storst = (vSum > hSum) ?  vSum : hSum;  //  Finner den STØRSTE av dem.
87      if (storst < min)  {                    //  Selv den største er under
88                                              //    minimumsgrensen for stien:
89          delete t;                           //  Noden selv slettes !!!
90          t = nullptr;                        //  MEDSENDT peker NULLSTILLES !!!
91      }
92
93      return storst;                          //  Returnerer største sum funnet.
94
95    } else                                    //  nullptr - returnerer bare
96      return sum;                             //    input-parameteren 'sum'.
97  }
98
99
100 /**
101  *  Hovedprogrammet:
102  */
103 int main() {
104
105     Node* p[20];
106     for (int i = 1; i <= 19; i++)  p[i] = new Node(i);
107     gRoot = p[1];                           //  Bygger treet:
108     p[1]->left = p[2];  p[1]->right = p[3];    //           1            //
109     p[2]->left = p[4];  p[2]->right = p[5];    //       /       \        //
110     p[3]->left = p[6];  p[3]->right = p[7];    //      2         3       //
111     p[4]->left = p[14]; p[4]->right = p[15];   //    /   \     /   \      //
112     p[5]->left = p[8];                         //   4     5   6     7     //
113     p[6]->right = p[9];                        //  / \    /     \   / \   //
114     p[7]->left = p[10]; p[7]->right = p[12];   // 14 15  8       9 10 12  //
115     p[8]->left = p[16]; p[8]->right = p[17];   //       / \     /   \     //
116     p[9]->left = p[11];                        //     16  17   11    13   //
117     p[10]->right = p[13];                      //         /     \        //
118     p[17]->left = p[18];                       //        18     19       //
119     p[11]->right = p[19];
120
121
122     //  Tester 3A:
123     cout << "\n\nInitielt tres min.høyde:  " << minimumHoyde(gRoot) << "\n\n";
124
125
126     //  Tester 3B (og mer 3A):
127     cout << "\n\nInitiell inorder traversering:\n\t\t";
128     traverseInorder(gRoot);   cout << "\n\n";
129
130     slettNoderPaaStiMedMinSum(gRoot, 0, 24);   //  14, 15, 4, 12 slettes.
131     cout << "Etter sletting av noder på stier MINDRE ENN enn 24:\n\t\t";
132     traverseInorder(gRoot);   cout << '\n';
133     cout << "Tres min.høyde nå:  " << minimumHoyde(gRoot) << "\n\n";
```

```
134
135     slettNoderPaaStiMedMinSum(gRoot, 0, 35);   //  16, 13, 10, 7 slettes.
136     cout << "Etter sletting av noder på stier MINDRE ENN enn 35:\n\t\t";
137     traverseInorder(gRoot);   cout << "\n\n";
138
139     slettNoderPaaStiMedMinSum(gRoot, 0, 50);   //  19, 11, 9, 6, 3 slettes.
140     cout << "Etter sletting av noder på stier MINDRE ENN enn 50:\n\t\t";
141     traverseInorder(gRoot);   cout << '\n';
142     cout << "Tres min.høyde nå:  " << minimumHoyde(gRoot) << "\n\n";
143
144     slettNoderPaaStiMedMinSum(gRoot, 0, 60);   //  ALT/resten slettes.
145     cout << "Etter sletting av noder på stier MINDRE ENN enn 60:\n";
146     traverseInorder(gRoot);
147     cout << "Tres min.høyde nå:  " << minimumHoyde(gRoot) << "\n\n";
148
149     if (gRoot == nullptr)  cout << "\n\n'gRoot' peker nå til 'nullptr'.\n\n";
150
151     cout << "\n\nLager EN node som rot.\n";
152     gRoot = new Node(77);
153     cout << "Tres min.høyde nå:  " << minimumHoyde(gRoot) << "\n\n\n\n";
154
155     return 0;
156 }
```

10.4.4 OPG 4

```cpp
/**
 *   Løsningsforlag til eksamen i AlgMet, desember 2021, oppgave 4.
 *
 *   @file     EX_H21_4.CPP
 *   @author   Frode Haug, NTNU
 */


#include <iostream>
using namespace std;


/**
 *  Skriver ut hele en arrays innhold på skjermen.
 *
 *  @param   tekst  -  Del av innledende ledetekst
 *  @param   arr    -  Arrayen hvis hele innhold skrives ut på skjermen
 *  @param   len    -  Antall elementer i arrayen, indeksert 0 til len-1
 */
void skriv(const char tekst[], const int arr[], const int len)  {
  cout << "\nArrayen " << tekst << ":  ";
  for (int i = 0;  i < len;  i++)
      cout << "  " << arr[i];
  cout << '\n';
}


/**
 *  Fletter sammen to sorterte arrayer, UTEN å bruke ekstra memoryplass.
 *
 *  @param   a     -  Arrayen med ledig plass, der det sammenflettede legges
 *  @param   b     -  Arrayen som skal flettes inn i arrayen 'a'
 *  @param   aLen  -  Antall elementer i arrayen 'a'
 *  @param   bLen  -  Antall elementer i arrayen 'b'
 */
void flettToSorterteArrayer(int a[], const int b[],
                            const int aLen, const int bLen) {
   int i, j, k,             //  Løkkevariable.
       bakerst = aLen-1;    //  Initierer til bakerste "skuff"/indeks.

   for (i = aLen-1;  i >= 0;  i--)  //  Flytter alle AKTUELLE tall
       if (a[i] != 0)               //     (IKKE 0 (null)) til bakerst i
          a[bakerst--] = a[i];      //     'a'-arrayen.
                                    //  NB:  Kan IKKE erstatte a[i] med '0'!!!
   skriv("A inni", a, aLen);

   i = bakerst+1;   j = k = 0;      //  'i' starter der første reelle tall
                                    //    er i 'a'.  'j' og 'k' på starten
                                    //    av arrayene 'b' og 'a'.
   while (i < aLen  &&  j < bLen)   //  Fletter sammen arrayene inn i 'a':
       if (a[i] < b[j])             //  Neste element i 'a' er minst:
          a[k++] = a[i++];
       else                         //  Neste element i 'b' er minst
          a[k++] = b[j++];          //    (eller likt).

   while (j < bLen)                 //  Om 'b' IKKE ble ferdiglest, MÅ dens
       a[k++] = b[j++];             //    tall legges til BAKERST i 'a'!!!
                                    //    Ble ikke 'a' ferdiglest, så ligger
}                                   //    tallene der allerede!!!


/**
 *  Hovedprogrammet:
 */
int main() {
```

```
66      int arrA[15] =  { -8, -4, 0, 0, -1, 3, 0, 7, 0, 9, 11, 0, 0, 15, 19 };
67      int arrB[6] =  { -10, -3, -1, 4, 9, 24};
68
69      skriv("A før", arrA, 15);
70      skriv("B før", arrB, 6);
71
72      flettToSorterteArrayer(arrA, arrB, 15, 6);
73
74      skriv("A etter", arrA, 15);
75
76      cout << "\n\n";
77      return 0;
78  }
```

10.5 H20

NTNU
Kunnskap for en bedre verden

Institutt for datateknologi og informatikk

Eksamensoppgave i IDATG2102 – Algoritmiske metoder

Faglig kontakt under eksamen:	**Frode Haug**
Tlf:	**950 55 636**
Eksamensdato:	**26.november 2020 - HJEMME**
Eksamenstid (fra-til):	**15:00-19:00 (4 timer)**
Hjelpemiddelkode/Tillatte hjelpemidler:	**F - Alle trykte og skrevne.** (kalkulator er *ikke* tillatt)
Annen informasjon:	
Målform/språk:	**Bokmål**
Antall sider (inkl. forside):	**4**

Informasjon om trykking av eksamensoppgaven
Originalen er:
1-sidig X **2-sidig** ☐
sort/hvit X **farger** ☐
Skal ha flervalgskjema ☐

Kontrollert av:

Dato Sign

Oppgave 1 (teori, 25%)

Denne oppgaven inneholder tre totalt uavhengige oppgaver fra pensum.

a) Et av eksemplene i pensum leser og omgjør et infix-uttrykk til et postfix-uttrykk.
Vi har infix-uttrykket: (((((4 * 6) + (5 * 3)) * (5 + 6)) + 8) * 3)
Hva blir dette skrevet på en postfix måte?
Skriv/tegn stakkens innhold etter hvert som koden leser tegnene i infix-uttrykket.

b) I de følgende deloppgaver er det key'ene "A L L E R S I S T E"
(i denne rekkefølge fra venstre mot høyre, og blanke regnes *ikke* med) som du skal bruke.
For alle deloppgavene gjelder det at den initielle heap/tre er *tom* før første innlegging ("Insert") utføres. **Skriv/tegn den *resulterende* datastruktur når key'ene legges inn i:**

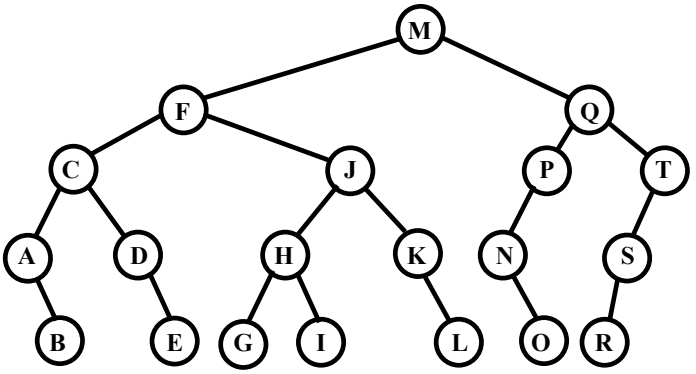
1) **en heap**
2) **et binært søketre**
3) **et 2-3-4 tre**
4) **et Red-Black tre**

c) Quicksort skal utføres på bokstavene/keyene "A L L E R S I S T E" (blanke regnes *ikke* med).
Lag en oversikt/tabell der du for hver rekursive sortering skriver de involverte bokstavene og markerer/uthever hva som er partisjonselementet.

Oppgave 2 (teori, 25%)

Denne oppgaven inneholder tre totalt uavhengige oppgaver fra pensum.

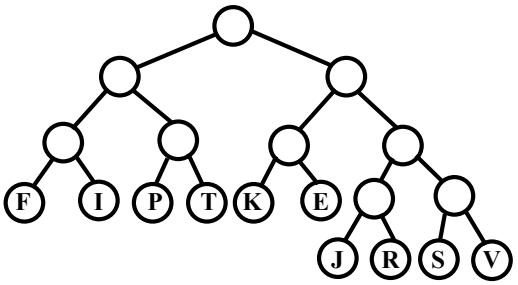
a)



Det skal fjernes («remove») noen noder fra det ovenfor gitte *binære søketreet*.
Skriv/tegn treet for hver gang, og fortell hvilken av «if ..... else if ..... else»-grenene i `EKS_28_BinertSokeTre.cpp` (dvs. Case 1, Case 2, Case 3) som er aktuelle når det etter tur fjernes henholdsvis bokstavene 'C', 'Q' og 'M'.

NB: For hver fjerning skal det *på nytt* tas utgangspunkt i *hele* det aktuelle treet.
Dvs. *på intet tidspunkt* skal det fra treet være fjernet *mer enn en* bokstav.

b) Vi har følgende ferdiglagde Huffman kodingstrie (innholdet i `gForeldre` er uinteressant):



Vi har også følgende bitstrøm (melding), som er kodet etter denne trien:
0001010110110011111011001011101111 1001
<u>Hva er bokstavenes bitmønster *og* hva er teksten i denne meldingen?</u>

c) Følgende kanter i en (ikke-retted, ikke-vekted) graf er gitt:

AC AE DB BC EB

Utfør Union-Find *m/weight balancing (WB) og path compression (PC)* på denne grafen.
<u>Skriv/tegn opp innholdet i `gForeldre` etter hvert som `unionerOgFinn2` kjøres/utføres.</u> **<u>Bemerk hvor WB og PC er brukt.</u>**
<u>Skriv/tegn også opp den resulterende union-find skogen.</u>

Oppgave 3 (koding, 28%)

Vi har et binært tre bestående av:

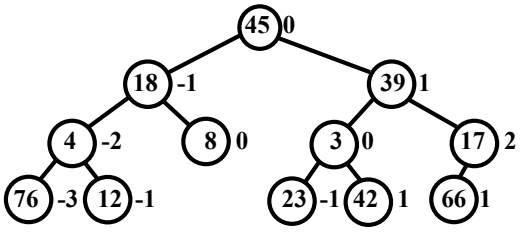
```
struct Node  {
  int ID;                 //   Nodens ID/key/nøkkel/navn (et tall).
  int avstand;            //   Nodens "vertikale avstand" ift. rota (brukes kun i 3A).
  Node *left, *right;     //   Referanse til begge subtrærne (evt. 'nullptr/NULL').
  Node(int id) { ID = id; left = right = nullptr; avstand = 0; }
};
```

Vi har *kun* den globale variabelen:

```
Node* gRoot = nullptr;    //   Rot-peker (har altså ikke at head->right er rota).
```

Det skal her lages/kodes to *helt* uavhengige funksjoner.
Hvordan et tre har blitt bygd/satt opp, trenger du ikke å tenke på.

a) **<u>Lag den *rekursive* funksjonen</u>** `void settVertikalAvstand(const Node* t)`
Funksjonen skal oppdatere *alle* nodene i treet med deres *relative vertikale avstand* ift. rota.
Dvs. `avstand` inni hver node er *en* mindre enn moren om den er et venstre barn,
mens den er *en* høyere enn mora om den er et høyre barn.
Eksempeltre, påskrevet vertikal avstand utenfor noden til høyre:



b) **Lag den *rekursive* funksjonen** `void skrivNoderUtenSosken(const Node* t)`
Funksjonen skal skrive ut `ID` for *alle* nodene i treet som *ikke* har søsken.
Husk å spesialbehandle selve rota (for *den* har jo ingen søsken).
For treet ovenfor, ville utskriften ha blitt: 45 66

Begge funksjonene kalles initielt fra `main` med `gRoot` som parameter.

NB: I *hele* oppgave 3 skal det *ikke* innføres flere globale data eller `struct`-medlemmer enn angitt ovenfor. Det skal heller *ikke* brukes andre hjelpestrukturer - som f.eks. array, stakk, kø eller liste.

Oppgave 4 (koding, 22%)

Lag den *ikke-rekursive* funksjonen `int fjernDuplikater(int a[], const int n)`

I arrayen `a` er indeksene 1 til `n` i bruk. `a` inneholder kun positive heltall, og er sortert stigende.
Men, den kan inneholde duplikate/like verdier (som pga sorteringen ligger rett etter hverandre).
Funksjonen skal fjerne eventuelle slike duplikate verdier fra arrayen. Den skal sørge for at alle de da unike/ulike verdiene blir liggende rett etter hverandre (og fortsatt sortert) i første del av arrayen.
Siste del av arrayen skal, om det fantes duplikater, fylles med 0'er (nuller).
Funksjonen skal også returnere antall forskjellige/unike/ulike verdier i arrayen.

Vi forutsetter at `n >= 0`, samt at `a[0] != a[1]`.
Antagelig bør koden spesialbehandle når `n` er 0 (null) eller 1.

For full score vektlegges effektivitet, og at ingen ekstra array brukes.

NB: I *hele* dette oppgavesettet skal du *ikke* bruke kode fra (standard-)biblioteker (slik som bl.a. STL og Java-biblioteket). Men de vanligste includer/import du brukte i 1.klasse er tilgjengelig.
Koden kan skrives valgfritt i C++ eller Java.

Løkke tæll!
FrodeH

10.5.1 OPG 1

```
/**
 *   Løsning til eksamen i AlgMet, november 2020, oppgave 1.
 *
 *   @file     EX_H20_1.TXT
 *   @author   Frode Haug, NTNU
 */



OPPGAVE A:
==========

    Infix-uttrykket:       ((((( 4 * 6 ) + ( 5 * 3 )) * ( 5 + 6 )) + 8 ) * 3 )

    skrevet POSTFIX blir:   4 6 * 5 3 * + 5 6 + * 8 + 3 *


                                       *           +
    Stakken underveis:     _  *  _  +  +  +  _  *  *  *  _  +  _  *  _

                           ('_' betyr at stakken er tom)




OPPGAVE B:
==========

    "ALLERSISTE" satt inn i:

        1)  Heap:   T S R S E L I A L E


        2)  Binært søketre:        A
                                    \
                                     L
                                    /  \
                                   E    L
                                    \    \
                                     I    R
                                    /      \
                                   E        S
                                             \
                                              S
                                               \
                                                T


        3)  2-3-4 tre:            E L R
                                 /  | |  \
                                A  EI L  SST


        4)  Red-Black tre:         L
                                  //  \\
                                  E     R
                                 / \   / \
                                A   I  L   S     (eller EI rotert andre veien)
                                   //   // \\
                                   E    S   T
```

```
66
67
68  OPPGAVE C:
69  ==========
70
71      "ALLERSISTE" sorteres vha. Quicksort.
72      Oversikten/tabellen for hver rekursive sortering blir da:
73      (NB: Partisjonselementet er skrevet med STOR bokstav,
74           mens resten er skrevet med små bokstaver.)
75
76
77                  1  2  3  4  5  6  7  8  9 10
78      Initielt:   A  L  L  E  R  S  I  S  T  E
79
80                  a  e  E  l  r  s  i  s  t  l
81                  a  E
82                           i  L  s  l  s  t  r
83                              l  R  s  t  s
84                                 S  t  s
85                                    S  t
86
```

10.5.2 OPG 2

```
/**
 *   Løsning til eksamen i AlgMet, november 2020, oppgave 2.
 *
 *   @file     EX_H20_2.TXT
 *   @author   Frode Haug, NTNU
 */



OPPGAVE A:
==========

   Når "C" fjernes:
        - "C" har et høyre barn uten sitt venstre barn, derfor er
        - den ANDRE setningen med  "else if (!fjernes->right->left) ....."  aktuell.

        - Treet under "F" etter at "C" er fjernet:

                              F
                            /     \
                           D      ....
                          / \
                         A   E
                          \
                           B


   Når "Q" fjernes:
        - vil ingen av de to første situasjonene være aktuelle, derfor er
        - den TREDJE setningen med  "else ......"  aktuell.
          Her vil den sekvensielt etterfølgende noden (dvs. "R")
          erstatte den som fjernes (dvs. "Q").

        - Treet etter at "Q" er fjernet:

                             M
                           /   \
                         ....    R
                                /   \
                               P     T
                              /     /
                             ....   S


   Når "M" fjernes:
        - vil ingen av de to første situasjonene være aktuelle, derfor er
        - den TREDJE setningen med  "else ......"  aktuell.
          Her vil den sekvensielt etterfølgende noden (dvs. "N")
          erstatte den som fjernes (dvs. "M").

        - Treet etter at "M" er fjernet:

                             N
                           /     \
                          F        Q
                        /   \     /   \
                      .... ....  P   ....
                                /
                               O




OPPGAVE B:
```

```
==========

   - Vi har følgende bitmønster for bokstavene:
                E     F     I     J     K     P     R     S     T     V
               101   000   001  1100   100   010  1101  1110   011  1111


   - Bitstrømmen utgjør derfor følgende tekst/melding:  "FETTISJERVI"




OPPGAVE C:
==========

   "gForeldre"-arrayen etterhvert:

              A  B  C  D  E
         AC:  1  -  A  -  -
         AE:  2  -  A  -  A
         DB:  2  D  A  1  A
         BC:  4  D  A  A  A    Weight Balancing
         EB:  4  A  A  A  A    Path Compression


   Resulterende skog:      A
                          / / \ \
                         B C   D E

_____________________________________________________________________________________
```

10.5.3 OPG 3

```cpp
/**
 *   Løsningsforlag til eksamen i AlgMet, november 2020, oppgave 3.
 *
 *   @file     EX_H20_3.CPP
 *   @author   Frode Haug, NTNU
 */


#include <iostream>
using namespace std;


/**
 *  Node (med ID/key, "avstand" vertikalt fra rota, og venstre/høyre subtre).
 */
struct Node {
    int ID;                     //  Nodens ID/key/nøkkel/navn (et tall).
    int avstand;                //  Nodens "vertikale avstand" ift. rota.
    Node *left, *right;         //  Peker til begge subtrærne (evt. 'nullptr').
    Node(int id) { ID = id;  left = right = nullptr;  avstand = 0;  }
};


Node* gRoot = nullptr;          ///<  Peker til HELE treets rot.


/**
 *  EKSTRA - Traverserer treet under 't' rekursivt inorder.
 *
 *  @param   t  -  Noden å besøke/behandle
 */
void traverseInorder(const Node* t) {
  if (t) {
      traverseInorder(t->left);
      cout << '\t' << t->ID << ":  " << t->avstand << '\n';
      traverseInorder(t->right);
  }
}


/**
 *  OPPGAVE 3A - Setter rekursivt vertikal "avstand" i hver node ift rota
 *               (-1 til venstre, +1 til høyre).
 *
 *  @param   t  -  Noden å besøke/behandle
 */
void settVertikalAvstand(const Node* t) {
    if (t) {                                    //  Reell node:
        if (t->left) {                          //  Har venstre barn:
            t->left->avstand = t->avstand - 1;  //  Får egen avstand - 1
            settVertikalAvstand(t->left);       //  Besøker venstre subtre.
        }
        if (t->right) {                         //  Har høyre barn:
            t->right->avstand = t->avstand + 1; //  Får egen avstand + 1
            settVertikalAvstand(t->right);      //  Besøker høyre subtre.
        }
    }
}


/**
 *  OPPGAVE 3B - Skriver rekursivt alle noder i treet som er uten søsken.
 *
 *  @param   t  -  Noden å besøke/behandle
 */
```

```
void skrivNoderUtenSosken(const Node* t) {
  if (t) {                                  //  Reell node:
      if (t == gRoot)                       //  Noden er selve rota:
          cout << ' ' << t->ID;             //  Skriver rotens ID.
      if (t->left && !t->right)             //  Har KUN v.barn UTEN h.søsken:
          cout << ' ' << t->left->ID;       //  Skriver venstres ID.
      if (!t->left && t->right)             //  Har KUN h.barn UTEN v.søsken:
          cout << ' ' << t->right->ID;      //  Skriver høyres ID.
      skrivNoderUtenSosken(t->left);        //  Besøker begge subtrærne:
      skrivNoderUtenSosken(t->right);
  }
}


void byggTre(); // Definisjon nedenfor 'main', da dette er en ekstra funksjon.


/**
 *  Hovedprogrammet:
 */
int main() {

    byggTre();

    //  Tester 3A:
    cout << "\n\nNodenes vertikale avstand ift. rota:\n";
    settVertikalAvstand(gRoot);
    traverseInorder(gRoot);


    //  Tester 3B:
    cout << "\n\nNoder uten søsken:\n\t";
    skrivNoderUtenSosken(gRoot);


    cout << "\n\n\n";
    return 0;
}


/**
 *  EKSTRA - Bygger et binært testtre (angitt til høyre nedenfor).
 */
void byggTre() {
    Node* p[20];
    for (int i = 1; i <= 19; i++)  p[i] = new Node(i);
    gRoot = p[1];
    p[1]->left = p[2];  p[1]->right = p[3];   //           1            //
    p[2]->left = p[4];  p[2]->right = p[5];   //       /       \        //
    p[3]->left = p[6];  p[3]->right = p[7];   //      2         3       //
    p[4]->left = p[14]; p[4]->right = p[15];  //    /   \     /   \      //
    p[5]->left = p[8];                        //   4     5   6     7     //
    p[6]->right = p[9];                       //  / \    /     \   / \   //
    p[7]->left = p[10]; p[7]->right = p[12];  // 14 15  8       9 10 12  //
    p[8]->left = p[16]; p[8]->right = p[17];  //       / \     /   \     //
    p[9]->left = p[11];                       //     16  17   11    13   //
    p[10]->right = p[13];                     //         /     \        //
    p[17]->left = p[18];                      //        18     19       //
    p[11]->right = p[19];
}
```

10.5.4 OPG 4

```cpp
/**
 *   Løsningsforlag til eksamen i AlgMet, november 2020, oppgave 4.
 *
 *   @file     EX_H20_4.CPP
 *   @author   Frode Haug, NTNU
 */


#include <iostream>             //  cout
using namespace std;


//  Ulike int-arrayer for testing av 'fjernDuplikater(...):
int t1[] = { 0 };                                                   //  n = 0
int t2[] = { 0,  7 };                                               //  n = 1
int t3[] = { 0,  7, 11, 12, 17 };                                   //  n = 4
int t4[] = { 0,  7,  7,  7,  7,  7,  7 };                           //  n = 6
int t5[] = { 0,  7,  7,  9,  9, 11, 11, 13, 13 };                   //  n = 8
int t6[] = { 0,  7,  7,  9, 10, 11, 11, 12, 13, 14, 14, 14 };       //  n = 11
int t7[] = { 0,  1,  1,  2,  2,  3,  3,  3,  3, 11, 12, 12,         //  n = 33
               12, 13, 14, 14, 18, 18, 19, 20, 21, 21, 22,
               22, 26, 26, 26, 27, 27, 28, 28, 29, 29, 30 };


/**
 *  OPPGAVEN - Komprimerer en array ved å fjerne duplikater, fyller på med
 *  0 (nuller) bakerst, og returnerer antall UNIKE tall i arrayen etterpå.
 *
 *  Forutsetning:   a[0] != a[1]   og at n >= 0.
 *
 *  @param    a  -  Array som får duplikater fjernet, og fylt på med 0'er
 *  @param    n  -  Initielt antall tall i 'a'
 *  @return   Antall UNIKE tall i 'a'
 */
int fjernDuplikater(int a[], const int n) {
    int i, j = 0;                       //  j = antall ulike/unike tall.

    if (n <= 1)  return n;              //  Null eller ett element i 'a'.

    for (i = 1;  i <= n;  i++)   //  Om a[i-1]==a[i] flyttes a[i] IKKE frem:
        if (a[i-1] != a[i])  a[++j] = a[i];
                                //  Fyller resten av 'a' (etter 'j') med '0':
    for (i = j+1;  i <= n;  i++)  a[i] = 0;

    return j;
}


/**
 *  Skriver ut hele en int-arrays innhold.
 *
 *  @param   tall  -  Arrayen som får sitt innhold skrevet ut på skjermen
 *  @param   n     -  Antall tall i arrayen, liggende i indeks 1 til n
 */
void skriv(const int tall[], const int n) {
  for (int i = 1;  i <= n;  i++)  cout << ' ' << tall[i];
}


/**
 *  Hovedprogrammet:
 */
int main() {

    cout << "\n\n't1' før: ";     skriv(t1, 0);
```

```
66      cout << "\nAntall ulike: " << fjernDuplikater(t1, 0);
67      cout << "\n't1' etter: ";     skriv(t1, 0);
68
69      cout << "\n\n't2' før: ";     skriv(t2, 1);
70      cout << "\nAntall ulike: " << fjernDuplikater(t2, 1);
71      cout << "\n't2' etter: ";     skriv(t2, 1);
72
73      cout << "\n\n't3' før: ";     skriv(t3, 4);
74      cout << "\nAntall ulike: " << fjernDuplikater(t3, 4);
75      cout << "\n't3' etter: ";     skriv(t3, 4);
76
77      cout << "\n\n't4' før: ";     skriv(t4, 6);
78      cout << "\nAntall ulike: " << fjernDuplikater(t4, 6);
79      cout << "\n't4' etter: ";     skriv(t4, 6);
80
81      cout << "\n\n't5' før: ";     skriv(t5, 8);
82      cout << "\nAntall ulike: " << fjernDuplikater(t5, 8);
83      cout << "\n't5' etter: ";     skriv(t5, 8);
84
85      cout << "\n\n't6' før: ";     skriv(t6, 11);
86      cout << "\nAntall ulike: " << fjernDuplikater(t6, 11);
87      cout << "\n't6' etter: ";     skriv(t6, 11);
88
89      cout << "\n\n't7' før: ";     skriv(t7, 33);
90      cout << "\nAntall ulike: " << fjernDuplikater(t7, 33);
91      cout << "\n't7' etter: ";     skriv(t7, 33);
92
93      cout << "\n\n";
94      return 0;
95  }
```