UMEÅ UNIVERSITY
Department of Computing Science
Assignment report

4 juni 2021

5DV149 - OU5, Version 2.0

Data Structures and Algorithms (C) Spring 2021, 7.5 Credits

Finns en väg?

Name	Johannes Strid, Otto Nilsson, Måns Lundmark
CAS-ID	tfy18jsd, tfy18onn, tfy18mlk

Innehåll

1 Introduktion

Vi har fått en god grund i hur man kan hantera listor och en-dimensionella fält som har stor potential i att lösa många olikaartade problem och idéer. Ett problem med dessa datatyper är att de löser för linjärt ordnade element. Så vad kan vi göra ifall vi vill lösa ett problem som saknar just detta. Ett alternativ är att använda sig av datatypen Graf, som varken kräver att elementen är linjärt ordnade eller har någon som helst ordningsrelation. När inga enklare lösningar duger så vänder man sig ofta till Graf. Detta öppnar därför upp för ännu mer olika tillämpningar och användningsområden. Exempel på användningsområden där Graf fungerar bra som modell kan vara för bilkartor, flygkartor, elektriska kretsar eller övriga icke ordnade programmodeller[1]. Syftet med denna uppgift är att skriva ett huvudprogram och två olika graf-lösningar som ska fungera tillsammans med givna existerande datatypsimplementationer. Programmet ska besvara frågor ifall det finns någon väg mellan olika noder motsvarande svenska flygplatser.

2 Användarhandledning

Programmets huvuduppgift är att kontrollera ifall det finns länkade flyg mellan 6 stycken flygdestinationer stadda i Sverige. Det första som händer när koden körs är att den ber användaren om att ange ursprung (origin) samt destination (destination) eller om programmet önskas att avslutas genom att texta quit. För att ange flygplatserna på ett korrekt sätt skriver man flygplatsen med en förkortning av tre stora bokstäver där Brommas flygplats i Stockholm förkortas BMA eller Umeås UME, se alla förkortningar och befintliga flygplatser ansatta i koden under appendix [A1]. Dessa flygplatser ska läsas in av programmet igenom att man kör koden med en map fil airmap1.map. Ifall någon annan destination eller förkortning skrivs in returnerar programmet det första felaktiga sök ordet med ett meddelande som informerar användaren om att noden inte existerar och att försöka igen. Alla flygplatser är inte sammanlänkade utan de flygplatser som är sammanlänkade är Pajalas med Luleås och Umeås med Bromma, Göteborg och Malmö. Ifall användaren skriver in ursprung och destination som inte har någon länk returnerar programmet ett meddelande om att det saknas anslutning samt att försöka igen. Om korrekta förkortningar används och det existerar en länk emellan destinationerna returneras en bekräftelse och programmet frågar om att ange destination på nytt tillsammans med information hur man avslutar. När programmet avslutad med quit skrivs en verifikation om att det avslutats på korrekt sätt.

Här kommer ett exempel på en körning av programmet för en uppsättning noder och bågar enligt Appendix[A1].

1. Vid körning av en korrekt fil kommer programmet fråga efter `origin` och `destination` enligt

```
PS C:\Users\Otto\Documents\C\DoA\OU5HANDIN> ./is_connected 1-airmap1.map
Enter origin and destination (quit to exit):
```

2. Om användaren svarar med en eller flera noder som inte existerar svarar programmet med

```
Enter origin and destination (quit to exit): Felnod UME
Node Felnod does not exist, try again!
```

3. Om användaren svarar med två noder som existerar(enligt formatet `"<Nod 1> <Nod 2>"`) letar programmet efter en väg mellan dessa två noder och returnerar ett meddelande beroende på om det finns en väg eller inte. Nedan visas utskrifter för både fallet då det finns en väg och när det inte gör det.

```
Enter origin and destination (quit to exit): UME BMA
There is a path from UME to BMA.

Enter origin and destination (quit to exit): BMA LLA
There is no path from BMA to LLA.
```

4. Om användaren vill avsluta programmet skriver hen `quit` och programmet avslutas med följande utskrift.

```
Enter origin and destination (quit to exit): quit
Normal exit.
```

3 Systembeskrivning

Vi har i denna labb gjort två olika implementationer av graf med hjälp av listor respektive endimensionella fält. Det fullständiga gränssnittet givet av `graph.h` är Funktionsspecifikationen ges av

Utöver dessa funktioner har grafimplementationen även en funktion **nodes_are_equal(n1, n2)** som jämför noderna $n1$ och $n2$ och returnerar 1 om de är lika och 0 om de är inte är det. Då vi har två olika implementationer av graf kommer dessa behandlas separat. För att se gränsytan till datatypen

Abstract Datatype	**Graph(node, edge)**
graph_empty	**( )** → **Graph(node, edge)**
graph_is_empty	(g : **Graph(node, edge)**) → **Bool**
graph_has_edges	(g : **Graph(node, edge)**) → **Bool**
graph_insert_node	(s : **node name**, g : **Graph(node, edge)**) → **Graph(node, edge)**
graph_find_node	(s : **node name**, g : **Graph(node, edge)**) → v : **node**
graph_node_is_seen	(v : **node**, g : **Graph(node, edge)**) → **Graph(node, edge)**
graph_node_set_seen	(v : **node**, g : **Graph(node, edge)**) → **Graph(node, edge)**
graph_reset_seen	(g : **Graph(node, edge)**) → **Graph(node, edge)**
graph_insert_edge	(e : **edge**, g : **Graph(node, edge)**) → **Graph(node, edge)**
graph_delete_node	(v : **node**, g : **Graph(node, edge)**) → **Graph(node, edge)**
graph_delete_edge	(v1,v2 : **node**, g : **Graph(node, edge)**) → **Graph(node, edge)**
graph_choose_node	(g : **Graph(node, edge)**) → **node**
graph_neighbours	(v : **node**, g : **Graph(node, edge)**) → **List(val)**
graph_kill	(g : **Graph(node, edge)**) → **( )**
graph_print	(g : **Graph(node, edge)**) → **( )**

graph see appendix [A3] Grafstrukturen har vi definierat som en lista(eller endimensionellt fält) med element av datatypen `node` som i sig har tre olika element:

- `identifier` som är en pekare till ett objekt av datatypen **const char**. Denna textsträng är det vi använder som nodnamn och är således det vi använder för att identifiera, söka och jämföra noder.
- `neighbours` som är en pekare till en lista/endimensionellt fält av datatypen **dlist**/**array_1d**. Denna nästlade lista/fält innehåller nodnamnen på alla grannar till noden; d.v.s. namnen på alla noder dit det går en båge från noden. Notera att den nästlade listan inte har element av datatypen nod utan enbart textsträngar med nodnamnen!
- `seen_status` som är ett objekt av datatypen **bool**. Detta objekt har alltså antingen värdet 1 eller 0, där 1 säger att noden har undersökts vid ett tidigare tillfälle och 0 säger att den inte är undersökt. Detta används i programmet `is_connected.c` för att optimera sökningen genom att inte söka igenom samma nod flera gånger.

3.1 Graf implementerad med hjälp av Lista

Den första implementationen `graph.c` är konstruerad med hjälp av datatypen Lista, för att se gränsytan till denna implementation, se appendix [A3]. Den innehåller alla funktioner som är beskrivna i gränsytan för datatypen graf, trots att funktionerna **graph_delete_edge** och **graph_print** är överflödiga för just detta problem.

Funktionen **graph_empty()** allokerar minne för grafen samt skapar en tom lista för noderna.

Funktionen **graph_is_empty(g)** anropar `dlist`-funktionen **dlist_is_empty** med listan av noder.

Funktionen **graph_has_edges(g)** itererar genom hela listan av noder för grafen g och avläser en nod vid varje iteration. Den avlästas nodens grannar skickas sedan in i `dlist`-funktionen **dlist_is_empty** för att avgöra om listan är tom. Om listan inte är tom betyder det att det finns en båge varpå funktionerna omedelbart returnerar värdet 1. Om ingen båge hittas i någon av noderna returnerar funktionen 0.

Funktionen **graph_insert_node(s, g)** allokerar minne för en `node`. Textsträngen s skrivs sedan över till nodens `identifier`, dess `seen_status` sätts till `false` och minne allokeras för en lista av eventuella grannar. Sedan sätts noden in i grafen g med hjälp av **dlist_insert** och grafen returneras.

Funktionen **graph_find_node(s, g)** itererar genom hela listan av noder för grafen g och avläser en nod vid varje iteration. Den avlästa nodens `identifier` jämförs med s med hjälp av den inbyggda C-funktionen **strcmp**. Om **strcmp** returnerar `true` returneras den inspekterade noden. Om ingen s inte matchar någon nods `identifier` returnerar funktionen `NULL`.

Funktionen **graph_node_is_seen(v, g)** returnerar `seen_status` för noden v.

Funktionen **graph_node_set_seen(v, g, seen)** ändrar `seen_status` för noden `g` till `seen` och returnerar grafen `g`.

Funktionen **graph_reset_seen(g)** itererar genom hela grafen `g`, läser av respektive nod med **dlist_is_empty** och sätter dess `seen_status` till 0. Slutligen returneras grafen `g`.

Funktionen **graph_insert_edge(n1, n2, g)** inleder med att läsa av grannlistan `neighbours` för `n1` samt nodnamnet `identifier` för `n2`. Sedan sätts nodnamnet in i grannlistan och grafen `g` returneras.

Funktionen **graph_delete_node(v, g)** itererar genom grafen `g` och vid varje iteration avläses en nod som jämförs med `v` genom funktionen **nodes_are_equal**. Om noderna är lika tas den avlästa noden bort från listan av noder och nodens samt dess grannlistas minne frigörs och grafen `g` returneras.

Funktionen **graph_choose_node(g)** läser av och returnerar den noden som ligger överst i nodlistan för `g`.

Funktionen **graph_neighbours(n, g)** inleder med att skapa en tom lista med hjälp av **dlist_empty** och läser av grannlistan `neighbours` till `n`. Sedan itererar funktionen över grannlistan och tar fram hela noden för varje granne med hjälp av **graph_find_node** och skriver över denna till den tomma listan. När alla grannars respektive noder har skrivits över till den tomma listan returnerar funktionen listan. Notera att grafimplementationen inte ansvarar för minnet som allokeras för denna lista och minnet måste således frigöras av användaren.

Funktionen **graph_kill(g)** går igenom hela listan av noder och vid varje iteration läses den översta noden av med **graph_choose_node** som sedan raderas med **graph_delete_node**. Slutligen raderas listan för **dlist_kill** och minnet för `g` frias.

3.2 Graf implementerad med hjälp av endimensionellt fält

Den andra graf implementationen, graph2.c, är konstruerad med hjälp av en endimensionell array se appendix [A5] för den endimensionella arrayens gränsyta.

Funktionen **graph_empty()** allokerar först minne för grafen och sedan skapas en array av noder.

Funktionen **graph_is_empty(g)** anropar **array_1d**-funktionen **array_1d_has_value()**. Denna funktion returnerar `TRUE` om arrayen har värde, och false om inte.

Funktionen **graph_has_edges(g)** inleds först med att skapa en array av noder. Därefter itererar funktionen igenom arrayen och avläser en nod som motsvarar indexet nuvarande loop i arrayen. Därefter skapas även en array av grannar och i det fallet då **array_1d_inspect_value(neighbours,index)** ger `TRUE` returneras 1. Om inte neighbours-arrayen har har värde returneras 0.

Funktionen **graph_insert_nodes(s, g)** tar in textsträngen s och grafen g. Där inne definieras en nod och dess identifier samt neighbours vars neighbours är bildad som en array. Därefter itererar den genom de olika noderna och skjuter in värden i grann-arrayen. Därefter sätts **seen_status** till `false` och detta sätts sedan in i noderna.

Funktionen **graph_find_node(s,g)** itererar över arrayen av noder för grafen g och avläser en nod för varje iteration. Därefter jämförs den inspekterade identifier hos noden med text-strängen som funktionen tar in med hjälp av **strcmp()**. För fallet då den inspekterade identifier och textsträngen är lika så returneras noden som inspekterades.

Funktionen **graph_node_is_seen(n,g)** returnerar **seen_status** för noden n.

Funktionen **graph_node_set_seen(g,n,seen)** tar in tre inparametrar, grafen g, noden n och **seen_status** seen. Denna returnerar den modiferade grafen med ändrad **seen_status**.

Funktionen **graph_reset_seen(g)** itererar över en en array av noder som skapas med och för varje inspekterad nod sätts dess **seen_status** till false. Därefter returneras den modifierade grafen.

Funktionen **graph_insert_edge(n1,n2,g)** tar in två noder, n1 och n2 samt grafen g. Därefter itererar den igenom en array av noder. I varje iteration inspekteras en nod med motsvarande iterationsindex och jämför med de angivna noderna. Ifall noderna stämmer överens med de angivna så sätts en båge mellan dem. Därefter returneras den mofiderade grafen.

Funtionen **graph_delete_node(n,g)** skapas först en array av grannar tillhörande den angivna noden n. Därefter itererar den över arrayens alla noder och inspekterar en nod varje iteration och därefter frigörs det elementet. Därefter dödas arrayen med **array_1d_kill()** och till slut frigörs allokerat minne som använts av noden.

Funktionen **graph_choose_node(g)** läser av och returnerar noden på första indexplatsen i arrayen.

Funktionen **graph_neighbours(n,g)** inleder med att skapa en tom lista med hjälp av **dlist_empty()** och läser av grann- och nod-arrayen. Därefter itererar den genom alla noder och ifall **array_1d_inspect_value()** returnerar ickefalskt så skapas en ny nod som sedan läggs till i listan av grannar. När grannars respketive noder har tillförts i den tomma listan returneras den nu icke-tomma grann-listan.

Funktionen **graph_kill(g)** skapar först en array av noder som den sedan itererar genom. I varje iteration skapas en ny nod med hjälp av **array_1d_inspect_value()** och därefter anropas **graph_delete_node()** för den inspekterade noden. När detta gjorts för samtliga noder så används sedan **array_1d_kill()** på noderna och därefter frigörs allt allokerat minne från grafen.

4 Algoritmbeskrivning

Funktion **first_non_white_space**

- 1. Upprepa tills stränge när slut eller tills `s`(index) inte är blanksteg.
  - 1.1 Uppdatera index
- 2. Returnera index om strängen inte tagit slut, annars -1.

För funktionen **last_non_white_space(s)**

- 1. Upprepa tills $0 \leq$ `i` samtidigt som strängen är ett blanksteg.
- 2. Om $0 \leq$ `i`
  - Returnera `i`.
- 3. Annars returnera -1.

Funktion **number_of_strings**

- 1. Itererar över textsträng mellan första och sista icke-blanksteget.
  - 1.1 Räkna antalet blanksteg
- 2. Returnera antalet strängar.

Funktion **remove_comment(s)**

- 1. Upprepa tills strängen inte är kommentar samt iterationstal mindre än längden av strängen.
  - 1.1 Spara nuvarande del av textsträngen
- 2. Returnera modifierad textsträng.

Funktion **line_has_one_string**

- 1. Returnera sant ifall linjen har en sträng.

Funktion **line_is_blank**

- 1. Returnerar sant om textsträngen är blanksteg.

Funktion **line_is_comment(s)**

- 1. Returnera sant om platsen i strängen är samtidigt som strängen är en kommentar.

Funktion **white_space(s)**

- 1. Upprepa tills strängen inte är `NULL` och strängen inte är blanksteg.
  - 1.1 uppdatera index.
- 2. om strängen inte är `NULL`

  - 2.2 Returnera index.
- 3. Annars returnera -1.

Funktion **count_white_spaces**

- 1. Upprepa tills strängen `s` är slut.
  - 1.1 Uppdatera räkningen om nuvarande plats i strängen är ett blanksteg.
- 2. Returnera räkning.

Funktion **trim**

- Ta bort inledande och avslutande blanksteg samt alla kommentarer från strängen.

Funktion **build_graph**

- 1. Upprepa över alla noder.
  - 1.1 Om start respektive destinations nod inte finns i grafen.
    - 1.1.1 Sätt in en noderna i grafen.
  - 1.2 Tag ut de ovan insatta noderna.
  - 1.3 Om båda noderna samtidigt är skilda från `NULL`
    - 1.3.1 Sätt in en båge mellan noderna.
- 2. Returnera grafen.

Funktion **find_path**

- 1. Upprepa till kö av grannar är tom.
  - 1.1 Läs av översta noden från kön.
  - 1.1 Returnera 1 om den inspekterade noden är samma som destinationen.
  - 1.2 Avlägsna inspekterad nod från kö.
  - 1.3 Läs av grannlista från inspekterad nod.
  - 1.3 Upprepa tills grannlistan är slut.
    - 1.3.1 Om inspekterad nod inte är sedd
      - 1.3.1.1 Markera noden som sedd samt lägg till den i kön.
  - 1.4 Frigör minne för grannlistan.
- 2. Frigör minne för kö, återställ sedd-status för alla noder och returnera 0.

För att se gränsytan till datatypen **queue** se appendix [A4].
Funktion **main**

- 1. Läs av inläst fil. Avsluta om filen är tom.
- 2. Iterera över alla rader i avlästa filen.
  - 2.1. Hoppa till nästa rad om avlästa raden enbart är kommentar eller blanksteg.
  - 2.2. Om detta är *första* raden som inte är enbart kommentar eller blanksteg.
    - 2.2.1. Avsluta programmet om raden består av flera strängar eller om den inte är ett tal.
  - 2.3. Hoppa till nästa rad om det bara finns en enskild sträng i raden.
  - 2.4. Ta bort alla blanksteg i början/slutet samt kommentarer i raden.
  - 2.5. Avsluta programmet om antalet separata strängar i raden inte är 2.
  - 2.6. Spara den modifierade raden.
- 3. Initiera graf med datan modifierad enligt ovan.
- 4. Upprepa tills användaren skrivet '`quit`'.
  - 4.1. Läs av ursprung och destination från användaren.
  - 4.2. Be användaren om nytt indata om ursprung eller destination inte existerar i grafen.
  - 4.3. Titta om det finns någon koppling mellan ursprung och destination m.h.a. **find_path**.
- 5. Frigör minne.

5 Informationsflöde

När användaren ska exekvera programmet så körs den enligt `./is_connected mapfile.map` där **is_connected** är skriptet som hanterar test och körning medan **mapfile.map** motsvarar en given map-fil som användaren vill köra. Mapfilen läses då in med hjälp av funktionen `fopen()`. Därefter kontrolleras mapfilens format genom att kontrollera ifall det endast existerar en sträng med line_has_one_string(). Därefter kontrolleras med hjälp av line_is_blank() om det saknas text. Med white_space kollar vi om det finns mellanrum i texten och sista testet görs med line_is_comment() för att se så att raden inte endast är en kommentar. Om filen passerar alla test så anropas funktionen `build_graph()` där en graf skapas i programmet med hjälp av `graph_insert_node()` samt `graph_insert_edge()`. Efter att grafen har skapats så läser programmet in två noder från användaren. Dessa noder testas först för att se om de existerar i grafen, ifall dom inte gör det skrivs ett felmeddelande ut och låter användaren försöka med nya noder. Ifall de existerar så anropas funktionen

`find_path(g,srcnode,destnode)`. Denna funktionen utför en sökning med **bredden först** där den börjar vid startnoden, tar ut grannoderna till denna nod med graph_neighbours och sätter dessa i en Kö. Sedan gör den samma behandling av alla grannar och fortsätter tills dess att Kön är tom.

6 Testkörningar

6.1 Implementation av graf som lista

I figur(1) nedan ser vi att användaren har angivit noder som ej existerar i grafen.

```
PS C:\Users\johan\OneDrive\Skrivbord\DoA\OU\OU5> gcc -std=c99 -Wall -o is_connected is_connected.c dlist.h dlist.c list.h list.c graph.h graph.c queue.h queue.c array_1d.h array_1d.c
PS C:\Users\johan\OneDrive\Skrivbord\DoA\OU\OU5> ./is_connected airmap1.map
Enter origin and destination (quit to exit): CAT BMA
Node CAT does not exist, try again!
Enter origin and destination (quit to exit): UME CAT
Node CAT does not exist, try again!
Enter origin and destination (quit to exit): PJA UME
There is no path from PJA to UME.

Enter origin and destination (quit to exit): PJA CAT
Node CAT does not exist, try again!
Enter origin and destination (quit to exit): quit
Normal exit.
PS C:\Users\johan\OneDrive\Skrivbord\DoA\OU\OU5>
```

Figur 1: Figur som visar körning med angiven nod som ej existerar i grafen, graf implementerad som dlist.

Då användaren har angett en nod som ej existerar skriver programmet ut att den angivna noden ej existerar i grafen och låter då användaren försöka igen med ett annat nod-namn.

I figur(2) visas en körning då angivna noder existerar i grafen.

```
PS C:\Users\johan\OneDrive\Skrivbord\DoA\OU\OU5> gcc -std=c99 -Wall -o is_connected is_connected.c dlist.h dlist.c list.h list.c graph.h graph.c queue.h queue.c array_1d.h array_1d.c
PS C:\Users\johan\OneDrive\Skrivbord\DoA\OU\OU5> ./is_connected airmap1.map
Enter origin and destination (quit to exit): BMA MMX
There is a path from BMA to MMX.

Enter origin and destination (quit to exit): BMA UME
There is a path from BMA to UME.

Enter origin and destination (quit to exit): UME PJA
There is no path from UME to PJA.

Enter origin and destination (quit to exit): PJA LLA
There is a path from PJA to LLA.

Enter origin and destination (quit to exit): quit
Normal exit.
PS C:\Users\johan\OneDrive\Skrivbord\DoA\OU\OU5>
```

Figur 2: Figur som visar körning med angiven nod som ej existerar i grafen, graf implementerad som dlist.

Som vi ser, om de två angivna noderna har en båge mellan sig så skriver programmet ut att det finns en väg mellan dem. För det fallet då det ej finns en båge skriver programmet ut att det inte finns en väg mellan de angivna noderna.

6.2 Implementation av graf som 1D-array

Precis som i föregående sektion visar figur(3) en körning då användaren har angett obefintliga noder.

Figur 3: Figur som visar körning med angiven nod som ej existerar i grafen, graf implementerad som 1D-array.

Precis som tidigare skriver då programmet ut att noden som har angetts inte existerar i grafen och ber användaren försöka igen.

I figur(4) anger användaren noder som existerar i grafen.

Figur 4: Figur som visar en korrekt körning med angivna noder som existerar i grafen, graf implementerad som 1D-array.

Liknande som innan, ifall de två noderna som anges har en båge mellan sig så skriver programmet ut att det finns en väg mellan destinationerna. Medan för det fallet då det ej existerar en väg skriver den ut att det inte existerar en väg mellan dem.

7 Arbetsfördelning

Under arbetets gång satt vi tre i ett samtal och streamade våra skärmar. Om t.ex Otto började på första funktionen i gränsytan och Måns den andra så började Johannes på tredje. Då en person kände sig nöjd med sin implementation tittade vi alla igenom den för att alla skulle förstå den, samtidigt felsöktes den för att gardera oss från möjliga fel. I det fall då en person inte riktigt lyckades implementera en funktion så gjordes den tillsammans med övriga medlemmar

av gruppen för att försäkra oss att alla har förståelsen av hur det fungerar. Sedan då samtliga var överens om att en funktion fungerar hoppade alla vidare och började på varsin ny funktion. Därefter upprepades stegen som nämndes ovan. Detta är grunden till hur vi fördelade arbetet i denna uppgift och vi alla känner oss nöjda med vårt individuella bidrag.

8 Reflektioner

Vi i gruppen är alla eniga om att uppgiften har varit klurig och tidskrävande men ändå rolig och samtidigt gett väldigt mycket.

Först och främst, genom att vi själva fick välja uppbyggnaden av implementationen ledde detta till att vi var tvungen att väga de olika gränsytornas för- och nackdelar. Vi valde att göra implementationerna med lista och endimensionellt fält väldigt lika varandra och vi kände att implementationen med lista var mycket enklare, smidig och lättkodad än för fältimplementationen. Anledningen till detta är att listan är riktad och därför erbjuder bättre möjligheter för sökning. Fördelen med fältimplementationen var att vi gjorde grannlistan till ett fält av ettor och nollor vilket troligtvis effektiviserar sökningen eftersom att vi då söker efter sant/falskt-villkor istället för hela textsträngar, men detta hade också kunnat göras med lista.

I helhet skiljer sig denna uppgift från tidigare obligatoriska uppgifter, då denna uppgift krävde planering samt implementation av fler än en gränsyta. Detta tyckte vi alla var väldigt bra, under planeringen tittade vi igenom de olika gränsytornas beståndsdelar för att avgöra vilken av dem skulle passa oss samt uppgiften så bra som möjligt. Detta i sig gav en fördjupad förståelse för respektive gränsytor, samt gav en helhetsbild av hur lösningen till problemet skulle gå till.

Appendix

A1

UME	Umeå
BMA	Bromma
MMX	Malmö
GOT	Göteborg
LLA	Luleå
PJA	Pajala

A2

Gränsyta dlist

dlist_empty(free_func) - Konstruerar en tom lista.

dlist_is_empty(l) - Returnerar **true** om listan l är tom.

dlist(l) - Kontrollerar och returnerar den första positionen i listan l.

dlist_next(l,p) - Tar in positionen p för att kontrollera samt returnerar det nästkommande värde i listan l.

dlist_is_end(l,p) - Kontrollerar om positionen p är sist i listan l för att returnerar **true** ifall de är.

dlist_inspect(l,p) - Tar in positionen p för att kontrollera samt returnerar värde från listan l.

dlist_insert(l,v,p) - Sätter in ett värde v på positionen p i listan l.

dlist_remove(l,p) - Tarbort elementet på position p i listan l.

dlist_kill(l) - Förstör listan l och returnerar allt dynamiskt minne från listan och dess element.

dlist_print(l,print_func) - Itererar över elementen i listan l och skriver ut dess värden.

A3

Gränsyta graph

nodes_are_equal(n1,n2)	- Returnerar **true** ifall två noder $n1$ och $n2$ är lika annars **false**.
graph_empty(max_nodes)	- Konstruerar en tom graf för max antal noder *max_empty*.
graph_is_empty(g)	- Kontrollera ifall grafen g är tom och returnerar **true** ifall den är, annars **false**.
graph_has_edges(g)	- Kontrollera ifall grafen g innehåller bågar, returnerar **true** om den har och **false** i annat fall.
graph_insert_node(g,s)	- Sätter in nodnamnet s i grafen g.
graph_find_node(g,s)	- Letar upp och returner noden s med identifikation s om den existerar i grafen g, annars returneras `NULL`.
graph_node_is_seen(g,n)	- Returnerar **true** om noden n i grafen g har setts tidigare.
graph_node_set_seen(g,n,seen)	- Sätter statuset *seen* för noden n i grafen g om noden har setts.
graph_reset_seen(g)	- Återställer *seen* status för alla noder i grafen g.
graph_insert_edge(g,n1,n2)	- Sätter in en båge i grafen g mellan ursprungsnoden $n1$ och destinations noden $n2$.
graph_delete_node(g,n)	- Tar bort noden n från grafen g.
graph_delete_edge(g,n1,n2)	- Tar bort bågen mellan nod $n1$ och $n2$ från grafen g.
graph_choose_node(g)	- Tar och returnerar en godtycklig nod från grafen g.
graph_neighbours(g,n)	- Returnerar en lista med grannar för noden n från grafen g.
graph_kill(g)	- Förstör grafen g och returnerar allt dynamiskt minne från grafen.
graph_print(g)	- Itererar över elementen i grafen g och skriver ut dess värden.

A4

Gränsyta queue

queue_empty(free_func)	- Konstruerar en tom kö.
queue_is_empty(q)	- Returnerar **true** om kön q är tom.
queue_enqueue(q,v)	- Sätter in ett värde v sist i kön q.
queue_dequeue(q)	- Tarbort det första värdet in kön q.
queue_front(q)	- Kontrollerar och returnerar det första värdet i kön q.
queue_kill(q)	- Förstör kön q och returnerar allt dynamiskt minne från kön och dess element.
queue_print(q,print_func)	- Itererar över elementen i kön q och skriver ut deras värden.

A5

Gränsyta array_1d

array_1d_create(lo,hi,free_func)	- Konstruerar en tom matris mellan låga och höga index *lo* respektive *hi*.
array_1d_low(a)	- Kontrollerar och returnerar låga index *lo* från matris *a*.
array_1d_high(a)	- Kontrollerar och returnerar höga index *hi* från matris *a*.
array_1d_inspect_value(a,i)	- Inspekterar och returnerar värdet på position *i* från matris *a*.
array_1d_has_value(a,i)	- Kontrollerar ifall ett värde finns på positionen *i* i matrisen *a* och returnerar **true** ifall de gör.
array_1d_set_value(a,v,i)	- Ansätter värdet *v* på position *i* i matrisen *a*.
array_1d_kill(a)	- Förstör matris *a* och returnerar allokerat minne.
array_1d_print(l,print_func)	- Itererar över elementen i matrisen *a* och skriver ut dess värden.

Referenser

[1] Lars-Erik Janlert, Torbjörn Wiberg
Datatyper och algoritmer
Upplaga 2, p. 337
2000