

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	Prefix-, infix- and postfix-notation	4
1.1	Infix to Postfix	4
1.1.1	Algorithm	4
1.1.2	Code	4
1.2	Postfix to Answer	5
1.2.1	Algorithm	5
1.2.2	Code	5
2	Trees	6
2.1	Parse tree	6
2.1.1	Algoritme:	6
2.1.2	Code:	6
2.2	Tree Traversal	7
2.3	Node	7
2.4	Preorder	7
2.4.1	Algoritme	7
2.4.2	Kode	7
2.5	Inorder	8
2.5.1	Algoritme	8
2.5.2	Kode	8
2.6	Postorder	8
2.6.1	Algoritme	8
2.7	Levelorder	9
2.7.1	Algoritme	9
2.7.2	Kode	9
2.8	Breadth First Search	10
3	Recursion	12
3.1	Preorder Rekursiv	12
3.2	Inorder Rekursiv	12
3.3	Postorder Rekursiv	12
4	Sorting algorithms	13
4.1	Common functions	13
4.2	Selection Sort	14
4.2.1	Algorithm	14
4.2.2	Code	14
4.3	Insertion Sort	15
4.3.1	Algorithm	15
4.3.2	Code	15
4.4	Shellsort	16
4.4.1	Algorithm:	16
4.4.2	Code:	16
4.5	Quicksort	18
4.5.1	Algorithm:	18
4.5.2	Code:	18
4.6	Heapsort	20
4.6.1	Algorithm:	20
4.6.2	Code:	20
5	Searching Algorithms	21
5.1	Binary search - array	21
5.2	Sequential search - array	21

6	Hashing	22
6.0.1	Separate Chaining	22
6.1	Linear Probing	22
6.2	Double Hashing	22
6.3	Merkle trees	22
6.3.1	Code	23
7	Graphs	25
7.1	Minimum Spanning Tree (MST) - Prim	25
7.1.1	Algoritme	25
7.1.2	Framgangsmåte	25
7.1.3	Orden	25
7.1.4	Code:	26
7.2	Shortest Path - Dijkstra	28
7.2.1	Framgangsmåte	28
7.2.2	Algoritme	28
7.3	A Star	29
7.3.1	Heuristikk	29
7.3.2	Orden	29
7.3.3	Framgangsmåte	29
7.3.4	Code - A star:	30
7.4	Union Find	33
7.4.1	Framgangsmåte	33
7.4.2	Code - Union find:	34
7.4.3	Union Find with Weight Balancing and Path Compression	36
7.5	fringe.h	37
8	Datastructures	41
8.1	Heap	41
8.1.1	Beskrivelse	41
8.1.2	UpHeap	41
8.1.3	DownHeap	41
8.1.4	Eksempel	41
8.1.5	Code - Heap class:	42

1 Prefix-, infix- and postfix-notation

1.1 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 + *

1.1.1 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.

1.1.2 Code

```
1 while ((tegn = cin.get()) != '\n'){ // reads all numbers and signs until \n
2     if(tegn == ')'){ // if it finds an end parenthesis
3         cout << stakk.top(); // prints the top sign of the stack
4         stakk.pop(); // removes the sign
5     }
6
7     // if it finds + or *, it is added to the stack
8     else if (tegn == '+' || tegn == '*') stakk.push(tegn);
9
10    while (tegn >= '0' && tegn <= '9'){
11        cout << tegn;
12        tegn = cin.get();
13    }
14
15    if (tegn != '(')
16        cout << ' ';
17 }
```

1.2 Postfix to Answer

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

1.2.1 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.

1.2.2 Code

```
1  while ((tegn = cin.get()) != '\n'){ // reads everything until \n
2      tall = 0; // reset to zero
3      while (tegn == ' ') tegn = cin.get(); // skips blanks
4
5      // if we read '+' or '*', we take the top two numbers and do
6      // the corresponding operation on them
7      if (tegn == '+'){
8          tall = stakk.top(); stakk.pop(); // tall = top number on the stack
9          tall += stakk.top(); stakk.pop(); // adds the new number now on top
10     } else if (tegn == '*'){
11         tall = stakk.top(); stakk.pop();
12         tall *= stakk.top(); stakk.pop();
13     }
14
15     // building a multi-digit number
16     while (tegn >= '0' && tegn <= '9'){
17         tall = (10 * tall) + (tegn - '0'); // converts from ascii to number
18         tegn = cin.get();
19     }
20
21     stakk.push(tall); // pushes built number
22 }
```

2 Trees

2.1 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 tree:
[MISSING IMAGE]

2.1.1 Algoritme:

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

NB!

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

2.1.2 Code:

```
1 struct Node{
2     char ID;
3     Node *left, *right;
4     Node(char id){
5         ID = id;
6         left = right = nullptr;
7     }
8 }
9
10 /* ... */
11
12 while ((tegn = cin.get()) != '\n'){
13     while(tegn == ' ') {
14         tegn = cin.get();
15     }
16     nyNode = new Node(tegn);
17     if (tegn == '+' || tegn == '*'){
18         nyNode->right = stakk.top(); stakk.pop();
19         nyNode->left = stakk.top(); stakk.pop();
20     }
21     stakk.push(nyNode);
22 }
```

2.2 Tree Traversal

2.3 Node

```
1 struct Node {
2     char ID;
3     bool besøkt; // 'besøkt' brukes KUN ifm. postorder.
4     Node *left, *right; // Initierende constructor:
5     Node(char id) { ID = id; left = right = nullptr; besøkt = false; }
6 };
```

2.4 Preorder

2.4.1 Algoritme

Besøker seg selv før den traverserer.

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

2.4.2 Kode

```
1 void traverserPreorder(Node* node){
2     if(node){
3         gStakk.push(node);
4         while (!gStakk.empty()){
5             node = gStakk.top(); gStakk.pop();
6             besøk(node);
7             if (node->right) gStakk.push(node->right);
8             if (node->left) gStakk.push(node->left);
9         }
10    }
11 }
```

2.5 Inorder

2.5.1 Algoritme

Besøker seg selv 'mellom' traverseringen

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

2.5.2 Kode

```
1 void traverserInorder(Node* node){
2     while (node || !gStakk.empty()){
3         if (node) {
4             gStakk.push(node);
5             node = node->left
6         } else{
7             node = gStakk.top(); gStakk.pop();
8             besok(node);
9             node = node->right;
10        }
11    }
12 }
```

2.6 Postorder

2.6.1 Algoritme

Besøker seg selv etter traversering

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

```
1 void traverserPostorder (Node* node) {
2     if (node) {
3         gStakk.push(node);
4         while (!gStakk.empty()){
5             node = gStakk.top(); gStakk.pop();
6             if (node->left->besokt = false) {
7                 gStakk.push(node->left);
8             }
9             if (node->right->besokt = flase) {
10                gStakk.push(node->right);
11            }
12            besok(node);
13        }
14    }
15 }
```


2.7 Levelorder

2.7.1 Algoritme

1. Leser tree linjevis

2.7.2 Kode

```
1 void traversalLevelorder(Node* node){
2     if (node){
3         gKo.push(node);
4         while (!gKo.empty()){
5             node = gKo.front(); gKo.pop();
6             besok(node);
7             if(node->left) gKo.push(node->left);
8             if(node->right) gKo.push(node->right);
9         }
10    }
11 }
```

Trees - searching

DFS - Depth First Search

Kode

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

```
1  /**
2  * @param nr - indeks (0 til ANTNODE-1) for noden som skal besøkes
3  */
4  void DFS(const int nr) {
5      gBesok[nr] = ++gBesokSomNr; // gir noden riktig "besøkt som" nr
6
7      for (int j = 0; j < ANTNODE; j++) // sjekker hele "linjen" til noden i en nabo-matrise
8          if (gNaboMatrise[nr][j]) // hvis forskjellig fra 0
9              if (gBesok[j] == USETT) DFS(j); // rekursivt besøk noden.
10 }
```

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

2.8 Breadth First Search

Kode

Iterativ algoritme som gjør et bredde først søk.

```
1  /**
2  * @param nr - Indeks (0 til ANTNODE-1) for STARTNODEN i besøket
3  */
4  void BFS(int nr) {
5      int j; // Indeks for aktuelle naboer.
6      gBesokeSenere.push(nr); // Legges BAKERST i besøkskø.
7      while (!gBesokeSenere.empty()) { // Ennå noder å besøke igjen:
8          nr = gBesokeSenere.front(); // AVLES iden til første noden på køen
9          gBesokeSenere.pop(); // FJERNER/TAR UT fra køen.
10         gBesok[nr] = ++gBesokSomNr; // Setter besøksnummeret.
11         for (j = 0; j < ANTNODE; j++) // Nodens linje i matrisen:
12             if (gNaboMatrise[nr][j]) // Er nabo med nr.j,
13                 if (gBesok[j] == USETT) { // og denne er ubesøkt:
14                     gBesokeSenere.push(j); // Legger nabo BAKERST i køen.
15                     gBesok[j] = SENERE; // Setter at delvis besøkt!!!
16                 }
17     }
18 }
```

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 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.

3.1 Preorder Rekursiv

```
1 void traverserPreorder (Node* node) {
2     if (node) {
3         gNivaa++;
4         besok(node);
5         traverserPreorder(node->left);
6         traverserPreorder(node->right);
7         gNivaa--;
8     }
9 }
```

3.2 Inorder Rekursiv

```
1 void traverserInorder (Node* node) {
2     if (node) {
3         gNivaa++;
4         traverserInorder(node->left);
5         besok(node);
6         traverserInorder(node->right);
7         gNivaa--;
8     }
9 }
```

3.3 Postorder Rekursiv

```
1 void traverserPostorder (Node* node) {
2     if (node) {
3         gNivaa++;
4         traverserPostorder(node->left);
5         traverserPostorder(node->right);
6         besok(node);
7         gNivaa--;
8     }
9 }
```

4 Sorting algorithms

4.1 Common functions

```
1  /**
2   * Swaps value of two variables passed by reference(!)
3   * Function uses a template value, IE it can be used with any
4   * variable types.
5   *
6   * @param a - Value to be swapped with b
7   * @param b - Value to be swapped with a
8   */
9  template <typename T>
10 void swap(char &a, char &b) {
11     const char tmp = a;
12     a = b;
13     b = tmp;
14 }
```

4.2 Selection Sort

4.2.1 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.

4.2.2 Code

```
1  /**
2   * @param a - value to be swapped with b
3   * @param b - value to be swapped with a
4   */
5  void swap(int &a, int &b) {
6      const int tmp = a;
7      a = b;
8      b = tmp;
9  }
10
11 /**
12  * Sorts an input-array incrementally using the selection sort algorithm.
13  * @param arr - Array to be sorted
14  * @param n - Number of elements in 'arr'
15  * @see swap(...)
16  */
17 void selectionSort(int arr[], const int n) {
18     int i, j, minIndex; // Minimum Index - smallest value in interval
19
20     for (i = 0; i < n-1; i++) {
21         minIndex = i;
22         for (j = i+1; j < n; j++) {
23             if (arr[j] < arr[minIndex]){
24                 minIndex = j;
25             }
26         }
27         if (minIndex != i) {
28             swap(arr[minIndex], arr[i]);
29         }
30     }
31 }
```

4.3 Insertion Sort

4.3.1 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 høire 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.

4.3.2 Code

```
1  /**
2   * Insertion sort algorithm
3   * IMPORTANT: This uses a sentinel key at arr[0]
4   * @param arr - Array to be sorted
5   * @param n - Number of elements in 'arr'
6   */
7  void insertionSort(int arr[], int n) {
8      int i, key, j;
9      for (i = 1; i < n; i++) {
10         key = arr[i];
11         j = i - 1;
12
13         /* Move elements of arr[0..i-1], that are
14            greater than key, to one position ahead
15            of their current position */
16         while (j >= 0 && arr[j] > key) {
17             arr[j + 1] = arr[j];
18             j--;
19         }
20         arr[j + 1] = key;
21     }
22 }
```

4.4 Shellsort

4.4.1 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: aktuell 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.

4.4.2 Code:

```
1  /**
2   * Sorts an array incrementally using shellsort.
3   * IMPORTANT: This uses a sentinel key at arr[0]
4   * @param arr - Array to be sorted
5   * @param n   - Number of elements in 'arr'
6   */
7  void shellSort(char arr[], const int n) {
8      int i, j, h;
9      char tmp;
10     for (h = 1; h <= n/9; h = (3*h)+1){
11         ; // Empty for loop, initialize h-variable to adequate gapsize
12     }
13
14     while (h > 0) {
15         for (i = h+1; i < n; i++) {
16             tmp = arr[i];
17             j = i;
18         }
```



```
19     while (j > h && arr[j-h] > tmp) {
20         arr[j] = arr[j-h];
21         j -= h;
22     }
23     arr[j] = tmp;
24 }
25 // The H value will eventually reach 1.
26 // When this happens, the algorithm is identical to insertion sort
27 h /= 3;
28 }
29 }
```

4.5 Quicksort

4.5.1 Algorithm:

Prinsipp: Splitt og hersk.

- Splitt i to, og sorter hver del

Fremgangsmåte:

1. Velg en 'pivot' (gjerne midten av arrayet $\rightarrow n/2$ 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 \rightarrow venstre for pivot, og alle elementer større enn pivot \rightarrow høye for pivot.
 - (c) når pekerene møtes, plasser elementet i 'pivot' i sin korrekte plass 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 \geq verdi, og fra høyre etter \leq 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 \leq partisjonselementet, og alle elementer til høyre er \geq partisjonselementet).
5. Gjenta rekursivt.

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

4.5.2 Code:

```
1  /**
2   * Reorders an array according to the following principles:
3   *   - elements are sorted in place
4   *   - elements <= pivot element will be placed to the left of
5   *     the pivot element in the array.
6   *   - elements >= pivot element will be placed to the right of
7   *     the pivot element in the array.
8   *   - The resulting sub-arrays to the left and right of the
9   *     pivot element are not necessarily sorted.
10  *
11  * @param   arr       - Array to be partitioned
12  * @param   left      - Lower array index for partitioning
13  * @param   right     - Upper array index for partitioning
14  * @return          - The array index where the pivot element was placed
15  */
16  int partition(char arr[], const int left, const int right) {
17      if (right <= left) { return 0; }
18
19      int i = left - 1;
20      int j = right;
21      const char pivotElement = arr[right];
22
23      while (true) {
24          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 }

```

4.6 Heapsort

4.6.1 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 igjen å sortere.

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

NB!

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

4.6.2 Code:

```
1  /**
2   * Sorts an unsigned char-array incrementally with (bottom-up) heapsort.
3   * Sorterer en unsigned char-array STIGENDE med (Bottom-Up) HEAPSORT.
4   *
5   * @param  arr      - Array to be sorted
6   * @param  n        - number of elements in 'arr'
7   * @see     Heap.downHeap(...)
8   * @see     swap(...)
9   * @see     heap(...) - under datastructures
10  */
11 void heapSort(unsigned char arr[], int n) {
12     for (int keyNum = n/2; keyNum >= 1; keyNum--) {
13         gHeap.downHeap(arr, n, keyNum);
14     }
15     while (n > 1) {
16         swap(arr[1], arr[n]);
17         gHeap.downHeap(arr, --n, 1);
18     }
19 }
```

5 Searching Algorithms

5.1 Binary search - array

```
1  /**
2  *  S ker BINERT i en SORTERT array.
3  *
4  *  @param   arr      - The sorted array to be searched.
5  *  @param   target   - The value to be searched for in 'arr'.
6  *  @param   n        - Amount of elements in 'arr'.
7  *  @return  The array-index of 'arr' where 'target' was found.
8  *           0/NULL if 'target' was not found.
9  */
10 int binarySearch(const int arr[], const int verdi, const int n) {
11     int left = 1,           // Left boundary of search interval.
12         right = n,         // Right boundary of search interval.
13         middle;            // Center/middle of the search boundary.
14
15     while (left <= right) {
16         middle = (left + right) / 2;
17         if (target == arr[middle]) { return middle; }
18         if (target < arr[middle]) { right = middle - 1; }
19         else { left = middle + 1; }
20     }
21     return 0;
22 }
```

5.2 Sequential search - array

```
1  /**
2  *  Sequentially searches in an array.
3  *  Can be used on UNSORTED arrays.
4  *
5  *  @param   arr      - The sorted array to be searched.
6  *  @param   target   - The value to be searched for in 'arr'.
7  *  @param   n        - Amount of elements in 'arr'.
8  *  @return  The array-index of 'arr' where 'target' was found.
9  *           0/NULL if 'target' was not found.
10 */
11 int sequentialSearch(const int arr[], const int target, const int n) {
12     int index = n + 1;
13     while (index > 0 && target != arr[--index]) { ; } //empty loop, for finding target
14     return index;
15 }
```

6 Hashing

6.0.1 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.

6.1 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 \rightarrow plasser elementet på den plassen
 - (a) Hvis plassen ikke er ledig \rightarrow prøv $\text{plass}+1$ til du finner en ledig plass å plassere elementet

6.2 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: $6 - (\text{nøkkel} \% 6)$ - 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 \rightarrow plasser elementet i (hash1)
 - (b) Hvis ikke ledig \rightarrow sjekk $\text{hash1} + \text{hash2}$ verdi
 - i. Gjenta til vi finner en ledig skuff

6.3 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 ($\text{hash-barn1} + \text{hash-barn2}$, 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.

6.3.1 Code

```
1  /**
2   * Container-klassen Hasing.
3   *
4   * Inneholder en char-array ('tabell'), max. antall elementer i
5   * arrayen ('lengde'), samt hvilken type tabellen er av ('hType').
6   */
7  class Hashing {
8  private:
9      char* array;          ///< Pointer to the character array used for hashing
10     int length;           ///< Maximum number of elements in the array
11     HashType hType;       ///< Type of the hashing technique used
12
13     // Methods
14     int hash1(const int modValue, const int kValue);
15     int hash2(const int hashValue, const int kValue);
16     int kValue(char character);
17
18 public:
19     // Methods
20     Hashing(const HashType hT, const int len);
21     ~Hashing();
22     void display() const;
23     void insert(const int hashVerdi, const char data);
24 };
25
26 /**
27  * Simple hash function.
28  *
29  * @param modValue The modulus value for the hash calculation.
30  * @param kValue The key value to be hashed.
31  * @return The hash value computed using modValue and kValue.
32  */
33 int Hashing::hash1(const int modValue, const int kValue) {
34     return (kValue % modValue);
35 }
36
37 /**
38  * Secondary hash function.
39  *
40  * @param hashValue The initial hash value.
41  * @param kValue The key value to be hashed.
42  * @return The secondary hash value computed.
43  */
44 int Hashing::hash2(const int hashValue, const int kValue) {
45     return (hashValue - (kValue % hashValue));
46 }
47
48 /**
49  * Computes the key value from a character.
50  *
51  * @param character The character to be converted into a key value.
52  * @return The key value corresponding to the character.
53  */
54 int Hashing::kValue(char character) {
55     character = toupper(character);
56     if (character >= 'A' && character <= 'Z') {
57         return (static_cast<int>(character - 'A') + 1);
58     } else {
59         return 0;
60     }
61 }
62
63 /**
64  * Constructor for the Hashing class.
65  */
```

```

66  * @param hT The type of hash to be used.
67  * @param len The length of the hash table.
68  */
69  Hashing::Hashing(const HashType hT, const int len) {
70      length = len;
71      hType = hT;
72      array = new char[len];
73      for (int i = 0; i < length; i++) {
74          array[i] = '-';
75      }
76  }
77
78  /**
79   * Destructor for the Hashing class.
80   */
81  Hashing::~Hashing() {
82      delete [] array;
83  }
84
85  /**
86   * Displays the contents of the hash table.
87   */
88  void Hashing::display() const {
89      for (int i = 0; i < length; i++) {
90          cout << setw(3) << i << ':';
91      }
92      cout << '\n';
93
94      for (int i = 0; i < length; i++) {
95          cout << " " << array[i] << ' ';
96      }
97      cout << "\n\n";
98  }
99
100 /**
101  * Inserts a character into the hash table.
102  *
103  * @param hashValue The initial hash value to consider for insertion.
104  * @param data The character data to be inserted.
105  */
106 void Hashing::insert(const int hashValue, const char data) {
107     int dataToKVal = kValue(data);
108     int index = hash1(length, dataToKVal);
109     int addition = hash2(hashValue, dataToKVal);
110
111     while (array[index] != '-') {
112         index = (hType == LinearProbing) ? (index+1) : (index+addition);
113         index %= length;
114     }
115     array[index] = data;
116     display();
117 }

```


7 Graphs

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

7.1 Minimum Spanning Tree (MST) - Prim

7.1.1 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. $gTilknytning[k]$ 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 $gTilknytning[k]$.
5. Noder på Fringen er (i $gKantvekt$) markert med negativ vekt (USETT = -999).

7.1.2 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.

7.1.3 Orden

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

7.1.4 Code:

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

```
1  #include <iostream>           // cout
2  #include <iomanip>            // setw
3  #include "fringe.h"          // Datatypen/-strukturen "Fringe"
4  using namespace std;
5  const int NUMNODES = 8;      ///< Antall noder i grafen (V).
6  const int UNSEEN = -999;     ///< IKKE sett - "stort" negativt tall.
7  Fringe gFringe;             ///< Lager Fringe.
8  int gEdgeWeight[NUMNODES+1]; ///< 'gKantVekt[k]' er kantens
9  int gConnection[NUMNODES+1]; ///< vekt mellom 'k' og dens
10                                ///< 'gTilknytning[k]'.
11 void findMST(int num);
12 void print();
13
14 int gNeighbourMatrix[NUMNODES+1][NUMNODES+1] ///< Nabomatrise for grafen:
15 = { { 0, 0, 0, 0, 0, 0, 0, 0, 0 },          // NB: Bruker IKKE indeks 0 !!!
16     { 0, 0, 3, 2, 0, 0, 0, 1, 0 },          // A = 1    B ----- F ----- H
17     { 0, 3, 0, 0, 0, 0, 2, 0, 0 },          // B = 2    |      2      |      3      / |
18     { 0, 2, 0, 0, 0, 1, 0, 3, 0 },          // C = 3    3|      2|      / |
19     { 0, 0, 0, 0, 0, 2, 0, 3, 1 },          // D = 4    |      1      |      2/ |
20     { 0, 0, 0, 1, 2, 0, 0, 1, 0 },          // E = 5    A ----- G----- | 1
21     { 0, 0, 2, 0, 0, 0, 0, 2, 3 },          // F = 6    |      / | \      |
22     { 0, 1, 0, 3, 3, 1, 2, 0, 2 },          // G = 7    2| ----- | ----- |
23     { 0, 0, 0, 0, 1, 0, 3, 2, 0 } };        // H = 8    | / 3      1|      3      \ |
24 //      A B C D E F G H                  C ----- E ----- D
25 //                                     1          2
26
27
28 /**
29  * Hovedprogrammet:
30  */
31 int main() {
32     for (int i = 1; i <= NUMNODES; i++) {
33         gEdgeWeight[i] = UNSEEN;
34     }
35     findMST(1);
36     cout << "\n\n";
37     return 0;
38 }
39
40 /**
41  * Finner ETT minimums spennetre for en sammenhengende graf/komponent.
42  *
43  * @param num - Grafens startnode, som inni funksjonen brukes/oppdateres
44  *             til å være aktuell besøkt node
45  */
46 void findMST(int num) {
47     if (gFringe.update(num, -UNSEEN)) {
48         gConnection[num] = 0;
49     }
50
51     while (!gFringe.empty()) {
52         cout << "\n\nOPPSTART:";
53         print();
54         num = gFringe.remove();
55         gEdgeWeight[num] = -gEdgeWeight[num];
56         cout << "\nNr.1 (" << char ('A'+num-1) << ") fjernet:";
57         print();
58
59         if (gEdgeWeight[num] == -UNSEEN) {
60             gEdgeWeight[num] = 0;
61         }
62
63         for (int j = 1; j <= NUMNODES; j++) {
64             const int weight = gNeighbourMatrix[num][j];
```

```

65         if (weight > 0 && gEdgeWeight[j] < 0) {
66
67             if (gFringe.update(j, weight) ) {
68                 gEdgeWeight[j] = -weight;
69                 gConnection[j] = num;
70                 cout << "\nOppdatering:";
71                 print();
72             }
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\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 }

```

7.2 Shortest Path - Dijkstra

7.2.1 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 kalkulerede 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.

7.2.2 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'.

```
1  // *****
2  // **  CODE IS ALMOST IDENTICAL TO MST PRIMS ALGORITHM,  **
3  // **  JUST INSERT THE BELOW CODE BEFORE THE TWO FINAL IF  **
4  // **  BLOCKS IN THE FUNCTION (example below):             **
5  // **      weight += gEdgeWeight[num];                      **
6  // **                                                         **
7  // *****
8
9  /* ... */
10  weight += gEdgeWeight[num]; // <-- ADD THIS LINE
11  if (weight > 0 && gEdgeWeight[j] < 0) {
12
13      if (gFringe.update(j, weight) ) {
14          gEdgeWeight[j] = -weight;
15          gConnection[j] = num;
16          cout << "\nOppdatering:";
17          print();
18      }
19  }
20  /* ... */
```

7.3 A Star

Algoritme for å effektivt finne korteste/raskeste vei fra en startnode til en sluttnode. $A^* = \text{Dijkstra} + \text{Heuristikk}$

7.3.1 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 startnode, pluss estimert avstand til målet.

7.3.2 Orden

Tidskompleksiteten avhenger av den heuristiske funksjonen.

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

7.3.3 Framgangsmåte

1. Initialiser
 - Lag to sett/mengder: 'Open Set' og 'Closed Set'.
 - Legg til startnode til 'Open Set'.
2. Definer kostnad-funksjonene
 - $g(n)$: Kostnaden fra startnode til node n .
 - $h(n)$: Den heuristiske distansen fra node n til målet.
 - $f(n)$: Estimert total kostnad til node n fra start til mål, $g(n) + h(n)$.
3. Hovedfunksjon
 - Så lenge 'Open Set' ikke er tomt:
 - Velg den node i Open Set med lavest $f(n)$ -verdi. Kall denne node '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 nodes tidligere $g(n)$ verdi.
 - Hvis den er lavere, oppdater $g(n)$ for node.
 - 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.

7.3.4 Code - A star:

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

```
1  #include <iostream>
2  #include <cmath>
3  #include "fringe.h"
4  using namespace std;
5
6  // Variables
7  const int DIMENSION = 20;
8  const int ARRLLEN = DIMENSION * 101;
9  const int UNSEEN = -999;
10 const int TARGET_SQUARE = 1814;
11 const int START_SQUARE = 208;
12 int gEdgeWeight[ARRLEN + 1];
13 int gAssociation[ARRLEN + 1];
14 Fringe gFringe(DIMENSION * DIMENSION);
15
16 // Functions
17 bool AStar();
18 int heuristics(int nr);
19 void createAndPrintRoute();
20
21 // Gameboard
22 char gGameBoard[DIMENSION + 2][DIMENSION + 3]
23 // 0123456789012345678901
24 = { "XXXXXXXXXXXXXXXXXXXXX",
25     "X          XXXXXX      X",
26     "X              X        X",
27     "X  X              X      X",
28     "X  X              X      X",
29     "X  X              X      X",
30     "X  XXXXXXXX      XXX  X",
31     "X      X          X      X",
32     "X      XX          X      X",
33     "X      X  XXXXXXXX  X",
34     "X                                X",
35     "X  XX XXXXXXXXXX  X",
36     "X  X              X      X",
37     "X  X              XXX      X",
38     "X              X        X",
39     "X              XXXX      X",
40     "XXXXXXXXXX          X",
41     "X              XXXXXX  X",
42     "X  XXXX          X      X",
43     "X  XXXX          XXX  X",
44     "X                                X",
45     "XXXXXXXXXXXXXXXXXXXXX" };
46
47 /**
48  * The main program.
49  */
50
51 int main() {
52     for (int i = 101; i <= (ARRLEN); i++) {
53         gEdgeWeight[i] = UNSEEN;
54     }
55
56     if (AStar()) {
57         createAndPrintRoute();
58     } else {
59         cout << "\n\n\tUnable to find a path from " << START_SQUARE
60              << " to " << TARGET_SQUARE << "\n\n";
61     }
62     return 0;
63 }
64
```

```

65  /**
66   * Tries efficiently and quickly to find a short path between two squares in a grid.
67   * Utilizes the A* pathfinding algorithm, combining Dijkstra's algorithm and a heuristic.
68   * The algorithm is often used in games and artificial intelligence for grid-based pathfinding.
69   *
70   * @return Whether a path was found from START_SQUARE to TARGET_SQUARE or not.
71   * @see     heuristics(...)
72   */
73  bool AStar() {
74      int i,
75          nr = START_SQUARE,
76          neighbor,
77          x,
78          y,
79          weight,
80          additional;
81
82      gFringe.update(nr, -UNSEEN);
83      gAssociation[nr] = 0;
84      gEdgeWeight[nr] = 0;
85
86      while (!gFringe.empty()) {
87          nr = gFringe.remove();
88          if (nr == TARGET_SQUARE) { return true; }
89
90          gEdgeWeight[nr] = -gEdgeWeight[nr];
91
92          // Check all 8 potential neighbors (including diagonals)
93          for (i = 1; i <= 8; i++) {
94              // Determine the neighbor based on the current index
95              switch (i) {
96                  case 1: neighbor = nr-100; additional = 2; break; // Up
97                  case 2: neighbor = nr-99; additional = 3; break; // Diagonal Up-Right
98                  case 3: neighbor = nr+1; additional = 2; break; // Right
99                  case 4: neighbor = nr+101; additional = 3; break; // Diagonal Down-Right
100                 case 5: neighbor = nr+100; additional = 2; break; // Down
101                 case 6: neighbor = nr+99; additional = 3; break; // Diagonal Down-Left
102                 case 7: neighbor = nr-1; additional = 2; break; // Left
103                 case 8: neighbor = nr-101; additional = 3; break; // Diagonal Up-Left
104             }
105
106             // Convert the neighbor square's number to grid coordinates
107             x = neighbor % 100; y = neighbor / 100;
108
109             // If the neighbor is not a wall ('X') and is unvisited
110             if ((gGameBoard[y][x] != 'X') && (gEdgeWeight[neighbor] < 0)) {
111                 weight = gEdgeWeight[nr] + additional;
112
113                 // If this is a new path or a better one, update the fringe
114                 if (gFringe.update(neighbor, weight + heuristics(neighbor))) {
115                     gEdgeWeight[neighbor] = -weight;
116                     gAssociation[neighbor] = nr;
117                 }
118             }
119         }
120     }
121     return false;
122 }
123
124 /**
125  * Calculates the "straight-line distance"
126  * (the hypotenuse in a triangle, i.e., Euclidean distance) between two squares.
127  *
128  * @param nr - Square number/ID
129  * @return Calculated "straight-line" from square 'nr' to TARGET_SQUARE
130  */
131 int heuristics(int nr) {
132     int dx = ((nr % 100) - (TARGET_SQUARE % 100));

```

```

133     int dy = ((nr / 100) - (TARGET_SQUARE / 100));
134     float straightLine = sqrt((dx * dx) + (dy * dy));
135     return (2 * straightLine);
136 }
137
138 /**
139  * Finds the path that has been taken, "draws" it on the grid, and prints this.
140  */
141 void createAndPrintRoute() {
142     int i, j, nr;
143     cout << "\n\n";
144
145     gGameBoard[START_SQUARE / 100][START_SQUARE % 100] = 'S';
146     gGameBoard[TARGET_SQUARE / 100][TARGET_SQUARE % 100] = 'M';
147
148     nr = gAssociation[TARGET_SQUARE];
149     while (gAssociation[nr] != 0) {
150         gGameBoard[nr / 100][nr % 100] = '.';
151         nr = gAssociation[nr];
152     }
153
154     for (i = 0; i < DIMENSION + 2; i++) {
155         for (j = 0; j < DIMENSION + 2; j++)
156             cout << gGameBoard[i][j];
157         cout << '\n';
158     }
159 }

```


7.4 Union Find

7.4.1 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.

7.4.2 Code - Union find:

```

1  /**
2   *   Progameksempel nr 36 - Union-Find.
3   *
4   *   Noen ganger er spørsmålet om node A er i samme komponent / (sub)graf /
5   *   set / ekvivalensklasse som node B (path'en imellom er uinteressant).
6   *   NB: Det bygges IKKE en lignende graf, men et tre/flere trær av de som
7   *       er i samme komponent/subgraf.
8   *
9   *   Funksjonen 'unionerUgFinn' setter noder til å være i samme komponent
10  *   om 'unioner' er lik 'true', dvs. en unionering skal skje.
11  *   Er 'unioner' lik 'false' er det interessant hva funksjonen returnerer,
12  *   dvs. om nodene allerede befinner seg i samme komponent eller ei.
13  *
14  *
15  *   Algoritme/virkemåte:
16  *   gForeldre[i] > 0 (=x) når node nr.'i' har 'x' som foreldre/mor
17  *   gForeldre[i] = 0      når node nr.'i' ennå ikke har noen foreldre,
18  *                       eller ender opp som rot for et tre
19  *   unioner = true       om noder skal knyttes sammen
20  *   = false            om det skal finnes ut om noder er i samme komponent
21  *
22  *   @file      EKS_36_UnionFind.CPP
23  *   @author    Frode Haug, NTNU
24  */
25
26
27  #include <iostream>
28  #include <iomanip>           // setw
29  using namespace std;
30  const int NUMNODES = 10;    ///< Nodene har 'ID' lik 'A'-'J' (1-10).
31  const int NUMEDGES = 14;    ///< Antall kanter i grafen.
32  int gParents[NUMNODES+1];   ///< I "skuff" nr.i er foreldre til nr.i.
33                               ///< Grafkantene:
34  char gEdges[NUMEDGES][3] = { "AB", "CG", "JI", "AJ", "BD", "HB", "DC",
35                                "DE", "GE", "FE", "CH", "IH", "BJ", "BC" };
36
37  /* Grafen ser slik ut:
38
39      A-----B-----D
40      |       / | \   /   \
41      |      /  |  \ /     \
42      |     /   |   \ /      \
43      |    /    |    \ /       \
44      J-----I-----H         G
45
46      C-----E-----F
47
48      (Note: The diagram shows a complex graph structure with nodes A-J and edges between them.)
49
50  */
51
52  void skriv();
53  bool unionFind(int nr1, int nr2, const bool unioned);
54
55  /**
56   *   Hovedprogrammet:
57   */
58  int main() {
59      int nr1 = 0, nr2 = 0;
60      char tegn = ' ';
61
62      for (int i = 0; i < NUMEDGES; i++) {
63          nr1 = static_cast<int>(gEdges[i][0] - 'A' + 1);
64          nr2 = static_cast<int>(gEdges[i][1] - 'A' + 1);
65          unionFind(nr1, nr2, true);
66          cout << '\n' << gEdges[i][0] << ' ' << gEdges[i][1] << ':';
67          skriv();
68          cout << "\t\t'D' og 'H' er " << (!unionFind(4, 8, false) ? "IKKE " : "")
69               << "i samme komponent";
70      }
71  }

```

```

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 }

```

7.4.3 Union Find with Weight Balancing and Path Compression

```
1  /**
2   * If 'unioned' is 'true', nodes 'nodeIndex1' and 'nodeIndex2' will be set to be
3   * in the same component, otherwise it returns whether they already are in the same component.
4   *
5   * Identical to 'unionFind(...)' in previous example except that
6   * the code also includes Path Compression (PC) and Weight Balancing (WB).
7   *
8   * @param nodeIndex1 - Index for the 1st node (becomes parent of or child to 'nodeIndex2')
9   * @param nodeIndex2 - Index for the 2nd node (becomes parent of or child to 'nodeIndex1')
10  * @param unioned     - Should the nodes end up in the same component or not
11  * @return Are 'nodeIndex1' and 'nodeIndex2' in the same component or not?
12  */
13 bool unionFind(int nodeIndex1, int nodeIndex2, const bool unioned) {
14     int i = nodeIndex1,
15         j = nodeIndex2;
16     while (gParents[i] > 0) { i = gParents[i]; }
17     while (gParents[j] > 0) { j = gParents[j]; }
18
19     // *****
20     // NEW (down to star line) compared to regular union find. Path Compression (PC):
21     int index = 0;
22     while (gParents[nodeIndex1] > 0) {
23         index = nodeIndex1;
24         nodeIndex1 = gParents[nodeIndex1];
25         gParents[index] = i;
26     }
27
28     while (gParents[nodeIndex2] > 0) {
29         index = nodeIndex2;
30         nodeIndex2 = gParents[nodeIndex2];
31         gParents[index] = j;
32     }
33
34     // *****
35     // NEW (down to star line) compared to regular union find. Weight Balancing (WB):
36     if (unioned && (i != j)) {
37         if (gParents[j] < gParents[i]) {
38             gParents[j] += gParents[i]-1;
39             gParents[i] = j;
40         } else {
41             gParents[i] += gParents[j]-1;
42             gParents[j] = i;
43         }
44     }
45     // *****
46
47     return (i == j);
48 }
```

7.5 fringe.h

```
1  #ifndef __FRINGE_H                // For at evt. bare skal includes EN gang:
2  #define __FRINGE_H
3
4  /**
5   * Enum to specify if the Fringe's key should be printed as a character or a number.
6   */
7  enum PrintType { Character, Number};
8
9  /**
10   * Class representing a Fringe, which contains the number of elements in each array
11   * and two arrays for storing the key/ID/data of a node and the lowest weight
12   * of the edge leading to this node found so far.
13   */
14  class Fringe {
15  private:
16      int* keys;                ///< 'keys[i]' is the nodes so far encountered.
17      int* weights;            ///< 'weight[i]' is the weight for connecting the node at 'keys[i]'.
18      int indexAmount;          ///< Last index used in both 'keys' and 'weights'.
19                               ///< incremented for every node encountered
20  public:
21      Fringe(const int max = 200);
22      ~Fringe();
23      void display(const PrintType type) const;
24      bool empty() const;
25      bool update(const int key, const int weight);
26      int remove();
27  };
28
29  /**
30   * Constructor for Fringe class. Initializes arrays to store keys and weights.
31   * @param max The maximum size of the arrays.
32   */
33  Fringe::Fringe(const int max) {
34      keys = new int[max];
35      weights = new int[max];
36      indexAmount = 0;
37  }
38
39  /**
40   * Destructor for Fringe class. Deallocates memory used for keys and weights arrays.
41   */
42  Fringe::~Fringe() {
43      delete [] keys;
44      delete [] weights;
45  }
46
47  /**
48   * Displays the contents of the Fringe.
49   * @param type The PrintType specifying how to display keys (as characters or numbers).
50   */
51  void Fringe::display(const PrintType type) const {
52      std::cout << "\tFringe:\t";
53      for (int i = 0; i < indexAmount; i++) {
54          if (type == Character) {
55              std::cout << char(keys[i]+'A'-1);
56          } else {
57              std::cout << keys[i];
58          }
59          std::cout << ':' << weights[i] << " ";
60      }
61  }
62
63  /**
64   * Checks if the Fringe is empty.
65   * @return True if empty, otherwise false.
```

```

66  */
67  bool Fringe::empty() const {
68      return (indexAmount == 0);
69  }
70
71  /**
72   * Updates the Fringe with a new node and its weight. Adds the node if it's new,
73   * or updates the weight if it's lower than the existing one.
74   * IMPORTANT: The node is added before any existing nodes with the same weight.
75   *
76   * @param key      - The key/ID/data of the node to be added.
77   * @param weight   - The weight of the edge to this node.
78   * @return         - True if the node was added or updated, otherwise false.
79   */
80  bool Fringe::update(const int key, const int weight) {
81      int j;
82      int i = 0;
83
84      // Search for the node's key in the existing keys.
85      // If found, check if the new weight is not lower than the existing weight.
86      while ((keys[i] != key) && (i < indexAmount)) {
87          i++;
88      }
89
90      // If the node exists and the new weight is not lower, do not update.
91      if ((i < indexAmount) && (weight >= weights[i])) {
92          return false;
93      }
94
95      // If the node exists and the new weight is lower, remove the old node.
96      if (i < indexAmount) {
97          for (j = i; j < indexAmount-1; j++) {
98              keys[j] = keys[j+1];
99              weights[j] = weights[j+1];
100          }
101          --indexAmount;
102      }
103
104      // Find the position where the new or updated node should be inserted.
105      i = 0;
106      while ((weights[i] < weight) && (i < indexAmount)) {
107          i++;
108      }
109
110      // Increase the size of the Fringe to accommodate the new node.
111      ++indexAmount;
112
113      // Shift existing nodes to make space for the new node.
114      for (j = indexAmount-1; j > i; j--) {
115          keys[j] = keys[j-1];
116          weights[j] = weights[j-1];
117      }
118
119      // Insert the new or updated node.
120      keys[i] = key;
121      weights[i] = weight;
122      return true;
123  }
124
125  /**
126   * Removes and returns the first element in the Fringe.
127   * @return         - The key of the first element.
128   */
129  int Fringe::remove() {
130      int key = keys[0];
131      for (int i = 0; i < indexAmount-1; i++) {
132          keys[i] = keys[i+1];
133          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.

8 Datastructures

8.1 Heap

8.1.1 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
```

8.1.2 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.

8.1.3 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).

8.1.4 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
```

8.1.5 Code - Heap class:

```
1  /**
2   *   Progameksempel nr 25 - Heap (prioritetskø) - selulaget enkel klasse.
3   *
4   *   Eksemplet viser en selulaget implementasjon av container-klassen Heap.
5   *   Det er laget kode for følgende funksjoner:
6   *
7   *   - Heap(const int lengde = 200)
8   *   - ~Heap
9   *   - void change(const int keyNr, const T nyVerdi)      // Oppgave nr.15
10  *   - void display()
11  *   - void downHeap(T arr[], const int ant, int keyNr)
12  *   - void extract(const int keyNr)                      // Oppgave nr.15
13  *   - void insert(const T verdi)
14  *   - T    remove()
15  *   - T    replace(const T verdi)
16  *   - void upHeap(int keyNr)
17  *
18  *   For mer forklaring av prinsipper og koden, se:  Heap.pdf
19  *
20  *   Orden ( O(...)):
21  *       Alle operasjonene (insert, remove, replace, upHeap, downHeap,
22  *       change og extract) krever færre enn 2 lg N sammenligninger
23  *       når utført på en heap med N elementer.
24  *
25  *   NB: - Det er bare funksjonene 'insert' og 'remove' som er laget
26  *       litt robuste. De andre ('change', 'extract' og 'replace')
27  *       er IKKE det.
28  *       - Heapen i denne koden fungerer ut fra at det er STØRSTE element
29  *       som skal være i element nr.1. Men, koden kunne enkelt ha vært
30  *       omskrevet slik at den i stedet fungerer for MINSTE element.
31  *       - Er en .h-fil, da skal includes og brukes av EKS_26_HeapSort.CPP
32  *
33  *   @file      EKS_25_Heap.H
34  *   @author    Frode Haug, NTNU
35  */
36
37
38 #include <iostream>          // cout
39 #include <limits>           // numeric_limits::max
40
41
42 /**
43  *   Container-klassen Heap.
44  *
45  *   Inneholder en array ('data') av typen 'T', to int'er som angir heapens
46  *   max.lengde og nåværende antall elementer i arrayen, samt en sentinel key.
47  */
48 template <typename T>
49 class Heap {
50 private:
51     int lengde, antall;
52     T sentinelKey;
53     T* data;
54     void upHeap(int keyNr);
55
56 public:
57     Heap(const int len = 200); // constructor
58     ~Heap();                  // destructor
59     void change(const int keyNr, const T nyVerdi);
60     void display();
61     void downHeap(T arr[], const int ant, int keyNr);
62     void extract(const int keyNr);
63     void insert(const T verdi);
64     T remove();
65     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

```

```

134 /**
135  * @brief Adjusts the position of a value in the heap to maintain heap property.
136  *
137  * This method is used during deletion operations. It moves the value at the given
138  * index downwards in the heap until the heap property is restored.
139  *
140  * @param arr The heap array.
141  * @param ant The number of elements in the heap.
142  * @param keyNr The index of the value to move down.
143  */
144 template <typename T>
145 void Heap<T>::downHeap(T arr[], const int ant, int keyNr) {
146     int j;
147     T verdi = arr[keyNr];
148     while (keyNr <= ant/2) {
149         j = 2 * keyNr;
150         if (j < ant && arr[j] < arr[j+1]) { j++; }
151         if (verdi >= arr[j]) { break; }
152         arr[keyNr] = arr[j];
153         keyNr = j;
154     }
155     arr[keyNr] = verdi;
156 }
157
158 /**
159  * @brief Extracts a value from the heap.
160  *
161  * The method for extracting values from the heap. Intended to be
162  * implemented in relation to specific task requirements.
163  *
164  * @param keyNr The index of the value to extract.
165  */
166 template <typename T>
167 void Heap<T>::extract(const int keyNr) {
168     // LAG INNMATEN ifm. Oppgave nr.15
169 }
170
171 /**
172  * @brief Inserts a new value into the heap.
173  *
174  * Adds a new value to the heap and then re-heapifies to maintain the heap property.
175  * If the heap is full, displays an error message.
176  *
177  * @param verdi The value to insert.
178  */
179 template <typename T>
180 void Heap<T>::insert(const T verdi) {
181     if (antall < lengde - 1) {
182         data[++antall] = verdi;
183         upHeap(antall);
184     } else {
185         std::cout << "\nHeapen er full med " << lengde << " elementer (inkl. sentinel key)!\n\n";
186     }
187 }
188
189 /**
190  * @brief Removes and returns the top element of the heap.
191  *
192  * Removes the top element (maximum or minimum based on heap type), re-heapifies, and returns
193  * the removed element. If the heap is empty, displays an error message and returns the sentinel key.
194  *
195  * @return The removed top element of the heap.
196  */
197 template <typename T>
198 T Heap<T>::remove() {
199     if (antall > 0) {
200         T verdi = data[1];
201         data[1] = data[antall--];

```

```

202     downHeap(data, antall, 1);
203     return verdi;
204 } else {
205     std::cout << "\nHeapen er helt tom - ingenting i 'remove'!\n\n";
206     return sentinelKey;
207 }
208 }
209
210 /**
211  * @brief Replaces the top element of the heap with a new value.
212  *
213  * Replaces the top element of the heap with a given value and then re-heapifies.
214  * This operation combines removal and insertion into a single step.
215  *
216  * @param verdi The value to replace the top element with.
217  * @return The replaced top element of the heap.
218  */
219 template <typename T>
220 T Heap<T>::replace(const T verdi) {
221     data[0] = verdi;
222     downHeap(data, antall, 0);
223     return data[0];
224 }

```