Marius Joldoş Iosif Ignat

Data Structures and Algorithms

Laboratory Guide

Editura U.T.PRES
Cluj-Napoca, 2002

Preface

A collection of values that share a common set of operations is called a *data type*. *Structured* or *composite* data types are collections of individual data items of the same or different data types. *Data structures* are collections of variables, possibly of different data types, connected in various ways. High-level languages prior to Pascal usually limited their concepts of data types to those provided directly by hardware (integers, reals, double precision integers and reals, and blocks of contiguous locations). Two objects had different types if it was necessary to generate different code to manipulate them. Pascal and later languages have taken a rather different approach, based on the concept of *abstract data types*. An *abstract data type* is a programming language facility for organizing programs into modules using criteria that are based on the data structures of the program. Also, an *abstract data type* can be defines as a set of values and a set of procedures that manipulate those values. The specification of the module should provide all information required for using the type, including the allowable values of the data and the effects of the operations. However, details about the implementation, such as data representations and algorithms for implementing the operations, are hidden within the module. This separation of specification from implementation is a key idea of abstract data types.

This laboratory guide is intended to facilitate understanding of the widely used data structures such as lists, trees, graphs, hash tables and the operations associated with them. Also, the main general algorithms development methods − greedy, backtracking, branch-and-bound, divide and conquer, dynamic programming and some heuristics − form the object of this guide, as well as some sorting algorithms.

The language used for implementations is C. In implementing the assignments a good programming style is very important, therefore some guide is given in Appendix A.

This guide is intended to be used by the students of the first year of the Automation and Computer Science Faculty of the Technical University of Cluj-Napoca, who study in English.

Cluj-Napoca, February 2003

Iosif Ignat and Marius Joldoş

Contents

1. Singly Linked Lists

1.1. Purpose

The purpose of this lab session is to acquire skills in working with singly linked lists.

1.2. Brief Theory Reminder

- A *list* is a finite ordered set of elements of a certain type.
- The elements of the list are called *cells* or *nodes*.
- A list can be represented *statically*, using arrays or, more often, *dynamically*, by allocating and releasing memory as needed. In the case of static lists, the ordering is given implicitly by the one-dimension array. In the case of dynamic lists, the order of nodes is set by *pointers*. In this case, the cells are allocated dynamically in the heap of the program. Dynamic lists are typically called *linked lists*, and they can be singly- or doubly-linked.

We will present some of the operations on singly-linked lists below.
The structure of a node may be:

```
typedef struct nodetype
{
  int key; /* an optional field */
  ... /* other useful data fields */
  struct nodetype *next; /* link to next node */
} NodeT;
```

A Singly-linked List Model

A singly-linked list may be depicted as in Figure 1.1.

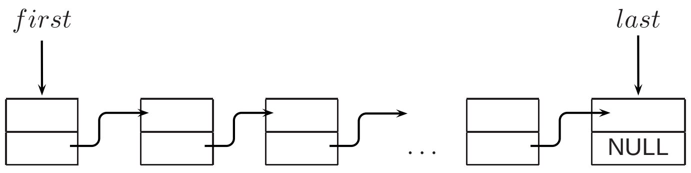


Figure 1.1.: A singly-linked list model.

1.3. Operations on a Singly-linked List

Creating a Singly-linked List

Consider the following steps which apply to *dynamic* lists:

1. Initially, the list is empty. This can be coded by setting the pointers to the first and last cells to the special value $NULL$, i.e. $first = NULL, last = NULL$.
2. Reserve space for a new node to be inserted in the list:

   ```
   /* reserve space */
   p = ( NodeT * )malloc( sizeof( NodeT ));
   ```

   Then place the data into the node addressed by p. The implementation depends on the type of data the node holds. For primitive types, you can use an assignment such as $*p = data$.
3. Make the necessary connections:

   ```
   p->next = NULL;  /* node is appended to the list */
   if ( last != NULL ) /* list is not empty */
     last->next = p;
   else
     first = p; /* first node */
   last = p;
   ```

Accessing a Node of a Singly-linked List

The nodes of a list may be accessed *sequentially*, gathering useful information as needed. Typically, a part of the information held at a node is used as a *key* for finding the desired information. The list is scanned linearly, and the node may or may not be present on the list. A function for searching a specific key may contain the following sequence of statements:

```
NodeT *p;
p = first;
while ( p != NULL )
   if ( p->key = givenKey )
   {
     return p; /* key found at address p */
   }
   else
     p = p->next;
   return NULL; /* not found */
```

Inserting a Node in a Singly-linked List

The node to be inserted may be created as shown at §1.3. We will assume here that the node to insert is pointed to by p.

- If the list was empty then there would be only one node, i.e.

  ```
  if ( first == NULL )
  {
     first = p;
     last = p;
     p->next = NULL;
  }
  ```

- If the list was not empty, then insertion follows different patterns depending on the position where the node is to be inserted. Thus, we have the following cases:

  1. Insertion before the first node of the list:

     ```
     if ( first != NULL )
     {
       p->next = first;
       first = p;
     }
     ```

  2. Insertion after the last node of the list (this operation is also called *append*):

     ```
     if ( last != NULL )
     {
       p->next = NULL;
       last->next = p;
       last = p;
     }
     ```

  3. Insertion before a node given by its *key*. There are two steps to execute:

     a) Search for the node containing the *givenKey*:

     ```
     NodeT *q, *q1;
     q1 = NULL; /* initialize */
     q = first;
     while ( q != NULL )
     {
        if ( q->key == givenKey ) break;
        q1 = q;
        q = q->next;
     }
     ```

     b) Insert the node pointed to by p, and adjust links:

     ```
     if ( q != NULL )
     {
      /* node with key givenKey has address q */
        if ( q == first )
        { /* insert after first node */
          p->next = first;
     ```

```
        first = p;
      }
      else
      {
        q1->next = p;
        p->next = q;
      }
    }
```

4. Insertion after a node given by its *key*. Again, there are two steps to execute:

   a) Search for the node containing the *givenKey*:

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
q = first;
while ( q != NULL )
{
  if ( q->key == givenKey ) break;
  q1 = q;
  q = q->next;
}
```

   b) Insert the node pointed to by p, and adjust links:

```
if ( q != NULL )
{
  p->next = q->next; /* node with key givenKey has address q */
  q->next = p;
  if ( q == last ) last = p;
}
```

Removing a Node of a Singly-linked List

When we are to remove a node from a list, there are some aspects to take into account: (i) list may be empty; (ii) list may contain a single node; (iii) list has more than one node. And, also, deletion of the first, the last or a node given by its key may be required. Thus we have the following cases:

1. Removing the first node of a list

```
NodeT *p;
if ( first != NULL )
{ /* non-empty list */
  p = first;
  first = first->next;
  free( p ); /* free up memory */
  if ( first == NULL ) /* list is now empty */
    last = NULL;
}
```

2. Removing the last node of a list

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
q = first;
if ( q != NULL )
{ /* non-empty list */
  while ( q != last )
  { /* advance towards end */
    q1 = q;
    q = q->next;
  }
  if ( q == first )
  { /* only one node */
    first = last = NULL;
  }
  else
  { /* more than one node */
    q1->next = NULL;
```

```
        last = q1;
      }
      free( q );
    }
```

3. Removing a node given by a *key*

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
q = first;
/* search node */
while ( q != NULL )
{
  if ( q->key == givenKey ) break;
  q1 = q;
  q = q->next;
}
if ( q != NULL )
{   /* found a node with supplied key */
  if ( q == first )
  { /* is the first node */
    first = first->next;
    free( q ); /* release memory */
    if ( first == NULL ) last = NULL;
  }
  else
  { /* other than first node */
    q1->next = q->next;
    if ( q == last ) last = q1;
    free( q ); /* release memory */
  }
}
```

Complete Deletion of a Singly-linked List

For a complete deletion of a list, we have to remove each node of that list, i.e.

```
NodeT *p;
while ( first != NULL )
{
  p = first;
  first = first->next;
  free( p );
}
last = NULL;
```

Stacks

A *stack* is a special case of a singly-linked list which works according to LIFO[1] algorithm. Access is restricted to one end, called the *top* of the stack. A *stack* may be depicted as in Figure 1.2. Its operations are:

push – push an element onto the top of the stack;
pop – pop an element from the top of the stack;
top – retrieve the element at the top of the stack;
delete – delete the whole stack. This can be done as explained in the previuos paragraph.

Queues

A *queue* is also a special case of a singly-linked list, which works according to FIFO[2] algorithm. Of the two ends of the queue, one is designated as the *front* – where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue*). A *queue* may be depicted as in Figure 1.3. The main operations are:

[1]L[ast] I[n] F[irst] O[ut]
[2]F[irst] I[n] F[irst] O[ut]

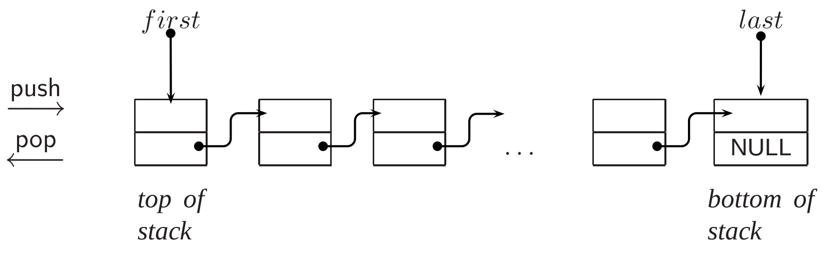


Figure 1.2.: A stack model.

enqueue – place an element at the tail of the queue;
dequeue – take out an element form the front of the queue;
delete – delete the whole queue. This can be done as explained at §1.3.

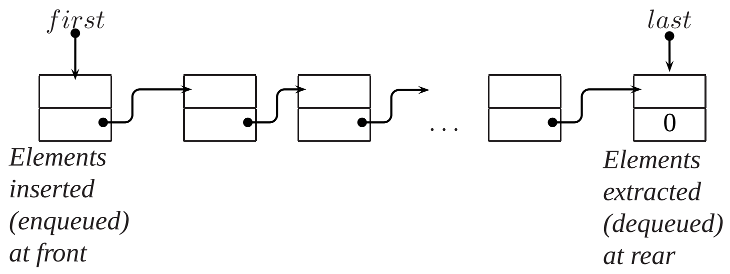

first
last
0
Elements inserted (enqueued) at front
Elements extracted (dequeued) at rear

Figure 1.3.: A queue model.

1.4. Lab.01 Assignments

For all the following tasks input and output is from files, given as command line arguments.

1.1. There is a garage where the access road can accommodate any number of trucks at one time. The garage is build such a way that only the last truck entered can be moved out. Each of the trucks is identified by a positive integer (a truck_id). Write a program to handle truck moves, allowing for the following commands:

   a) `On_road` (truck_id);
   b) `Enter_garage` (truck_ id);
   c) `Exit_garage` (truck_id);
   d) `Show_trucks` (`garage` or `road`);

   If an attempt is made to get out a truck which is not the closest to the garage entry, the error message `Truck` x `not near garage door`.

1.2. Same problem, but with a circular road and two entries: one for entry, another for exit. Trucks can get out only in the order they got in. .
   I/O description. Input:

```
On_road(2)
On_road(5)
On_road(10)
On_road(9)
On_road(22)
Show_trucks(road)
Show_trucks(garage)
Enter_garage(2)
Show_trucks(road)
Show_trucks(garage)
Enter_garage(10)
Enter_garage(25)
Exit_garage(10)
Exit_garage(2)
show_trucks(road)
show_trucks(garage)
```

   Output:

```
road:␣2␣5␣10␣9␣22
garage:␣none
road:␣5␣10␣9␣22
garage:␣2
error:␣25␣not␣on␣road!
error:␣10␣not␣at␣exit!
road:␣2␣5␣9␣22
garage:␣10
```

1.3. Define and implement functions to operate on a list cell structure defined as follows:

```
typedef struct node
{
  struct node *next;
  void *data;
} NodeT;
```

using the model given in Figure 1.4. Data cells should contain a numeric key and a string, e.g. a student's name and id number. .
I/O description. Operations should be coded as: `cre`=create empty list, `del` key=delete node with key from list, `dst`=delete the first node (not the sentinel), `dla`=delete the last node (not the sentinel), `ins` $data$=insert data element in ascending order of keys, `ist` $data$=insert a node with $data$ as first, `ila` $data$=insert a node with $data$ as last, `prt`=print list.

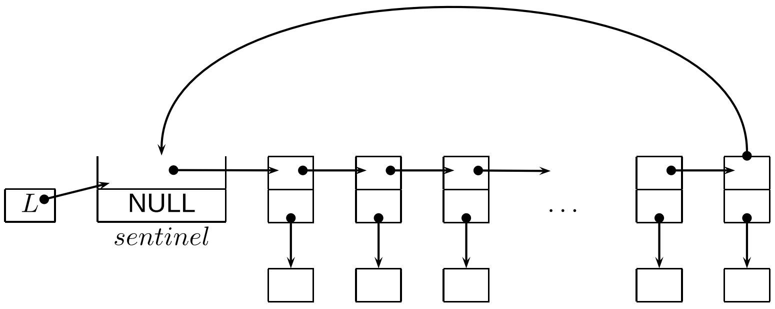


Figure 1.4.: A model of a list for problem 1.3.

1.4. Implement a list as a static array containing pointers to information nodes allocated on the heap, using the model of Figure 1.5 .
I/O description. Operations should be coded as: `cre`=create empty list, `del` key=delete node with key from list, `ist` $data$=insert node with $data$ as first node, `ila` $data$=insert node with $data$ as last node, `dst`=delete first node, `dla`=delete last node, `prt`=print list. The data nodes could be records containing for instance a product id, product name and amount.

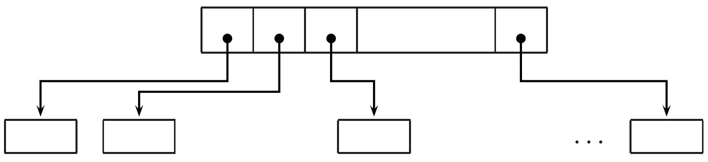


Figure 1.5.: A model of a list for problem 1.4.

1.5. Topological sort. The elements of a set M are denoted by strings. Read pairs of elements $(x, y),\ x, y \in M$ meaning "element x precedes element y. List the elements of the set such that if an element precedes another, then it should be listed *before* its successors. Your algorithm should only make use of lists (i.e. no separate sorting). In case that conflicting pairs are given, the message `Pair` x, y `ignored due to conflict` should be issued..
I/O description. Input:

```
alpha,delta
alpha,epsilon
epsilon,delta
```

1.6. Create a singly linked list, ordered alphabetically, containing words and their occurrence frequency. The input is paragraphs read from a file given as an argument to the program. Output the list alphabetically to another file, given

as an argument to the rpogram. .
I/O description. Examples of input and output are given in Figure 1.6 Each item in the output is of the form $word : number\_of\_occurences$.

Input:

```
The␣quick␣brown␣fox␣jumps␣over␣the␣lazy␣dog.
```

Output (case sensitive):

```
The:1␣brown:1␣dog:1␣fox:1␣jumps:1␣lazy:1␣over:1␣quick:1␣the:1
```

Output (case insensitive):

```
brown:1␣dog:1␣fox:1␣jumps:1␣lazy:1␣over:1␣quick:1␣the:2
```

Figure 1.6.: Example input and output for problem 1.4.6.

1.7. Write a program which creates two ordered list, based on an integer key, and then merges them together. .
I/O description. Input:

```
i1␣23␣47␣52␣30␣2␣5␣-2
i2␣-5␣-11␣33␣7␣90
p1
p2
m
p1
p2
```

Output:

```
1:␣-2␣2␣5␣23␣30␣47␣52
2:␣-11␣-5␣7␣33␣90
1:␣-11␣5␣2␣2␣5␣7␣23␣...
2:␣empty
```

Thus, "commands" accepted are: `i`n=insert onto list $n \in \{1, 2\}$, `p`n=print contents of list n, `m`=merge lists.

1.8. Imagine an effective dynamic structure for storing sparse matrices. Write operations for addition, and multiplication of such matrices. .
I/O description. Input:

```
m1␣40␣40
(3,␣3,␣30)
(25,␣15,␣2)
m2␣40␣20
(5,␣12␣1)
(7␣14␣22)
m1+m2
m1*m2
```

, where `m1`=read elements for matrix `m1`, and the following triples are $(row, col, value)$ for that matrix. Reading ends when either another command or the end of file marker is encountered. **E.g.** `m1+m2`=add matrix 1 to matrix 2, and `m1*m2`=multiply matrix `m1` with matrix `m2`. Output should be given in similar format, i.e triplets.

1.9. Imagine an effective dynamic structure for storing polynomials. Write operations for addition, subtraction, and multiplication of polynomials. .
I/O description. Input:

```
p1=3x^7+5x^6+22.5x^5+0.35x-2
p2=0.25x^3+.33x^2-.01
p1+p2
p1-p2
p1*p2
```

, Output:

```
<list␣of␣sum␣polynomial>
<list␣of␣difference␣polynomial>
<list␣of␣product␣polynomial>
```

2. Circular Singly Linked Lists

2.1. Purpose

In this lab session we will enhance singly-linked lists with another feature: we'll make them circular. And, as was the case in the previous lab session, we will show how to implement creation, insertion, and removal of nodes.

2.2. Brief Theory Reminder

A *circular singly linked list* is a singly linked list which has the last element linked to the first element in the list. Being circular it really has no ends; then we'll use only one pointer $pNode$ to indicate one element in the list – the newest element. Figure 2.1 show a model of such a list.

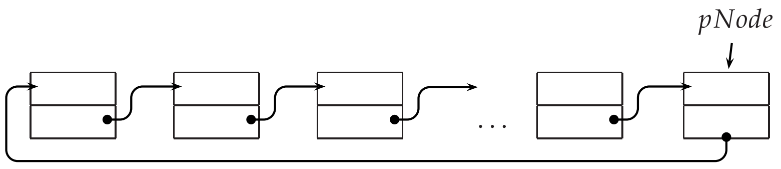


Figure 2.1.: A model of a circular singly-linked list.

The structure of a node may be:

```
typedef struct nodetype
{
  int key; /* an optional field */
  /* other useful data fields */
  struct nodetype *next;
  /* link to next node */
} NodeT;
```

2.3. Operations on a Circular Singly-linked List

Creating a Circular Singly-linked List

1. Initially, the list is empty, i.e. $pNode = NULL$.
2. Generate a node to insert in the list:

```
/* reserve space */
p = ( NodeT * )malloc( sizeof( NodeT ));
Then read the data into the node
addressed by p
```

3. Link it in the list:

```
p->next = NULL;
/* node is appended to the list */
if ( pNode == NULL )
{ /* empty list */
  pNode = p;
  pNode->next = p;
}
else
{ /* nonempty list */
  p->next = pNode->next;
  pNode->next = p;
  pNode = p;
  /* pNode points to the newest
     node in the list */
}
```

Accessing a Node of a Circular Singly-linked List

The nodes of a list may be accessed sequentially, starting at node $pNode$, as follows:

```
NodeT *p;

p = pNode;
if ( p != NULL )
  do
  {
    access current node and get data;
    p = p->next;
  }
  while ( p != pNode );
```

Another choice is to look for a key, say $givenKey$. Code for such list scan is given below:

```
NodeT *p;

p = pNode;
if ( p != NULL )
  do
  {
    if ( p->key = givenKey )
    { /* key found at address p */
      return p;
    }
    p = p->next;
  }
  while ( p != NULL );
return NULL; /* not found */
```

Inserting a Node of a Circular Singly-linked List

A node may be inserted *before* a node containing a given key, or *after* it. Both cases imply searching for that key, and, if that key exists, creating the node to insert, and adjusting links accordingly.

Inserting Before a node with key *givenKey*

There are two steps to execute:

1. Find the node with key $givenKey$:
```
NodeT *p, *q, *q1;

q1 = NULL; /* initialize */
q = pNode;
do
{
  q1 = q;
  q = q->next;
  if ( q->key == givenKey ) break;
}
while ( q != pNode );
```
2. Insert the node pointed to by p, and adjust links:
```
if ( q->key == givenKey )
{ /* node with key givenKey has address q */
  q1->next = p;
  p->next = q;
}
```

Inserting after a node with key *givenKey*

Again, there are two steps to execute:

1. Find the node with key $givenKey$:

```
NodeT *p, *q;

q = pNode;
do
{
  if ( q->key == givenKey ) break;
  q = q->next;
}
while ( q != pNode );
```

2. Insert the node pointed to by p, and adjust links:

```
if ( q->key == givenKey )
{ /* node with key givenKey has address q */
  p->next = q->next;
  q->next = p;
}
```

Removing a Node from a Circular Singly-linked List

Again there are two steps to take:

1. Find the node with key $givenKey$:

```
NodeT *p, *q, *q1;
q = pNode;
do
{
  q1 = q;
  q = q->next;
  if ( q->key == givenKey ) break;
}
while ( q != pNode );
```

2. Delete the node pointed to by q. If that node is $pNode$ then we adjust $pNode$ to point to its previous.

```
if ( q->key == givenKey )
{ /* node with key givenKey has address q */
  if ( q == q->next )
  {
    /* list now empty */
  }
  else
  {
    q1->next = q->next;
    if ( q == pNode ) pNode = q1;
  }
  free( q );
}
```

Complete Deletion of a Circular Singly-linked List

For complete deletion of a list, we have to remove each node of that list, i.e.

```
NodeT *p, *p1;
p = pNode;
do
{
  p1 = p;
  p = p->next;
  free( p1 );
}
while ( p != pNode );
pNode = NULL;
```

2.4. Lab.02 Assignments

2.1. Define and implement functions for operating on the data structure given below and the model of figure 2.2.

```
struct circularList
{
  int length;
  NodeT *first;
}
```

Operations should be coded as: `ins` *data*=insert *data* in the list if it is not already there (check the key part of data area for that purpose), `del` *key*=delete data containing *key* from list (if it is on the list), `ist` *data*=insert node with *data* as first node, `dst`=delete first node, `prt`=print list, `fnd` *key*=find data node containing *key* in its data area.

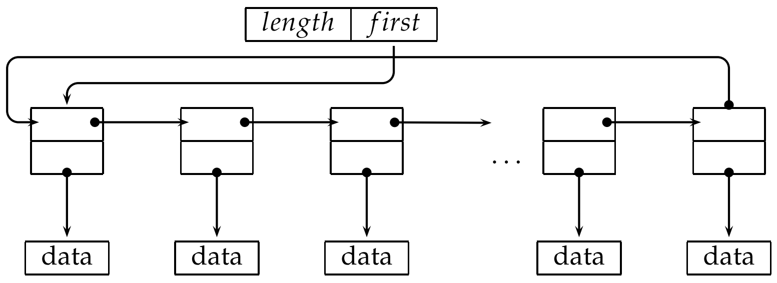


Figure 2.2.: Another model of a circular singly-linked list.

2.2. Define and implement functions for operating on the data structure given below and the model of figure 2.3.

```
struct circularList
{
  int length;
  NodeT *current;
}
```

The field `current` changes to indicate: the last inserted node if the last operation was insert, the found node for find, the next node for delete, Operations should be coded as: `ins` *data*=insert a node containing *data* in the list if an element with the same key is not already there, `fnd` *key*=find data node containing *key* in its data area (check the key part of data area for that purpose), `del` *key*=delete node with *key* in ints data area from list (if it is on the list), `icr` *data*=insert after current position, `dcr`=delete current node, `prt`=print list.

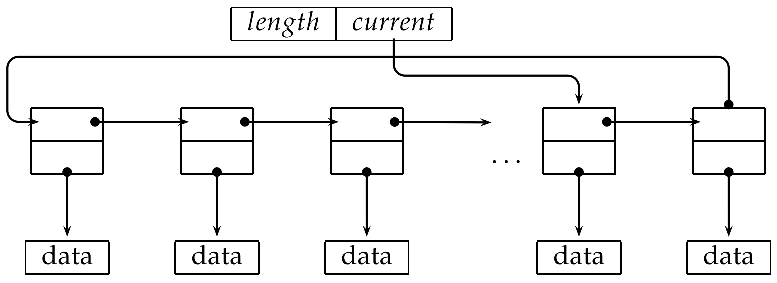


Figure 2.3.: Yet another model of a circular singly-linked list.

2.3. Implement a circular buffer[1] to hold records containing student-related data. A producer-consumer principle is to be implemented according to the following:

a) Records are accepted as they are produced.
b) If there are no records in the buffer, consumer is postponed till producer places one.
c) If the buffer fills up, producer is postponed till consumer takes one out.

Use a limit of 1o for queue size, so it fills up fast. .
I/O description. Input:

```
p227,Ionescu␣I.␣Ion,␣3071,␣DSA,␣10
p231,Vasilescu␣T.␣Traian,␣3087,␣DSA,␣5
lq
p555,John␣E.␣Doe,␣3031,␣DSA,␣9
p213,King␣K.␣Kong,␣3011,␣DSA,␣2
p522,Mickey␣M.␣Mouse,␣3122,␣ART,␣10
```

[1]buffer=memory area of fixed size used as temporary storage for I/O operations

```
...␣/*␣'...'␣means␣more␣records␣*/
lq
p573,Curtis␣W.␣Anthony,␣3012,␣ART,␣10
p257,Bugs␣R.␣Bunny,␣3000,␣GYM,␣10
c
c
c
c
lq
```

Output:

```
227,Ionescu␣I.␣Ion,␣3071,␣DSA,␣10
231,Vasilescu␣T.␣Traian,␣3087,␣DSA,␣5
eoq
227,Ionescu␣I.␣Ion,␣3071,␣DSA,␣10
231,Vasilescu␣T.␣Traian,␣3087,␣DSA,␣5
555,John␣E.␣Doe,␣3031,␣DSA,␣9
213,King␣K.␣Kong,␣3011,␣DSA,␣2
522,Mickey␣M.␣Mouse,␣3122,␣ART,␣10
...
eoq
queue␣full.␣postponed␣573
queue␣full,␣postponed␣257
522,Mickey␣M.␣Mouse,␣3122,␣ART,␣10
...
573,Curtis␣W.␣Anthony,␣3012,␣ART,␣10
257,Bugs␣R.␣Bunny,␣3000,␣GYM,␣10
eoq
```

where `227`=student id, `Ionescu I. Ion`=name, `3071`=group id, `DSA`= 3-letter course code, `10`=mark; `eoq`=end-of-queue, `p`=record from producer, `c`=record is consumed, `lq`=list queue contents [command].

2.4. Implement two dynamically allocated circular lists to hold solve the following problem:

- A number of person names (given as last name, first name) are inserted in the first list.
- A count is read
- Count in a circle, left to right, and move each n^{th} person to the second list, till a single person remains in the first list. Counting continues with the person following the moved one.
- Output the contents of both lists.

.

I/O description. Input:

```
Ionescu␣Ion
Vasilescu␣Traian
Doe␣John
Kong␣King
Mickey␣Mouse
Curtis␣Anthony
Bugs␣Bunny
7
```

Output:

```
First␣list:
-----------
Mickey␣Mouse
Second␣list:
------------
Bugs␣Bunny
Ionescu␣Ion
Doe␣John
Curtis␣Anthony
Vasilescu␣Traian
Kong␣King
```

2.5. Use a stack to reverse the order of the elements in a circular list sorted in ascending order. The data (which is also the key) in the list are person names like in the previous problem. .
I/O description. Input:

```
Ionescu␣Ion
Vasilescu␣Traian
Doe␣John
Kong␣King
Mickey␣Mouse
Curtis␣Anthony
Bugs␣Bunny
```

Output:

```
Vasilescu␣Traian
Mickey␣Mouse
Kong␣King
Ionescu␣Ion
Doe␣John
Curtis␣Anthony
Bugs␣Bunny
```

2.6. Use a stack to reverse the order of the elements in a circular list sorted in descending order. The data (which is also the key) in the list are person names like in the previous problem. .
I/O description. Input:

```
Ionescu␣Ion
Vasilescu␣Traian
Doe␣John
Kong␣King
Mickey␣Mouse
Curtis␣Anthony
Bugs␣Bunny
```

Output:

```
Bugs␣Bunny
Curtis␣Anthony
Doe␣John
Ionescu␣Ion
Kong␣King
Mickey␣Mouse
Vasilescu␣Traian
```

3. Doubly Linked Lists

3.1. Purpose

This lab session is intended to help you develop the operations on doubly-linked lists.

3.2. Brief Theory Reminder

A *doubly-linked* list is a (dynamically allocated) list where the nodes feature two relationships: *successor* and *predecessor*. A model of such a list is given in figure 3.1. The type of a node in a doubly-linked list may be defined as follows:

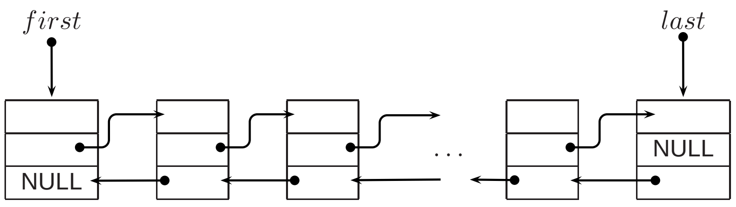


Figure 3.1.: A model of a doubly linked list.

```
typedef struct node_type
{
  KeyT key; /* optional */
  ValueT value;
  /* pointer to next node */
  struct node_type *next;
  /* pointer to previous node */
  struct node_type *prev;
} NodeT;
```

As we have seen when discussing singly-linked lists, the main operations for a doubly-linked list are:

- creating a cell;
- accessing a cell;
- inserting a new cell;
- deleting a cell;
- deleting the whole list.

3.3. Operations on a Doubly Linked List

In what follows we shall assume that the list is given by a pointer to its header cell, i.e.

```
/* header cell */
struct list_header
{
  NodeT *first;
  NodeT *last;
};
/* list is defined as a pointer
   to its header */
struct list_header *L;
```

Creating a Doubly Linked List

We shall take the list to be initially empty, i.e.

```
L->first = L->last = NULL;
```

After allocating space for a new node and filling in the data (the *value* field), insertion of this node, pointed by p, is done as follows:

```
if ( L->first == NULL )
{ /* empty list */
  L->first = L->last = p;
  p->next = p->prev = NULL;
}
else
{ /* nonempty list */
  L->last->next = p;
  p->prev = L->last;
  L->last = p;
}
```

Accessing a Node of a Doubly Linked List

Stating at a certain position (i.e. at a certain node) we can access a list:

- In sequential forward direction
  ```
  for ( p = L->first; p != NULL; p = p->next )
  {
    /* some operation o current cell */
  }
  ```
- In sequential backward direction
  ```
  for ( p = L->last; p != NULL; p->p->prev )
  {
    /* some operation o current cell */
  }
  ```
- Based on a key. Finding a node based on a given key may be achieved as we did at Lab. 1, §1.3.

Inserting a Node of a Doubly Linked List

We can insert a node before the first node in a list, after the last one, or at a position specified by a given key:

- before the first node:
  ```
  if ( L->first == NULL )
  { /* empty list */
    L->first = L->last = p;
    p_>next = p->prev = NULL;
  }
  else
  { /* nonempty list */
    p->next = L->first;
    p->prev = NULL;
    L->first->prev = p;
    L->first = p;
  }
  ```
- after the last node:
  ```
  if ( L->first == NULL )
  { /* empty list */
    L->first = L->last = p;
    p_>next = p->prev = NULL;
  }
  else
  { /* nonempty list */
    p->next = NULL;
    p->prev = L->last;
    L->last->next = p;
    L->last = p;
  }
  ```
- After a node given by its key:
  ```
  p->prev = q;
  p->next = q->next;
  if ( q->next != NULL ) q->next->prev = p;
  ```

```
q->next = p;
if ( L->last == q ) L->last = p;
```

Here we assumed that the node with the given key is present on list L and it we found it and placed a pointer to it in variable q.

Deleting a Node of a Doubly Linked List

When deleting a node we meet the following cases:

- Deleting the first node:

```
p = L->first;
L->first = L->first->next; /* nonempty list assumed */
free( p ); /* release memory taken by node */
if ( L->first == NULL )
  L->last == NULL; /* list became empty */
else
  L->first->prev = NULL;
```

- Deleting the last node:

```
p = L->last;
L->last = L->last->prev; /* nonempty list assumed */
if ( L->last == NULL )
  L->first = NULL; /* list became empty */
else
  L->last->next = NULL;
free( p ); /* release memory taken by node */
```

- Deleting a node given by its key. We will assume that the node of key $givenKey$ exists and it is pointed to by p (as a result of searching for it)

```
if ( L->first == p && L->last == p )
{ /* list has a single node */
  L->first = NULL;
  L->last = NULL; /* list became empty */
  free( p );
}
else
if ( p == L->first )
{ /* deletion of first node */
  L->first = L->first->next;
  L->first->prev = NULL;
  free( p );
}
else
{ /* deletion of an inner node */
  p->next->prev = p->prev;
  p->prev->next = p->next;
  free( p );
}
```

Deleting a Doubly Linked List

Deleting a list completely means deleting each of its nodes, one by one.

```
NodeT *p;

while ( L->first != NULL )
{
  p = L->first;
  L->first = L->first->next;
  free( p );
}
L->last = NULL;
```

3.4. Lab.03 Assignments

3.3.1. Define and implement the operations the data structure given below:

```
typedef struct
{
  int length;
  NodeT *first, *last;
} ListT;
typedef struct node
{
  void *data; // data is dynamically allocated
  struct node* *prev, *next;
} NodeT;
```

and the model of Figure 3.2.
Now assume that data is records with two fields: a name and a birthdate (see example input below).

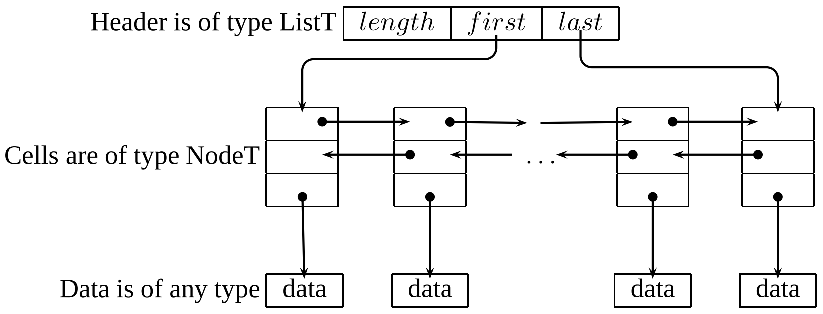


Figure 3.2.: A list model for problem 3.3.1.

.
I/O description. Operations should be coded as: `ins<record>`=insert `<record>` in the list, `del<birthdate>`=delete record(s) containing `<birthdate>` from list, `fnd<birthdate>`=find record containing <birthdate>, `ist<number>`=insert `<record>` as first, `dst`=delete first, `prt`=print list, `ita<number>`=insert `record` as last, `dta`=delete last node.
An example follows:

Input:

```
ins␣"Ionescu␣Ion"␣19870101
ins␣"Marin␣Maria"␣19880202
ist␣"Vasile␣Vlad"␣19850303
prt
fnd␣20000101
dst
prt
dta
prt
ita␣"Zaharia␣Zicu"␣19550505
prt
del␣19550505
prt
```

Output:

```
19850303␣Vasile␣Vlad
19870101␣Ionescu␣Ion
19880202␣Marin␣Maria
--
20000101␣not␣found
--
19870101␣Ionescu␣Ion
19880202␣Marin␣Maria
--
19870101␣Ionescu␣Ion
--
19870101␣Ionescu␣Ion
19550505␣Zaharia␣Zicu
--
19870101␣Ionescu␣Ion
--
```

Note that after each output a line with `-` is printed.

3.3.2. Implement the same operations as in the previous problem, using the same data for the list model given in Figure 3.3.

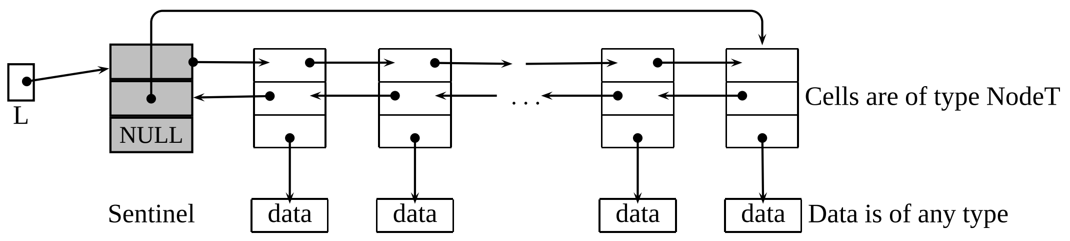


Figure 3.3.: A list model for problem 3.3.2.

3.3.3. Read a paragraph containing words from an input file. Then create a doubly-linked list containing the distinct words read, and their occurrence frequency. This list should have the structure as depicted in Figure 3.4.
The data structures are defined as follows:

```
typedef struct
{
  int length;
  NodeT *first , *last;
} ListT;
typedef struct
{
  char *word;
  double frequency;
  } DataT;
typedef struct node
{
  struct node *prev, *next;
  DataT data;
} NodeT;
```

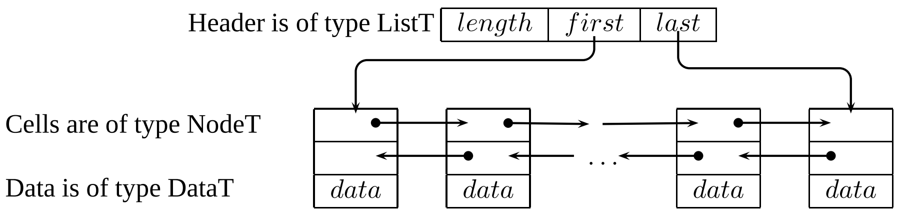


Figure 3.4.: A list model for problem 3.3.3.

The list should be alphabetically ordered. Print the words and their frequencies in alphabetical ascending and descending order, case insensitive.
Example data:
Input:

```
Everything LaTeX numbers for you has a counter associated with it.  The name of the counter
is the same as the name of the environment or command that produces the number, except
with no \.  Below is a list of some of the counters used in LaTeX's standard document styles
to control numbering.
```

Output:

```
\.:1                                you:1
a:2                                 with:2
as:1                                used:1
associated:1                        to:1
Below:1                             the:7
command:1                           that:1
control:1                           styles
counter:2                           standard:1
counters:1                          some:1
document:1                          same:1
environment:1                       produces:1
Everything:1                        or:1
except:1                            of:4
for:1                               numbers:1
has:1                               numbering.:1
in:1                                number,:1
is:2                                no:1
it.:1                               name:2
LaTeX:1                             list:1
LaTeX’s:1                           LaTeX:1
list:1                              LaTeX’s:1
name:2                              it.:1
no:1                                is:2
number,:1                           in:1
numbering.:1                        has:1
numbers:1                           for:1
of:4                                except:1
or:1                                Everything:1
produces:1                          environment:1
same:1                              document:1
some:1                              counters:1
standard:1                          counter:2
styles                              control:1
that:1                              command:1
the:7                               Below:1
to:1                                associated:1
used:1                              as:1
with:2                              a:2
you:1                               \.:1
```

3.3.4. Simulate a game, using two circular doubly linked list. The game involves a number of children, and you are supposed to read their names from a file. The children whose names are on lines numbered by prime numbers should be placed on the first list, and the others on the second list. Starting with the child whose name is on the line in the middle (or $\lfloor numberOfChildren/2 \rfloor$) of the second list, children on that list are counted clockwise. Every m^{th} child, where m is the number of elements in the first list is eliminated from that list. Counting goes on with the next child. Repeat this counting m times or till the second list gets empty. Your program should output the initial lists and the final second list.
Example. Input:

```
John
James
Ann
George
Amy
Fanny
Winston
Bill
Suzanna
Hannah
Alex
Gaby
Thomas
```

Output:

```
Prime␣[7␣children]:
------
John
James
Ann
Amy
Winston
Alex
Thomas

Non-prime␣[6␣children]:
----------
George
Fanny
Bill
Suzanna
Hannah
Thomas

Starter:␣Bill
Count:␣7

Empty␣list
```

3.3.5. The same problem as the one before, but counting counter-clockwise.

3.3.6. Read a paragraph containing words from an input file. Then create a doubly-linked list containing the distinct words read, where the words of the same length are placed in the same list, in ascending order.
The data structures are defined as follows:

```
typedef struct
{
  int length;
  NodeT *first , *last;
} ListT;
typedef struct data
{
  char *word;
  struct data *next ;
} DataT;
typedef struct node
{
  int wordLength;
  struct node *prev, *next ;
  DataT *words;
} NodeT;
```

Print the lists in descending order of their word length, case insensitive.
Example data:
Input:

```
Everything LaTeX numbers for you has a counter associated with it.  The name of the counter
is the same as the name of the environment or command that produces the number, except
with no \.  Below is a list of some of the counters used in LaTeX’s standard document styles
to control numbering.
```

Output:

```
1:␣a
2:␣\.␣as␣in␣is␣no␣of␣or␣to
3:␣for␣has␣it.␣the␣you
4:␣list␣name␣same␣some␣that␣used␣with
5:␣Below␣LaTeX
6:␣except␣styles
7:␣command␣control␣counter␣LaTeX’s␣number,␣numbers
8:␣counters␣document␣produces␣standard
10:␣associated␣Everything␣numbering.
```

```
11:␣environment
```

4. Trees

4.1. Purpose

In this lab session we will work on basic operations for binary trees, completely balanced binary trees and arbitrary trees.

4.2. Brief Theory Reminder

The *binary* tree – a tree in which each node has at most two descendants – is very often encountered in applications. The two descendants are usually called the $left$ and $right$ children.

4.3. Operations on Binary Trees

Creating a Binary Tree

In order to create a binary tree, we can read the necessary information, presented in $preorder$ from the standard input. We have to distinguish between empty and nonempty (sub)trees, and thus we have to mark the empty trees with a special sign, such as '*'. For the tree of figure 4.1, the description (on the standard input) is: `ABD*G***CE**F*H**`
The node structure is:

```
typedef struct node_type
{
  char id; /* node name */
  label ; /* appropriate type for label */
  struct node_type *left, *right;
} NodeT;
```

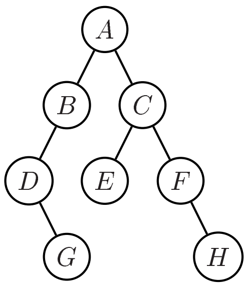


Figure 4.1.: A binary tree.

The construction of a binary tree may be achieved with a function containing the following code:

```
NodeT *createBinTree()
{
  NodeT *p;
  char c;

  /* read node id */
  scanf("%c", &c);
  if ( c == '*' )
    return NULL; /* empty tree; do nothing */
  else /* else included for clarity */
  { /* build node pointed to by p */
    p = ( NodeT *) malloc( sizeof( NodeT ));
    if ( p == NULL )
      fatalError( "Out␣of␣space␣in␣createBinTree" );
    /* fill in node */
    p->id = c;
    p->left = createBinTree();
    p->right = createBinTree();
  }
  return p;
}
```

The function $createBinTree$ may be invoked like this:

```
root = createBinTree();
```

Binary Tree Traversals

There are three kinds of traversals: preorder, inorder, and postorder. Listing 4.1 shows the C implementation for the operations of construction and traversal of a binary tree.

Listing 4.1: Construction and traversals of binary trees

```
#include <stdio.h>
```

```
#include <stdlib.h>

typedef struct node_type
{
  char id; /* node name */
  ... /* other useful info */
  struct node_type *left, *right;
} NodeT;

void fatalError( const char *msg )
{
 printf( msg );
 printf( "\n" );
 exit ( 1 );
}

void preorder( NodeT *p, int level )
/*
 * p = current node;
 * level = used for nice printing
*/
{
  int i;

  if ( p != NULL )
  {
    for ( i = 0; i <= level; i++ )
      printf( "␣" ); /* for nice listing */
    printf( "%2.2d\n", p->id );
    preorder( p->left, level + 1 );
    preorder( p->right, level + 1 );
  }
}
void inorder( NodeT *p, int level )
{
  int i;

  if ( p != NULL )
  {
    inorder( p->left, level + 1 );
    for ( i = 0; i <= level; i++ )
      printf( "␣" ); /* for nice listing */
    printf( "%2.2d\n", p->id );
    inorder( p->right, level + 1 );
  }
}
void postorder( NodeT *p, int level )
{
  int i;

  if ( p != NULL )
  {
    postorder( p->left, level + 1 );
    postorder( p->right, level + 1 );
    for ( i = 0; i <= level; i++ )
      printf( "␣" ); /* for nice listing */
    printf( "%2.2d\n", p->id );
  }
}
NodeT *createBinTree( int branch, NodeT *parent )
{
  NodeT *p;
  int id;
  /* read node id */
  if ( branch == 0 )
     printf( "Root␣identifier␣[0␣to␣end]␣=" );
  else
  if ( branch == 1 )
     printf( "Left␣child␣of␣%d␣[0␣to␣end]␣=",
             parent->id );
  else
     printf( "Right␣child␣of␣%d␣[0␣to␣end]␣=",
             parent->id );
  scanf("%d", &id);
  if ( id == 0 )
    return NULL;
  else
  { /* build node pointed to by p */
    p = ( NodeT *)malloc(sizeof( NodeT ));
    if ( p == NULL )
      fatalError( "Out␣of␣space␣in␣createBinTree" );
    /* fill in node */
    p->id = id;
    p->left = createBinTree( 1, p );
    p->right = createBinTree( 2, p );
  }
  return p;
}
int main()
{
  NodeT *root;

  root = createBinTree( 0, NULL );
  while ('\n' != getc(stdin));
  printf( "\nPreorder␣listing\n" );
  preorder( root, 0 );
  printf( "Press␣Enter␣to␣continue." );
  while ('\n' != getc(stdin));
  printf( "\nInorder␣listing\n" );
  inorder( root, 0 );
  printf( "Press␣Enter␣to␣continue." );
  while ('\n' != getc(stdin));
  printf( "\nPostorder␣listing\n" );
  postorder( root, 0 );
  printf( "Press␣Enter␣to␣continue." );
  while ('\n' != getc(stdin));
  return 0;
}
```

4.4. Completely Balanced Binary Trees

A *completely balanced* binary tree is a binary tree where the number of nodes in any of the left subtrees is at most one more than the number of nodes in the corresponding right subtree.
The following algorithm can be used to build such a tree with n nodes:

1. Designate a node to be the root of the tree.
2. Set the number of nodes in the left subtree to $n_{left} = \frac{n}{2}$.
3. Set the number of nodes in the right subtree to $n_{right} = n - n_{left} - 1$.
4. Repeat the steps recursively, taking the number of nodes to be n_{left} till there are no more nodes.
5. Repeat the steps recursively, taking the number of nodes to be n_{right} till there are no more nodes.

Listing 4.2: Construction and traversals of completely balanced binary trees

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node_type
```

```
{
  char id; /* node name */
  ... /* other useful info */
  struct node_type *left, *right;
} NodeT;

void fatalError( const char *msg )
{
 printf( msg );
 printf( "\n" );
 exit ( 1 );
}

void preorder( NodeT *p, int level )
/*
 * p = current node;
 * level = used for nice printing
*/
{
  int i;

  if ( p != NULL )
  {
    for ( i = 0; i <= level; i++ )
      printf( "␣" ); /* for nice listing */
    printf( "%2.2d\n", p->id );
    preorder( p->left, level + 1 );
    preorder( p->right, level + 1 );
  }
}
void inorder( NodeT *p, int level )
{
  int i;

  if ( p != NULL )
  {
    inorder( p->left, level + 1 );
    for ( i = 0; i <= level; i++ )
      printf( "␣" ); /* for nice listing */
    printf( "%2.2d\n", p->id );
    inorder( p->right, level + 1 );
  }
}
void postorder( NodeT *p, int level )
{
  int i;

  if ( p != NULL )
  {
    postorder( p->left, level + 1 );
    postorder( p->right, level + 1 );
    for ( i = 0; i <= level; i++ )
      printf( "␣" ); /* for nice listing */
    printf( "%2.2d\n", p->id );
  }
}
NodeT *creBalBinTree( int nbOfNodes )
{
  NodeT *p;
  int nLeft, nRight;

  if ( nbOfNodes == 0 ) return NULL;
  else
  {
    nLeft = nbOfNodes / 2;
    nRight = nLeft -1;
    p = ( NodeT * ) malloc( sizeof( NodeT ));
    printf( "\nNode␣identifier=" );
    scanf( "%d", &( p->id ));
    p->left = creBalBinTree( nLeft );
    p->right = creBalBinTree( nRight );
  }
  return p;
}
int main()
{
  NodeT *root;
  int nbOfNodes = 0;

  printf("\nNumber␣of␣nodes␣in␣the␣tree=");
  scanf("%d", &nbOfNodes );
  creBalBinTree( nbOfNodes );
  while ('\n' != getc(stdin));
  printf( "\nPreorder␣listing\n" );
  preorder( root, 0 );
  printf( "Press␣Enter␣to␣continue." );
  while ('\n' != getc(stdin));
  printf( "\nInorder␣listing\n" );
  inorder( root, 0 );
  printf( "Press␣Enter␣to␣continue." );
  while ('\n' != getc(stdin));
  printf( "\nPostorder␣listing\n" );
  postorder( root, 0 );
  printf( "Press␣Enter␣to␣continue." );
  while ('\n' != getc(stdin));
  return 0;
}
```

4.5. Arbitrary Trees

An *arbitrary* tree is a tree where the interior nodes have more than two children.
A node of such a tree is described by:

```
/* maximum number of children */
#define MAX_CHILD <appropriate value>
typedef struct node_type
{
  char id; /* node name */
  ... /* other useful info */
  struct node_type *children[MAX_CHILD];
} NodeT;
```

To build such a tree:

1. Read for each node, in *postorder*, fields: id, other useful info[1], and the number of children and push this info onto a stack.
2. When parent node is read, pop children nodes of the stack, fill children pointers in parent node, and then push a pointer to parent onto stack. Finally the only pointer on the stack will be a pointer to the root node, and the process stops.

Tree traversal is achieved in level order, according to the following:

- Use a queue to keep pointers to nodes to be processed. The queue is initially empty.
- Enqueue a root node pointer
- Dequeue a node, process node information, and enqueue pointers to the children of the currently processed node.
- Repeat previous step till the queue becomes empty.

[1]if defined

4.6. Lab.04 Assignments

Write code to:

4.6.1. Read a mathematical expression, in postfix form, as a character string, from a text file input. The allowed operators are the usual binary additive (+, -), multiplicative (*, /), and unary sign change (+, -). Build the tree for that expression. Every node should contain either an operator or an operand. A missing operand (for unary operators) is signaled by a # character in the input. Operands can be arbitrary alphabetic strings. .
I/O description. Input:

```
i:a␣c␣+␣d␣-␣#␣e␣-␣*
p:
```

where `i:`=signals the expression follows on input, `p:`=print the expression. Output: a pretty print the tree. E.g.

```
␣␣␣␣␣*
␣␣␣␣/␣\
␣␣␣-␣␣␣-
␣␣/␣\␣␣/␣\
␣+␣␣␣d␣#␣␣e
/␣␣\
a␣␣c
```

4.6.2. Read a mathematical expression, in prefix form, as a character string, from a text file input. The allowed operators are the usual binary additive (+, -), multiplicative (*, /), and unary sign change (+, -). Build the tree for that expression. Every node should contain either an operator or an operand. A missing operand (for unary operators) is signaled by a # character in the input. Operands can be arbitrary numbers, separated by spaces. Example input. For the expression `((1.05-(-55+22)))*10.3`, the tree representing the expression is:

```
␣␣␣␣␣␣␣␣␣*
␣␣␣␣␣␣␣␣/␣␣\
␣␣␣␣␣␣␣-␣␣␣␣10.3
␣␣␣␣␣/␣␣\
␣␣1.05␣␣␣+
␣␣␣␣␣␣␣␣/␣\
␣␣␣␣␣␣-55␣␣22
```

and the input is:

```
*␣-␣1.05␣+␣-55␣22␣10.3
e?
```

where `i:`=signals the expression follows on input, `e?`=evaluate the expression. Output: the value of the expression.

4.6.3. Construct a family tree organized as follows: the root node holds a parents's name (names must be unique) as a key; each name is unique. Then, after reading pairs of names from a file of your choice, output the relations between pairs or persons.
.6 .
I/O description. Input. `(`$< parentName >$ `(`$< childName > < childName >$ ... $< childName >$`))`
...
`?`$< name1 >$`,` $< name2 >$
The angle brackets ('<', '>') do not exist in the input, they just mark changeable amounts. All names (i.e. $< childName >, < parentName >, < name1 >, < name2 >$ are alphabetic strings with no spaces. '(', ')' are delimiters. '?' signals query about persons. Output is relation degree (i.e, child, parent, cousin, etc.). For the example tree in Figure 4.2 below, a part of the input would be:

```
(Matilde␣(Sebastian␣Leonor))
(Sebastian␣(Philip␣Mary))
(Philip␣(Frederick␣Ethel))
(Raul␣(Joaquim␣Julia))
...
?␣Matilde,␣Agnes
?␣Agnes,␣Matilde
?␣Joaquim␣Julia
```

and part of output would be:

```
Matilde␣is␣grand-grand-mother/father␣for␣Agnes
Agnes␣is␣grand-grand-son/daughter␣for␣Matilde
Joaquim␣is␣sibling␣for␣Julia
```

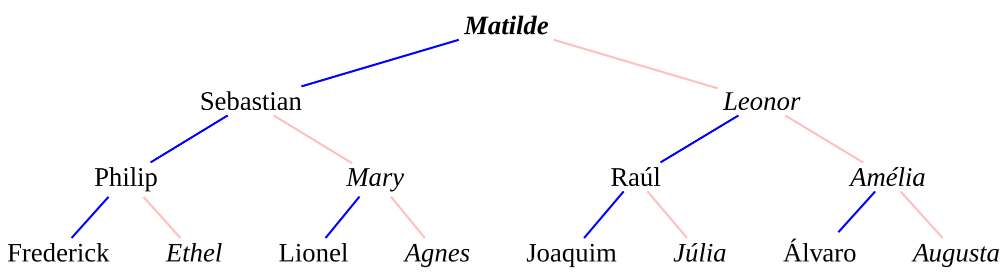


Figure 4.2.: Example Family Tree

4.6.4. Consider the inverse of Problem 4.3. Given the output data of Problem 4.3(i.e. relations among relatives) construct its input.

4.6.5. Transform a binary tree holding character strings into a doubly-linked list such a way that you can reconstruct the tree. The input is specified as for Problem 4.3. Output is prefix and infix traversals of the tree on two separate lines.

4.6.6. Write the code for representing trees as lists of children. Input is as for Problem 4.3. Output is prefix, postfix, and infix traversals of the tree on three separate lines.

4.6.7. Write the code for representing trees as lists of children. Input is as for Problem 4.3. Output is the tree in level order, i.e. the first line is the root alone; the second is the root identification, followed by a colon, and all its children, separated one space each; the third line is the identification of the leftmost child of the root, followed by a colon, and then all its children, separated one space each; the fourth line is the second child of the root followed by a colon, and then all its children, separated one space each, and so on up to the lowest level. .6

4.6.8. Implement AVL trees with character string keys. Input are commands for insertion, deletion, and searching the tree. Commands are always specified in the first column. Use `i` for insertion, `d` for deletion `f` for find. Each command is followed by a space and an argument – a key. Output of insert and delete are the prefix and infix traversals of the tree. Output for $find$ is the input key followed by a space and `found` or `not found` as suitable..6

4.6.9. Implement 2-3 trees holding birthday records. (Records contain a person's name and birthday). The key here is the birthday (given as yyyymmdd, e.g. 19991011). Input and output is similar to Problem 4.8.

4.6.10. Implement binary trees holding books in a library. The data for each book is: ISBN (this is the key), Author, Title, Status (borrowed, on-shelf). Operations are marked in the input as follows: `a` for adding a new book; `b` to borrow a book; `r` for returning a book; `d` for removing a book form the library tree; `l` for printing the contents of the whole library. Command `a` is followed by three elements: the ISBN, author's name enclosed in double quotes, and boot title enclosed in double quotes, all separated by commas. Borrow and return commands require the ISBN alone. Listing of shelves contents must print ISBN, author, title and status for each book, in a format similar to input.

4.6.11. Implement the priority queue ADT using with a POT using an array as presented in Lecture 3. Input data are numbers. Operations are specified by letters `i` for $insert$, and `d` for $deleteMin$. Your program must output a printout of the contents of the tree after each operation, in level order similar to that of Problem 4.8.

4.6.12. Implement the heap ADT using an array as presented in Lecture 3. Input data are strings. Operations are specified by letters `i` for $insert$, and `d` for $deleteMax$. Your program must output a printout of the contents of the tree after each operation, in level order similar to that of Problem 4.8.

5. Binary Search Trees

5.1. Purpose

In this lab session you will experiment with *binary search trees* (BSTs). The major operations on binary search trees are: insert, find, delete and traversals. A specific kind of BSTs are the optimal BSTs, and you will study them here, as well.

5.2. Brief Theory Reminder

Binary Search Tree Structure

BSTs are used quite frequently for fast storing and retrieval of information, based on keys. The keys are stored in the nodes, and must be distinct.
A structure for a BST node may be:

```
typedef struct node_type
{
  KeyType key;
  ElementType info;
  struct node_type *left, *right;
} NodeT;
```

The root of the tree could then be declared as a global variable:

```
NodeT *root;
```

The resulting structure of a BST depends upon the order of node insertions.

Inserting a Node in a BST

A BST is build by inserting new nodes into it. This can be done by performing the following steps:

5.2.1. If the tree is empty, then create a new node, the root, with key value *key*, and empty left and right subtrees.
5.2.2. If root key is *key*, then insertion is not possible, as the keys must be distinct – nothing to do.
5.2.3. If the given key, say *key*, is less than the key stored at the root node, then continue with the first step, for the left subtree.
5.2.4. If the given key, say *key*, is bigger than the key stored at the root node, then continue with the first step, for the right subtree.

Insertion can be achieved non-recursively using the procedure presented below:

```
typedef int KeyType;
typedef int ElementType;
/* for simplicity. Both should occur
   before NodeT declaration */

void nInsert( KeyType givenKey)
{
  NodeT *p, *q;

  /* build node */
  p = ( NodeT *) malloc ( sizeof ( NodeT ));
  p->key = givenKey;
  /* the info field should be filled in here */
  p->left = p-> right = NULL; /* leaf node */
  if ( root == NULL )
  { /* empty tree */
    root = p;
    return;
  }
  /* if we reach here then the tree is not empty;
     look for parent node of node pointed to by p */
  q = root;
  for ( ; ; )
  {
    if ( givenKey < q->key )
    { /* follow on left subtree */
      if ( q-> left == NULL )
      { /* insert node here */
        q->left = p;
        return;
      }
      /* else */
        q = q->left;
    }
    else
    if ( givenKey > q->key )
    { /* follow on right subtree */
      if ( q-> right == NULL )
      { /* insert node here */
        q->right = p;
        return;
      }
      /* else */
      q = q->right;
    }
    else
    { /* key already present;
         could write a message... */
      free( p );
    }
  }
}
```

Finding a Node of a Given Key in a BST

The algorithm for finding a node is quite similar to the one for insertion. One important aspect when looking for information in a BST node is the amount of time taken. The number of comparisons would be optimal if there would be at most $\lceil \log_2 n \rceil$ where n is the number of nodes stored. The worst case is encountered when insertion is executed with the nodes sorted in either ascending or descending order. In that case the tree turn out to be a list.
A *find* function may be implemented as[1]:

```
NodeT *find( NodeT * root, KeyType givenKey )
{
  NodeT *p;

  if ( root == NULL ) return NULL; /* empty tree */
  for ( p = root; p != NULL; )
  {
    if ( givenKey == p->key ) return p;
    else
    if ( givenKey < p->key ) p = p->left;
    else                     p = p->right;
  }
  return NULL; /* not found */
}
/* invoke with: q = find( root, key ); */
}
```

Deleting a Node in a BST

When a node is deleted, the following situations are met:

5.2.1. The node to delete is a *leaf*. Child node pointer in parent node (left or right) must be set to zero (NULL).
5.2.2. The node to delete has only one child. In this case, in parent node of the one to be deleted its address is replaced by the address of its child.

[1]see remarks on KeyType from previous section

5.2.3. The node to be deleted has two descendants. Replace the node to be deleted with either the leftmost node of right subtree or with the rightmost node of left subtree.

When deleting, we first have to look for the node to delete (based on some *key*) and then evaluate the situation according to the previous discussion.

Deleting a Whole BST

When total removal of the nodes of a BST is required, that can be accomplished by a postorder traversal of the tree and node-by-node removal using a recursive function like this:

```
void delTree( NodeT *root )
{
  if ( root != NULL )
  {
    delTree( root ->left );
    delTree( root ->right );
    free( root );
  }
}
```

BST Traversals

As with any tree, there are three systematic traversals: preorder, inorder and postorder traversals. Here is some code for traversals:

```
void preorder( NodeT *p )
{
  if ( p != NULL )
  {
    /* code for info retrieval here */
    preorder( p->left );
    preorder( p->right );
  }
}
void inorder( NodeT *p )
{
  if ( p != NULL )
  {
    inorder( p->left );
    /* code for info retrieval here */
    inorder( p->right );
  }
}
void postorder( NodeT *p )
{
  if ( p != NULL )
  {
    postorder( p->left );
    postorder( p->right );
    /* code for info retrieval here */
  }
}
```

Optimal BSTs

The length of the path traversed when looking for a node of key x, in a BST, is:

$$\text{length of path} = \begin{cases} h_f & \text{if successful} \\ h_{nf} + 1 & \text{if not found} \end{cases}$$

where h_f is the level of the node containing the given key, and h_{nf} is the level of the last node met before deciding that the node is not present.
Let $S = \{c_1, c_2, \dots, c_n\}$ be the set of keys leading to success, in ascending order, i.e. $c_1 < c_2 < \dots < c_n$ and let p_i be the probability of looking for key c_i, for $i = 1, 2, \dots, n$. If we denote C the set of possible keys, the $C - S$ is the set of keys which lead to unsuccessful search. We can partition this set into subsets:

k_0	the set of keys less than c_1
k_n	the set of keys greater than c_n
k_i, for $1 \leq i \leq n$	the set of keys with the range (c_i, c_{i+1})

Let now q_i be the probability of looking for a key in the set k_i. Each key k_i involves the same search path; then the length of this path will be h'_i. If we denote L the average search path and call it the cost of the tree, we have:

$$L = \sum_{i=1}^{n} p_i h_i + \sum_{j=0}^{n} q_j h'_i$$

provided that:

$$\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1$$

An *optimal* tree is a BST in which for given $p_i,\ q_j$ the cost L is *minimal*. Optimal BSTs are not subject to insertions and deletions.
Occurrence frequencies should be used instead of p_i and q_j when minimizing the cost function L. Those frequencies will be found by using test data runs.
If we denote by $A_{i,j}$ an optimal tree with nodes $c_{i+1}, c_{i+2}, \ldots, c_j$, then the *weight* $w_{i,j}$ of $A_{i,j}$ is:

$$w_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k$$

$g_{i,j}$ can be calculated as:

$$\begin{cases} w_{i,i} = q_i & \text{for } 0 \leq i \leq n \\ w_{i,j} = w_{i,j-1} & \text{for } 0 \leq i < j \leq n \end{cases}$$

It follows that the cost of the optimal tree $A_{i,j}$ can be evaluated as:

$$\begin{cases} c_{i,i} = w_{i,i} & \text{for } 0 \leq i \leq n \\ c_{i,j} = w_{i,j} + \min\limits_{i<k\leq j}(c_{i,k-1} + c_{k,j}) & \text{for } 0 \leq i < j \leq n \end{cases}$$

Let $r_{i,j}$ be the value of k for which a minimal $c_{i,j}$ is obtained. The node of key $c[r_{i,j}]$ will be the root of the sub-optimal tree $A_{i,j}$, with subtrees $A_{i,k-1}$ and $A_{k,j}$, where $k = r_{i,j}$. Computing the values of the matrix C is $O(n^3)$, but it has been demonstrated that the running time can be reduced to $O(n^2)$.
In order to build an optimal tree you can use function *buildOptTree*, given below:

```
NodeT *buildOptTree( int i, int j )
{
  NodeT *p;

  if ( i == j ) p = NULL;
  else
  {
    p = ( NodeT * ) malloc( sizeof ( NodeT ));
    p->left = buildOptTree( i, r[ i ][ j ] - 1 );
    p->key = keys[ roots[ i ][ j ] ];
    p->right = buildOptTree( r[ i ][ j ], j );
  }
  return p;
}
```

5.3. Code Samples

Listing 5.1: Binary search trees.

```
#include <stdio.h>
#include <stdlib.h>

#define LEFT 1
#define RIGHT 2
```

```
typedef struct node
{
  int key;
  struct node *left, right;
} NodeT;

NodeT root;

void fatalError( const char *msg )
{
 printf( msg );
 printf( "\n" );
 exit ( 1 );
}

void preorder( NodeT *p, int level )
/*
 * p = current node;
 * level = used for nice printing
*/
{
  int i;

  if ( p != NULL )
  {
    for ( i = 0; i <= level; i++ ) printf( "␣" ); /* for nice listing */
    printf( "%2.2d\n", p->key );
    preorder( p->left, level + 1 );
    preorder( p->right, level + 1 );
  }
}
void inorder( NodeT *p, int level )
{
  int i;

  if ( p != NULL )
  {
    inorder( p->left, level + 1 );
    for ( i = 0; i <= level; i++ ) printf( "␣" ); /* for nice listing */
    printf( "%2.2d\n", p->key );
    inorder( p->right, level + 1 );
  }
}
void postorder( NodeT *p, int level )
{
  int i;

  if ( p != NULL )
  {
    postorder( p->left, level + 1 );
    postorder( p->right, level + 1 );
    for ( i = 0; i <= level; i++ ) printf( "␣" ); /* for nice listing */
    printf( "%2.2d\n", p->key );
  }
}
void insert( int key ) /* non recursive version of insert */
{
  NodeT *p, *q;

  p = ( NodeT *) malloc( sizeof( NodeT ));
  p->key = key;
  p->left = p->right = NULL;
  if ( root == NULL )
  {
    root = p;
    return;
  }
  q = root;
  for ( ; ; )
  {
    if ( key < q->key )
    {
      if ( q->left == NULL )
      {
        q->left = p;
        return;
      }
      else q = q->left;
    }
    else
    if ( key > q->key )
    {
      if ( q->right == NULL )
      {
        q->right = p;
        return;
      }
      else q = q->right;
    }
    else
    { /* keys are equal */
      printf( "\nNode␣of␣key=%d␣already␣exists\n",
              key );
      free( p );
      return;
  }
NodeT *recInsert( NodeT *root, int key ) /* recursive version of insert */
{
  NodeT *p, *q;

  if ( root == NULL )
  {
    p = ( NodeT *) malloc( sizeof( NodeT ));
    p->key = key;
    p->left = p->right = NULL;
    root = p;
    return;
  }
  else
  {
    if ( key < root->key )
      root->left = recInsert( root ->left, key );
    else
    if ( key > root->key )
      root->right = recInsert( root ->right, key );
    else /* key already there */
      printf( "\nNode␣of␣key=%d␣already␣exists\n",
              key );
  }
  return root;
}
NodeT *find( NodeT *root, int key )
{
  NodeT *p;

  if ( root == NULL ) return NULL;
  p = root;
  while ( p != NULL )
  {
    if ( p -> key == key )
      return p; /* found */
    else
    if ( key < p->key ) p = p->left;
    else                p = p->right;
  }
  return NULL; /* not found */
}
NodeT *delNode( NodeT *root, int key )
{
  NodeT *p; /* points to node to delete */
  NodeT *pParent; /* points to parent of p */
  NodeT *pRepl; /* points to node that will replace p */
  NodeT *pReplParent; /* points to parent of node that will replace p */
  int direction; /* LEFT, RIGHT */
  if ( root == NULL ) return NULL; /* empty tree */
  p = root;
  pParent = NULL;
  while ( p != NULL && p->key != key )
  {
    if ( key < p->key )
    { /* search left branch */
      pParent = p;
      p = p->left;
      direction = LEFT;
    }
    else
    { /* search right branch */
      pParent = p;
      p = p->right;
      direction = RIGHT;
    }
  }
  if ( p == NULL )
  {
    printf( "\nNo node of key value=%d\n", key );
    return root;
  }
  /* node of key p found */
  if ( p->left == NULL )
    pRepl = p->right; /* no left child */
  else
  if ( p->right == NULL
    pRepl = p->left; /* no right child */
  else
  { /* both children present */
    pReplParent = p;
    pRepl = p->right; /* search right subtree */
    while ( pRepl->left != NULL )
    {
      pReplParent = pRepl;
      pRepl = prepl->left;
    }
    if ( pReplParent != p )
    {
      pReplParent->left = pRepl->right;
      pRepl->right = p->right;
    }
    pRepl->left = p->left;
  }
  free( p );
  printf( "\nDeletion of node %d completed\n", key);
  if ( pParent == NULL )
    return pRepl; /* original root was deleted */
  else
  {
    if (direction == LEFT )
      pParent->left = pRepl;
    else
      pParent->right = pRepl;
    return root;
  }
}
void delTree( NodeT *root )
{
  if ( root != NULL )
  {
    delTree( root->left );
    delTree( root->right );
    free( root );
  }
}
void main()
{
  NodeT *p;
  int i, n, key;
  char ch;

  printf( "Choose␣insert␣function:\n" );
  printf( "\t[R]ecursive\n\t[N]onrecursive:␣" );
  scanf( "%c", &ch );
  printf( "Number␣of␣nodes␣to␣insert=␣" );
  scanf( "%d", &n );
  root = NULL;
  for ( i = 0; i < n; i++ )
  {
    printf( "\nKey=␣" );
    scanf( "%d", &key );
    if ( ch == 'R' || ch == 'r' )
      recInsert( root, key );
    else
      insert( key );
  }
  printf( "\nPreorder␣listing\n" );
  preorder( root, 0 );
  printf( "Press␣Enter␣to␣continue." );
  while ('\n' != getc(stdin));
  printf( "\nInorder␣listing\n" );
  inorder( root, 0 );
  printf( "Press␣Enter␣to␣continue." );
  while ('\n' != getc(stdin));
  printf( "\nPostorder␣listing\n" );
  postorder( root, 0 );
  printf( "Press␣Enter␣to␣continue." );
  while ('\n' != getc(stdin));
  printf( "Continue␣with␣find␣(Y/N)?␣" );
  scanf( "%c", &ch );
  while ( ch == 'Y' || ch == 'y' )
  {
    printf( "Key␣to␣find=␣" );
    scanf( "%d", &key );
    p = find( root, key );
    if ( p != NULL )
      printf( "Node␣found\n" );
    else
      printf( "Node␣NOT␣found\n" );
    while ('\n' != getc(stdin));
    printf( "Continue␣with␣find␣(Y/N)?␣" );
    scanf( "%c", &ch );
  }
  printf( "Continue␣with␣delete␣(Y/N)?␣" );
  scanf( "%c", &ch );
  while ( ch == 'Y' || ch == 'y' )
  {
    printf( "Key␣of␣node␣to␣delete=␣" );
    scanf( "%d", &key );
    root = delete( root, key );
    inorder( root, 0);
    while ('\n' != getc(stdin));
    printf( "Continue␣with␣delete␣(Y/N)?␣" );
    scanf( "%c", &ch );
  }
  printf( "Delete␣the␣whole␣tree␣(Y/N)?␣" );
  scanf( "%c", %ch );
  if ( ch == 'Y' || ch == 'y' )
  {
    delTree( root );
    root = NULL;
    printf("␣Tree␣completely␣removed\n" );
  }
  printf( "Press␣Enter␣to␣exit␣program" );
  while ('\n' != getc(stdin));
}
```

Listing 5.2: Optimal search trees.

```
#include <stdio.h>
#include <stdlib.h>

#define MAXN 25

typedef struct node
{
  char key;
  struct node *left, right;
} NodeT;

char keys[ MAXN ]; /* keys c1, c2, ..., cn */
int p[ MAXN ]; /* p holds key search freq. */
int q[ MAXN ]; /* q holds key relative search freq. */
int roots[ MAXN ][ MAXN ] /* optimal subtrees roots */

void printMatrix( const char *msg,
                  int matrix[ MAXN ][ MAXN ] )
{
  int i, j;
  printf{ "\mMatrix␣of␣%s\n", msg )
  for ( i = 0; i <= n; i++ )
    {
      for ( j = i; ; j < n; j++ )
        printf("%d␣", matrix[i][j] );
      printf( "\n" );
    }
  printf( "Press␣Enter␣to␣continue" );
  while ('\n' != getc(stdin));
}
void detTreeStruct( int n, float *avgPath )
/* determine tree structure */
{
  int c[MAXN]{MAXN]; /* subtrees costs */
  int w[MAXN]{MAXN]; /* subtrees weights */
  int i, j, k, l, m;
  int x, min;
  /* calculate matrix of weights */
  for ( i = 0; i <= n; i++ )
  {
    w[i][i] = q[i];
    for ( j = i + 1; j < n; j ++ )
      w[i][j] = w[i][j-1] + p[j] + q[j];
  }
  /* calculate matrix of costs */
  for ( i = 0; i <= n; i++ )
    c[i][i] = w[i][i];
  for ( i = 0; i <= n - 1; i++ )
  {
    j = i + 1;
    c[i][j] = c[i][i] + c[i][j] + w[i][j];
    roots[i][j] = j;
  }
  for ( l = 2; l <= n; l++ )
    for ( i = 0; i <= n -l ; i++ )
    {
      min = 32000;
      for ( k = i + 1; k <= l + i; k++ )
      {
        x = c[i][k-1] + c[k][l+i];
        if ( x < min )
        {
          min = x;
          m = k;
        }
      }
      c[i][l+i] = min + w[i][l+i];
      roots[i][l+i] = m;
    }
  printMatrix( "weights", w );
  printMatrix( "costs", c );
  printMatrix( "roots", r );
  printf( "c[0][n]=%ld\nw[0][n]=%ld\n", c[0][n], w[0][n] );
  printf( "Press Enter to continue" );
  while ('\n' != getc(stdin));
  *avgPath = c[0][n] / (float)w[0][n];
}
NodeT *buildTree( int i, int j )
/* builds an optimal search tree */
{
  NodeT *p;

  if ( i == j ) p = NULL;
  else
  {
    p = ( NodeT * ) malloc( sizeof( NodeT * ));
    p->left = buildTree( i, roots[i][j]-1 );
    p->key = keys[ roots[i][j] ];
    p->right = buildTree( roots[i][j], j );
  }
  return p;
}
void inorder( NodeT *p, int level )
{
  int i;

  if ( p != NULL )
  {
    inorder( p->left, level + 1 );
```

```
      for ( i = 0; i <= level; i++ )
        printf( "␣" ); /* for nice listing */
      printf( "%c d\n", p->key );
      inorder( p->right, level + 1 );
    }
}
void main()
{
    NodeT *root;
    int n; /* number of keys */
    float avgPath;
    int i;

    printf( "Number␣of␣keys=␣");
    scanf( "%d", &n );
    /* Read keys and key search freq. */
    for ( i = 0; i <= n; i++ )
    {
      printf( "Key[%d]=", i );
      scanf("%c", &keys[i] );
      while ('\n' != getc(stdin));
      printf( "Frequency[%d]=", i );
      scanf("%d", &p[i] );
      while ('\n' != getc(stdin));
    }
    /* Read search freq. between keys */
    for ( i = 0; i <= n; i++ )
    {
      printf( "q[%d]=", i );
      scanf("%c", &q[i] );
      while ('\n' != getc(stdin));
    }
    detTreeStruct( n, &avgPath );
    printf( "Average␣path=%f\n", avgPath );
    printf( "Press␣Enter␣to␣continue" );
    while ('\n' != getc(stdin));
    root = buildTree( 0, n );
    printf( "Press␣Enter␣to␣continue" );
    while ('\n' != getc(stdin));
}
```

5.4. Lab.05 Assignments

5.4.1. You have to maintain information for a shop owner. For each of the products sold in his/hers shop the following information is kept: a unique code, a name, a price, amount in stock, date received, expiration date. For keeping track of its stock, the clerk would use a computer program based on a search tree data structure. Write a program to help this person, by implementing following the following operations:

- Insert an item with all its associated data.
- Find an item by its code, and support updating of the item found.
- List valid items in lexicographic order of their names.
- List expired items in lexicographic order of their names.
- List all items.
- Delete an item given by its code.
- Delete all expired items.
- Create a separate search tree for expired items.
- Save stock in file `stock.data`.
- Exit

If file `stock.data` exists, your program must automatically load its contents. .

I/O description. This program should be interactive.

You have to maintain information for school classes. For each of the students in a class the following information is kept: a unique code, student name, birth date, home address, id of class he is currently in, date of enrollment, status (undergrad, graduate). For keeping track of the students, the school secretary would use a computer program based on a search tree data structure. Write a program to help the secretary, by implementing following the following operations:

- Insert an item with all its associated data.
- Find an student by his/hers unique code, and support updating of the student info if found.
- List students by class in lexicographic order of their names.
- List all students in lexicographic order of their names.
- List all graduated students.
- List all undergrads by their class in lexicographic order of their names.
- Delete an student given by its code.
- Delete all graduates.
- Save all students in file `student.data`.
- Exit

If file `student.data` exists, your program must automatically load its contents. .
I/O description. This program should be interactive.

5.4.2. Build an optimal BST (binary search tree) to hold C language keywords in its nodes. In order to determine frequencies p_i and q_i use code samples from the lab book. .
I/O description. Keywords and frequencies may be read from a file with a name of your choice (e.g. ’keyword.data’). The menu must include the following:

- Printing the tree in preorder.
- Listing $keyword, frequency$ pairs for each node.
- Finding a $keyword$. If found/not found your program should output the path traversed in determining the answer, followed by `yes` if found or `no` if not found.

Hint. To determine frequencies you may use a list sorted in frequency order.

5.4.3. Build an optimal BST (binary search tree) to hold unextended Pascal language keywords in its nodes. The keyword list can be found e.g. at `http://pascal-central.com/ppl/chapter3.html#Unextended`. In order to determine frequencies p_i and q_i use code samples downloaded from the Internet. .
I/O description. Keywords and frequencies may be read from a file with a name of your choice (e.g. ’keyword.data’). The menu must include the following:

- Printing the tree in postorder.
- Listing $keyword, frequency$ pairs for each node.
- Finding a $keyword$. If found/not found your program should output the path traversed in determining the answer, followed by `yes` if found or `no` if not found.

Hint. To determine frequencies you may use a list sorted in frequency order.

5.4.4. Write a program to illustrate operations on a BST holding numeric keys. The menu must include:

- Insert
- Delete
- Find
- Show

Allow a maximum depth of 5 (note that root is at depth 0), and a maximum number of nodes on the last level of 32. After each step a print of the tree should be shown as below:

```
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> i 20
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> i 10
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> i 11
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> i 1
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> i 23
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> i 26
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> s
     20
    / \
   10  23
  / \   \
 01  11  26
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> d 12
12 not found
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> d 20
     23
    / \
   10  26
  / \
 01  11
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> d J
     23
    / \
   11   26
  /
 01
```

5.4.5. Write a program to illustrate operations on a BST holding alphabetic uppercase letter keys. The menu must include:

- Insert
- Delete
- Find
- Show

Allow a maximum depth of 5 (note that root is at depth 0), and a maximum number of nodes on the last level of 32. After each step a print of the tree should be shown as below:

```
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> i V
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> i J
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> i K
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> i A
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> i X
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> i Z
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> s
     V
    / \
   J   X
  / \   \
 A   K   Z
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> d M
M not found
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> d V
     X
    / \
   J   Z
  / \
 A   K
I. Insert D. Delete  F. Find  S. Show  Q. Quit
> d J
     X
    / \
   K   Z
  /
 A
```

5.4.6. Write a program to illustrate the merge operation of two BSTs holding numeric keys between 1 and 99. The menu must include:

- Creation of the trees with a random number of random keys with at most 16 keys with values between 1 and 99.
- Merge

After each step a print of the tree should be shown as below:

```
C. Create trees
M. Merge
Q. Quit
> C
Tree 1:
21
--
Tree 1:
  21
 /
7
--
Tree 1:
    21
   /
  7
   \
    19
--
Tree 1:
    21
   / \
  7   52
   \
    19
--
Tree 1:
    21
   / \
  7   52
   \
    19
   /
  10
--
Tree 2:
17
--
Tree 2:
17
  \
   42
--
Tree 2:
17
  \
   42
  /
19
--
```

```
C. Create trees
M. Merge
Q. Quit
> M
Merged trees
17
--
Merged trees
   17
  /
21
--
Merged trees
   17
  / \
21   42
--
Merged trees
     17
    / \
  21   42
  /
19
--
Merged trees
     17
    / \
  21   42
  /     \
19       52
--
Merged trees
        17
       / \
     21   42
     /     \
   19       52
   /
  7
--
Merged trees
        17
       / \
     21   42
     /     \
   19       52
   /
  7
   \
   10
--
```

6. Hash Tables

6.1. Purpose

In this lab session we will work with hash tables. We will study the main operations on hash tables, i.e. create, insert, find and retrieve items.

6.2. Brief Theory Reminder

Table Types

A *table* is a collection of elements of the same kind, which are identified by *keys*. The elements stored in a hash table are also called *records*.
Tables may be organized in two ways:

- *Fixed* tables, where the number of elements to be stored is known at the moment of creation.
- *Dynamic* tables, with a variable number of elements.

A reserved keyword table for a programming language is an example of a fixed table. It is typically arranged as a pointer table, where pointers indicate reserved keywords in alphabetic order. Binary search is used when looking for a particular keyword. Dynamic tables may be organized as: singly linked lists, search trees or hash tables. If dynamic tables are arranged as lists, then search is performed linearly, which slows down the process. A search tree reduces the time to find an element. Hash tables are used if the keys are alphanumeric because comparisons are time-consuming for long alphanumeric keys (remember that hash tables construct shorter keys).

Hash Functions

A *hash function* is a function which transforms a key into a natural number called a *hash value*, i.e.

$$f : K \to H,$$

where K is the set of keys and H is a set of natural numbers. Function f is a many-to-one function. If two different keys, say k_1 and k_2, $k_1 \neq k_2$ have the same hash value, i.e. $f(k_1)f(k_2)$ then the two keys are said to *collide* and the corresponding records are called *synonyms*. Two restrictions are imposed on f:

6.2.1. For any $k \in K$ the value should be obtained as fast as possible.
6.2.2. It must minimize the number of collisions.

An example of hash function is:

$$f(k) = \gamma(k) \text{ modulus } B,$$

where γ is a function which maps a key to a natural number, and B is a natural number, possibly prime. The expression of function γ depends on the keys. If the keys are numeric, the $\gamma(k) = k$. The simplest function γ on alphanumeric keys is the sum of the (ASCII) codes for each character of the key. A simple function on strings is:

```
#define B ? /*suitable value for B */
int f( char *key )
{
  int i, sum;
  sum = 0;
  for ( i = 0; i < strlen( key ); i++ )
    sum += key[ i ];
  return( sum % B );
}
```

Collision Resolution

Collision resolution may be achieved as follows: insert all records with colliding keys in a singly-linked list. There will be a number of lists, called *buckets* one for each hash value. So, an open hash table is modelled as figure 6.1 shows.
A cell in a bucket may be described by:

```
typedef struct cell
{
  char *key;
  /* other useful info */
  struct cell *next;
} NodeT;
```

Then, a bucket table header is:

```
NodeT *BucketTable[ B ];
```

Initially, all the buckets are empty, i.e.:

```
for ( i = 0; i < B; i++ )
  Buckettable[ i ] = NULL;
```

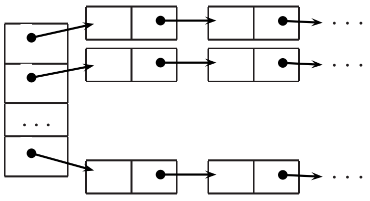


Figure 6.1.: A model of open hashing.

The steps needed to find a record of hash value key in a hash table are:

6.2.1. Determine the hash value for the key $h = f(key)$.

6.2.2. Linearly search the bucket for hash value h:

```
p = BucketTable[ h ];
while ( p != NULL )
{
   if (strcmp( key, p->key ) == 0 )
     return p;
  p = p->next;
}
return NULL; /* not found */
```

Insertion in a hash table takes the following steps:

6.2.1. Make a new node pointed to by p and fill it:

```
p = ( NodeT *) malloc( sizeof( NodeT ));
fillNode( p );
```

6.2.2. Calculate the hash value:

```
h = f( p->key );
```

6.2.3. If the bucket table header entry is empty, insert as first list element:

```
if ( BucketTable[ h ] == NULL )
{
  BucketTable[ h ] = p;
  p->next = NULL;
```

Otherwise check if the record to insert is not already present. If already there we have two choices: to apply some operation such as delete or update or signal a "double key" error. If the record is not present, insertion as first node in the bucket may be effected:

```
q = find( p->key );
if ( q == 0 )
{ /* not found. Insert it */
  p -> next = BucketTable[ h ];
  Buckettable[ h ] = p;
}
else /* double key */
  processRec( p, q );
```

A hash table is build by repeatedly insertion nodes.
A complete list of a hast table contents may be achieved with:

```
for ( i = 0; i < B; i++ )
  if ( BucketTable[ i ] != NULL )
  {
    printf( "Bucket␣for␣hash␣value␣%d\n", i );
    p =BucketTable[ i ];
    while ( p != NULL )
    {
      showNode( p );
      p = p->next;
    }
  }
```

6.3. Lab.06 Assignments

Write code in C or C++ to:

6.3.1. Write a program to manage an hash table, using *open addressing*, where the keys are student full names. Your code should provide create, insert, find and delete operations on that table. .
I/O description. Input:

```
i<name>
d<name>
f<name>
l
```

`i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters <, and > are *not* part of the input. Use a hash function suitable for character strings and motivate your answer in a comment stored in the heading area of your hash table processing routines. Output for find should be `yes`, followed by the table index if found or `no` if not found.

6.3.2. Write a program to manage an hash table, using *open addressing*, where the keys are book ISBNs. Your code should provide create, insert, find and delete operations on that table. .
I/O description. Input:

```
i<name>
d<name>
f<name>
l
```

`i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters <, and > are *not* part of the input. Use a hash function suitable for character strings and motivate your answer in a comment stored in the heading area of your hash table processing routines. Output for find should be `yes`, followed by the table index if found or `no` if not found.

6.3.3. Write a program to manage an hash table, using *collision resolution by chaining*, where the keys are student full names. Your code should provide create, insert, find and delete operations on that table. .
I/O description. Input:

```
i<name>
d<name>
f<name>
l
```

`i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters <, and > are *not* part of the input. Use a hash function suitable for character strings and motivate your answer in a comment stored in the heading area of your hash table processing routines. Output for find should be `yes`, followed by the table index if found or `no` if not found.

6.3.4. Write a program to manage an hash table, using *open addressing*, with character string keys, where the hash function should be selectable before each run. The methods you should use in building your has functions are linear, quadratic and double hashing (cf. Lecture 3). Your code should provide create, insert, find and delete operations on that table. .
I/O description. Input:

```
f<number>
i<name>
```

```
d<name>
f<name>
l
```

`f<number>`=select the hash function numbered with <number>, `i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters `<`, and `>` are *not* part of the input. Output for find should be `yes`, followed by the table index if found or `no` if not found.

6.3.5. Write a program to manage an hash table, using *open addressing*, with numeric long integer keys, where the hash function should be selectable before each run. The methods you should use in building your has functions are linear, quadratic and double hashing (cf. Lecture 3). Your code should provide create, insert, find and delete operations on that table. .
I/O description. Input:

```
f<number>
i<name>
d<name>
f<name>
l
```

`f<number>`=select the hash function numbered with <number>, `i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters `<`, and `>` are *not* part of the input. Output for find should be `yes`, followed by the table index if found or `no` if not found.

6.3.6. Write a program to manage an hash table, using *collision resolution by chaining*, with numeric long integer keys, where the hash function should be selectable before each run. The methods you should use in building your has functions are memory address, integer cast, and component sum (cf. Lecture 3). Your code should provide create, insert, find and delete operations on that table. .
I/O description. Input:

```
f<number>
i<name>
d<name>
f<name>
l
```

`f<number>`=select the hash function numbered with <number>, `i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters `<`, and `>` are *not* part of the input. Output for find should be `yes`, followed by the table index if found or `no` if not found.

6.3.7. Write a program to manage an hash table, using *collision resolution by chaining*, with numeric long integer and string keys, where the hash function should use polynomial accumulation (cf. Lecture 3). Your code should provide create, insert, find and delete operations on that table. .
I/O description. Input:

```
i<name>
d<name>
f<name>
l
```

`i<name>`=insert `<name>`, `d<name>`=delete `<name>`, `f<name>`=find `<name>` in the table; `l`=list table contents. Note that the characters `<`, and `>` are *not* part of the input. Output for find should be `yes`, followed by the table index if found or `no` if not found.

---

7. Graph Representations and Traversals

7.1. Purpose

In this lab session we will implement some graph representation methods and graph traversals.

7.2. Brief Theory Reminder

Basic Notions

A *directed graph* (*digraph*, for short) $G = (V, E)$ is an ordered pair of elements of two sets: V is a set of *vertices* (or nodes) and $E : V \times V$ is a set of arcs.

An *arc* (also called an *edge*) is an ordered pair of vertices (v, w); v is called the *tail* and w the *head* of the arc. We say that arc $v \rightarrow w$ is *from v to w*, and that s*w is adjacent to v*.

A *path* in a digraph is a sequence of vertices $v_1, v_2, ..., v_n$ such that $v_1 \rightarrow v_2$, $v_2 \rightarrow v_3$, . . ., $v_{n-1} \rightarrow v_n$, are arcs, i.e $(v_i, v_{i+1}) \in E$. This path is *from* vertex v_1 to vertex v_n and *passes through* vertices $v_2, v_3, \ldots, v_{n-1}$, and ends at vertex v_n. There is a path of length 0 from any vertex to itself.

A path is *simple* if all vertices on the path, except possibly the first and the last, are distinct.

A simple *cycle* is a simple path of length at least one that begins and ends at the same vertex.

A *labelled* directed graph is a digraph where every arc and/or vertex has a label associated with it. The label may be a name, a cost or an arbitrary value.

A digraph is *strongly connected* if for any two vertices v and w there is a path from v to w (and, of course, one from w to v).

A graph $G' = (V', E')$ is a *subgraph* of a graph G if $V' \subset V$ and $E' \subset E$. The subgraph *induced* by $V' \subset V$ is the graph $G' = (V', E \cap (V' \times V'))$.

An *undirected graph* (or simply a *graph*) $G = (V, E)$ is a pair of a set of vertices, V, and a set of arcs (or *edges*), E. An edge is an unordered pair $(v, w) = (w, v)$ of nodes. Definitions given for digraphs also hold for graphs.

Representation Methods

Both digraph and undirected graphs are usually represented using *adjacency matrices* and *adjacency lists*.
For a digraph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$, the adjacency matrix is defined as:

$$A[i][j] = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{if } (i, j) \notin E \end{cases}$$

A *labelled* adjacency matrix, A, also called a *cost* matrix is defined as

$$A[i][j] = \begin{cases} \text{label of arc } (i, j) & \text{if } (i, j) \in E \\ \text{an arbitrary symbol} & \text{if } (i, j) \notin E \end{cases}$$

The adjacency matrix is symmetric for undirected graphs.

Which of the two representations is best?

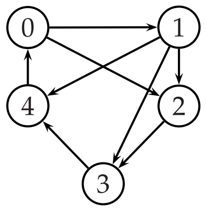
The adjacency *matrix* is useful when the presence or absence of an arc is frequently tested. It performs badly when the number of arcs if much less than the square of the number of nodes (i.e. $|E| << |V| \times |v|$).

The adjacency *list* makes better use of memory, but looking for arcs is more difficult. In an adjacency list the list of arcs for adjacent nodes is kept. The whole graph may be represented by a table indexed by nodes, each entry for a node i in the table containing the address of the list of nodes adjacent to i. That list may be a static or a dynamic list. Figure 7.1 shows a graph and its adjacency matrix representation. Figure 7.2 shows static/dynamic adjacency lists representations.

Traversals

Breadth-first Search

Breadth first search involves the following actions:



(a) A digraph.

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	1	1
2	0	0	0	1	0
3	0	0	0	0	1
4	1	0	0	0	0

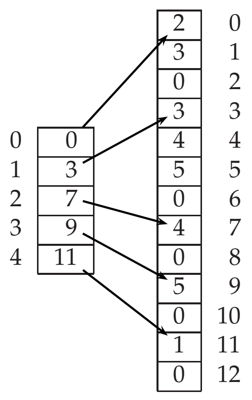
(b) Adjacency matrix representation for the digraph of Figure 7.1(a)

Figure 7.1.: A digraph and its adjacency matrix representation

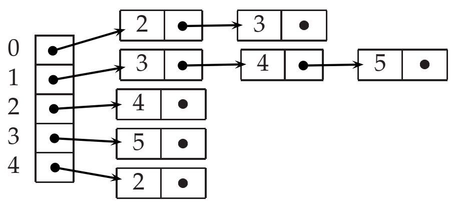
1. Enqueue the start vertex in an empty queue.
2. Dequeue one node to process and enqueue all its unvisited adjacent nodes.
3. Repeat step 2 until the queue becomes empty.

A sketch of the algorithm is:

```
enum { UNVISITED, VISITED };
void bfs( int nbOfNodes, int srcNode )
{
  int mark[ nbOfNodes ]; /* for marking visited nodes */
  QueueT Q; /* queue of nodes - integers */
  int i, v, w; /* nodes */

  makeNull( Q );
  for ( i = 0; i < nbOfNodes; i++ ) /* mark all nodes unvisited */
    mark[ i ] = UNVISITED;
  mark[ srcNode ] = VISITED; /* mark source node visited */
  process info for srcNode;
  enqueue( srcNode, Q );
  /* srcNode will be the first node dequeued in the loop below */
  while( ! empty( Q ))
  {
    v = dequeue( Q );
    for ( each node w adjacent to v )
      if ( mark[ w ] == UNVISITED )
      {
        mark[ w ] = VISITED;
        process info for w;
        enqueue( w, Q );
      }
  }
```



(a) Static adjacency list

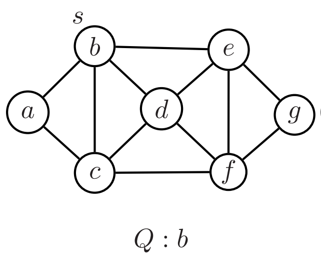


(b) Dynamic adjacency list.

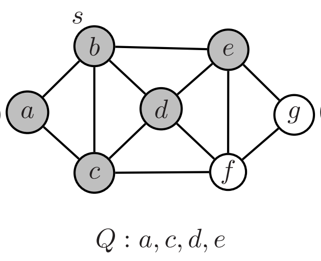
Figure 7.2.: Adjacency list representations for the graph of figure 7.1(a)

```
}
```

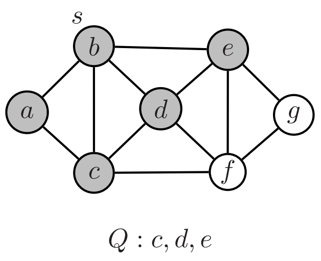
A trace of the relevant steps in breadth-first search is shown in Figure 7.3.



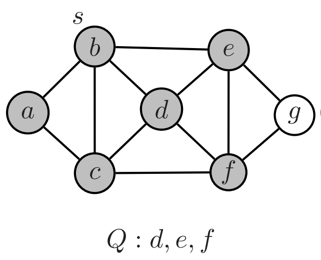
(a) Before entering the while-loop.



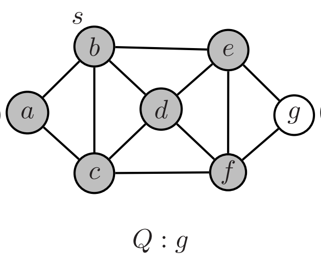
(b) After processing nodes adjacent to b.



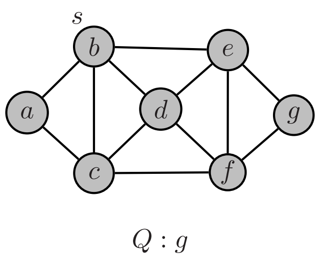
(c) After processing nodes adjacent to a.



(d) After processing nodes adjacent to c.



(e) After processing nodes adjacent to d and e.



(f) After processing nodes adjacent to f.

Figure 7.3.: A trace of breadth-first search on a graph.

Depth-first Search

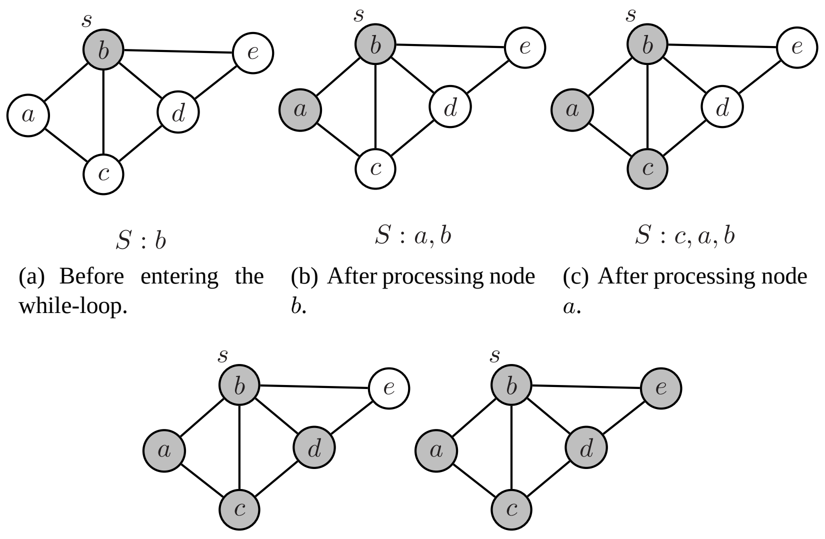
When searching a graph in *depth-first* search mode, the source node is marked *visited* first. Then every adjacent node is visited, recursively. After visiting all the nodes reachable form a given source, the traversal ends. If there still are unvisited nodes, another node is chosen as a source an the steps are repeated. The algorithm sketch is:

```
enum { UNVISITED, VISITED };
void dfs( int nbOfNodes, int srcNode )
{
  int mark[ nbOfNodes ]; /* for marking visited nodes */
  StackT S; /* stack of nodes */
  int i, v, w; /* nodes */

  for ( i = 0; i < nbOfNodes; i++ ) /* mark source node visited */
    mark[ i ] = UNVISITED;
  mark[ srcNode ] = VISITED; /* mark source node visited */
  process info for srcNode;
  push( srcNode, S );
  while ( !empty( S ) )
  {
    v = top( S );
    let w be the next unvisited node on AdjList( v );
    if ( w exists )
    {
      mark[ w ] = VISITED;
      process info for w;
      push( w, S );
    }
    else pop( S );
}
```

Note that this algorithm uses a stack to eliminate recursion. A trace of the relevant steps in breadth-first search is shown in Figure 7.4.

7.3. Lab.07 Assignments

Use C or C++ to solve the following:



$S : b$

(a) Before entering the while-loop.

$S : a, b$

(b) After processing node b.

$S : c, a, b$

(c) After processing node a.

$S : d, c, a, b$

(d) After processing node c.

$S : e, d, c, a, b$

(e) After processing node d.

Figure 7.4.: A trace of depth-first search on a graph.

7.7.1. Let G be a digraph, i.e. $G = (V, E)$ and let V' be one of its subgraphs. If the nodes are represented by integers which are to be read from a text file, output the induced subgraph $G' = (V', E')$ to a text file. Implement the necessary code. Input is given as follows:.7

```
V␣nodes␣1␣3␣5␣6␣7
V␣arcs␣(1␣3)␣(1␣5)␣(1␣7)␣(3␣6)␣(3␣7)␣...
V'␣nodes␣1␣3␣5
```

Output should look like this:

```
V'␣nodes␣1␣3␣5
V'␣arcs␣(1␣3)␣(1␣5)␣...
```

Hint. Use *fgets* and *sscanf* to read input. E.g. reading a pair of arcs can be accomplished with
`sscanf(&buffer[i], "(%d%d)", &v, &w)`
where v and w are integers.

7.7.2. Implement breadth-first and depth-first traversals for a graph represented by an adjacency matrix as presented in Lectures 5 and 6. Input is as for Problem 7.1. Output the nodes traversed in sequence to a text file of your choice.

7.7.3. For an undirected graph $G = (V, E)$, where node names are positive integers, implement a graph construction method, and a function named `havePath(v, w)`, where $v \in V$ and $w \in V$ are nodes, to check if there is a path between the two given nodes, say v and w of graph $G = (V, E)$. Graphs will be read from a file as before, and path will be printed to a text file as a sequence of nodes (i.e integer numbers). The invocation of function `havePath(v, w)` will be specified in the input file after graph specification input as a question mark and two node names (i.e. numbers), e.g.

```
V␣nodes␣1␣3␣5␣6␣7
V␣arcs␣(1␣3)␣(1␣5)␣(1␣7)␣(3␣6)␣(3␣7)␣...
?1,5
?1,6
?5,1
...
```

7.7.4. For an undirected graph $G = (V, E)$, where node names are positive integers, implement a graph construction method, and a function which returns the longest simple path, named `longestPath(a, b)`, where $a \in V$ and $b \in V$ are nodes. Input as before, output is a space separated list of nodes along the path.

7.7.5. For an undirected graph $G = (V, E)$, where node names are positive integers, implement a graph construction method, and a function named `isConnected` to check whether or not the graph is strongly connected. Input as before, output $\in \{"yes", "no"\}$.

8. Graph Processing Algorithms

8.1. Purpose

In this lab session we will implement some of the graph processing algorithms, i.e. Dijkstra's and Floyd's for minimum cost paths, and Kruskal's and Prim's for minimum spanning trees.

8.2. Brief Theory Reminder

Finding Shortest (Minimum Cost) Paths from a Single Vertex

Let $G = (V, E)$ be a labelled directed graph, where every arc is labelled with a non-negative number called a *cost*. The graph may be represented using the labelled adjacency matrix (also called a cost matrix).
Given two vertices, one denoted as the source and the other as the destination, one problem is to find the minimum cost path from the source to the destination. Dijkstra's solution to this problem is characterized by the following:

- A set S of vertices $j \in V$, for which there is at least one path from the source vertex, say s, to the destination, j, is maintained. Initially $S = \{s\}$.
- At each step, a vertex v whose distance to a vertex $w \in S$ is minimal, is added to S.

In order to record the minimal paths from the source s to each other vertex, we will use an array $parent$, in which $parent[k]$ holds the vertex preceding k on the shortest path.
We use the following notations in describing Dijkstra's algorithm:

- n is the number of vertices in V, i.e. $n = |V|$.
- The set S is represented by its characteristic vector, i.e. $S[i] = 1$ if $i \in S$, and $S[i] = 0$ if $i \notin S$.
- An $n \times n$ matrix, $cost$ defined as follows:

$$cost : \begin{cases} cost[i,j] = c,\ c > 0 & \text{if } (i,j) \in E \\ cost[i,j] = 0 & \text{if } i = j \\ cost[i,j] = \infty & \text{if } (i,j) \notin E \end{cases}$$

- The vector $parent$ holds the vertices which are accessible from the source vertices. It allows for path reconstruction – from a source to any accessible node.
- For nodes inaccessible from a source node $S[i] = 0$ and $dist[i] = \infty$.

Then, Dijkstra's algorithm is:

```
#define NMAX ? /* max no. of nodes */
#define INFTY ? /* big value for infinity */
double dist[ NMAX ]; /* distances */
double cost[ NMAX ][ NMAX ];
int parent[ NMAX ];
int S[ NMAX ];

void Dijkstra( int nbOfNodes, int source )
/* nbOfNodes = number of nodes in the graph
   source = source node id */
{
  int i, j, k, step;

  /* initialize */
  for ( i = 1; i <= n; i++ )
  {
    S[ i ] = 0; /* set S empty */
    dist[ i ] = cost[ source ][ i ];
    if ( dist[i] < INFTY )  parent[ i ] = source;
    else                    parent[ i ] = 0;
  }
  /* add source to set S */
  S[ source ] = 1;
  parent[ source ] = 0;
  dist[ source ] = 0;
  /* build vectors dist and parent */
  for ( step = 1; step <= n-1; step++ )
  {
    find unselected vertex k with dist[k] minimal;
    if ( minimum found == INFTY ) return;
    S[ k ] = 1; /* add k to set S */
    for ( j = 1; j <= n; j++)
      if ( S[ j ] == 0 && dist[ k ] + cost[ k ][ j ] < dist[ j ] )
      {
        dist[ j ] = dist[ k ] + cost[ k ][ j ];
        parent[ j ] = k;
      }
  }
}
```

All Pairs Shortest Paths

By repeatedly applying Dijkstra's algorithm with each node as a source, in turn, we can obtain the minimum paths for all the pairs of vertices in a graph. Another choice is R. W. Floyd's algorithm. Floyd's algorithm was developed for solving the all pairs shortest paths problem.
The algorithm maintains the minimal costs in an array, say A. Initially, A is identical to the cost matrix, $cost$. The minimal distances computation is accomplished in n iterations, where n is the number of nodes. At iteration k, $A[i][j]$ holds the minimum distance between nodes i and j using paths which do not contain nodes numbered higher than k, except possibly for the end nodes, i and j. A is calculated with the following formula:

$$A_{ij}{}^{(k)} = \min(A_{ij}{}^{(k-1)}, A_{ik}{}^{(k-1)} + A_{kj}{}^{(k-1)})$$

Because $A_{ik}{}^{(k)} = A_{ik}{}^{(k-1)}$ and $A_{kj}{}^{(k)} = A_{kj}{}^{(k-1)}$ we may use a single copy of array A. Floyd's is then:

```
#define NMAX ? /* max no. of nodes */
double cost[ NMAX ][ NMAX ];
double A[ NMAX ][ NMAX ];

void Floyd( int nbOfNodes )
{
  int i, j, k;

  /* initialize A */
  for ( i = 1; i <= n; i++ )
    for ( j = 1; j <= n; j++ )
      A[ i ][ j ] = cost[ i ][ j ];
  for ( i = 1; i <= n; i++ )
    A[ i ][ i ] = 0;
  for ( k = 1; k <= n; k++ )  /* all nodes */
    for ( i = 1; i <= n; i++ ) /* all lines */
      for ( j = 1; j <= n; j++ ) /* all columns */
        if ( A[ i ][ k ] + A[ k ][ j ] < A[ i ][ j ] )
          A[ i ][ j ] = A[ i ][ k ] + A[ k ][ j ];
}
```

For keeping track of the shortest paths, we can make use of an additional table, p, where $p[i, j]$ will hold the vertex k which lead to the minimum distance $A[i, j]$. If $p[i, j] = 0$, then the edge (i, j) is the shortest from i to j. In order to be able to display the intermediate vertices along the path from i to j, we can use the following scheme:

```
void path( int i, int j )
{
  int k;
  k = p[ i ][ j ];
  if ( k != 0 )
  {
    path( i, k );
    list node k;
    path( k, j );
  }
}
```

8.3. Minimum Spanning Tree

The minimum spanning tree problem is stated as follows:

Let $G = (V, E)$ be an undirected connected graph. Each edge $(i, j) \in E$ has a non-negative cost attached to it. We have to find a partially connected graph, say $A = (V, T)$, such as the sum of the costs of the edges in T to be a minimum. Note that this partial graph is the *spanning tree*.

One solution is Prim's algorithm:

- Start with a subset $W = \{s\}$, where s is a start node , and $T = \emptyset$.
- At each step, select the edge (w, u) of minimum cost, with $winW$ and $u \in V \setminus W$. Add u to W, i.e. $W = W \cup \{u\}$, and (w, u) to T, i.e. $T = T \cup \{(w, u)\}$
- Finally W will contain all the nodes in V, and T will be the minimum cost spanning tree.

```
void Prim( int n )
{
  W = { 1 }; /* start from node 1 */
  T = {};    /* T initially empty */
  while ( W != V )
  {
    select edge of minimum cost (w, u), with  $w \in W$ and $u \in V \setminus W$;
    add u to W;
    add (u, w) to T;
  }
}
```

Another solution to this problem was given by Kruskal. Kruskal maintains the edges in the order of ascending costs. The spanning tree will have $n - 1$ edges. At each step, a minimum cost edge is selected, such a way that it does not form a cycle together with the rest of the edges contained in T. The sketch for Kruskal's is:

```
void Kruskal( int n )
{
  T = {};
  while ( T is not a spanning tree )
  {
    select edge of min. cost (w, u) from V;
    delete edge (w, u) from V;
    if ( (w, u)  does not close a cycle in T ) add (w, u) to T;
  }
}
```

The difficult part in Kruskal's is checking for a cycle. In order to implement Kruskal's algorithm, a data structure which supports union (or merge) and find operations on disjoint sets is useful. Three operations are necessary:

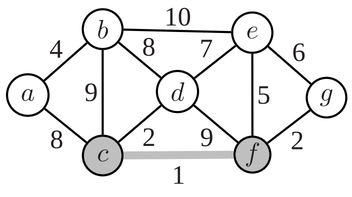
CreateSet(u) $-$ create a set containing a single item u.
FindSet(u) $-$ find the set that contains a given item u.
Union(u, v) $-$ merge the set containing u and the set containing v into a set containing both items.

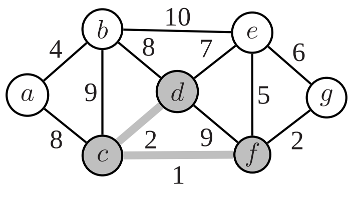
Thus the sketch for Kruskal's algorithm becomes:

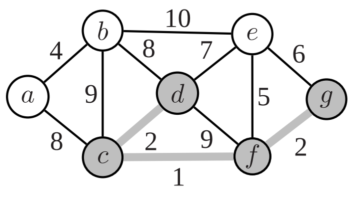
```
SetT Kruskal( GraphT G, VertexT v )
/* G=(V, E), where V is the set of vertices,
             and E is the set of edges for
             graph G.
*/
{
  SetT A;

  makeEmpty(A);   /* A is initially empty */
  for ( each u in V ) CreateSet( u ); /* create set for each vertex */
  Sort E in increasing order by weight w;
    if ( FindSet( u ) ) != FindSet( v ) )
    { /* if u and v are in different trees */
      Add( u, v ) to A;
```

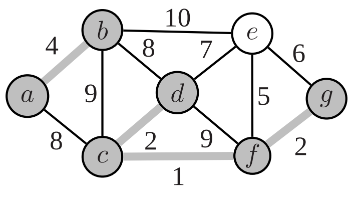
```
        Union( u, v );
      }
   return A;
}
```

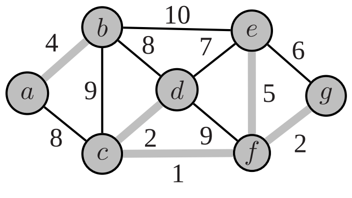
Figure 8.1 shows a trace of the previous algorithm on a graph.


(a) Step 1. Adding edge $c - f$ of weight 1.


(b) Step 2. Adding edge $c - d$ of weight 2.


(c) Step 3. Adding edge $f - g$ of weight 2.


(d) Step 4. Adding edge $a - b$ of weight 4.


(e) Step 5. Adding edge $e - f$ of weight 5.

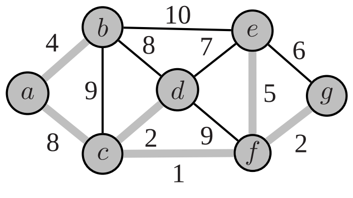

(f) Step 6. Adding edge $a - c$ of weight 8.

Figure 8.1.: A trace of Kruskal's algorithm.

8.4. Lab.08 Assignments

8.8.1. Implement an algorithm to find the *longest* cost simple path in a *directed* graph. .
I/O description. Input consists of pairs of nodes given as numbers, separated by a comma and followed by an equal sign followed by the cost attached to that edge $n_1, \ n_2 = n_3$ which are connected by an edge. .8**E.g.**

```
1,2=22
1,3=11
1,5=5
1,7=5
2,3=55
2,4=41
2,7=33
3,4=8
3,5=4
3,6=9
4,6=17
4,7=9
5,7=20
```

The output is the longest simple path, which should be given similar to this (Note that it is NOT the output, is given just for aspect reference):

```
99␣1-(5)-7-(20)-5-(4)-3-(9)-6-(17)-4-(41)-2
```

where the first number is the maximum cost, followed by tail vertex, a hyphen, cost of arc enclosed in parenthesis, a hyphen, and head vertex, a.s.o. What is the time performance of your algorithm?

8.8.2. Implement an algorithm to find the *longest* cost simple path in an *undirected* graph. Input and output as for Problem 8.1. What is the time performance of your algorithm?

8.8.3. Implement an algorithm to find a *maximum* cost spanning tree of a *directed* graph. Use the same format as in the previous problem for input. The output should be the preorder and inorder traversal of the tree. What can you tell about the running time of this algorithm?

8.8.4. Implement an algorithm to find a *maximum* cost spanning tree of an *undirected* graph. Use the same format as in the previous problem for input. The output should be the postorder and inorder traversal of the tree. What can you tell about the running time of this algorithm?

8.8.5. Implement an algorithm to find the articulation points in an undirected graph. Same format for input. Output is the list of names of vertices which are articulation points, separated by commas, or the text `Graph is strongly connected` if there are no articulation points.

9. Algorithm Design. Greedy and Backtrack

9.1. Purpose

In this lab session we will experiment with greedy and backtracking algorithms.

9.2. Brief Theory Reminder

Greedy Algorithm Development

This method applies to problems where out of a set, say A, a subset B must be selected, such a way that it satisfies certain requirements. Once an element was selected, it is included in the final solution. Once excluded, it will never be examined again. This is the reason for which it has been called "‘greedy"’. The method finds a single solution.

Two variations of the method are generally used:

9.2.1. Start with an empty B set. Select an unselected element from set A. See if adding it to set B leads to a solution. If so, add it to B. This could be sketched as follows:

```
#define MAXN ? /* suitable value */

void greedy1( int A[ MAXN ], int n,
              int B[ MAXN ], int *k )
/* A = set of candidate n elements
   B = set of k elements solution */
{
  int x, v, i;
  *k = 0; /* empty solution set */
  for ( i = 0; i < n; i++ )
  {
    select( A, B, i, x );
    /* select x, the first of A[ i ], A[ i + 1 ], ..., A[ n − 1 ],
       and swap it with element at position i */
    v = checkIfSolution( B, x );
    /* v = 1 if by adding x we get a solution and v = 0 otherwise */
    if ( v == 1 )
      add2Solution( B, x, *k );
    /* add x to B, specifying the number of elements in B */
  }
}
```

Here, it is the job of $select$ to set the criterion used in obtaining the final solution.

9.2.2. First set the order used to consider the elements of set A. Then take elements of A one by one using the established order and check if, by adding it to the partial solution stored in set B we can get a possible solution. If so, add it to B. A sketch of this is:

```
#define MAXN ? /* suitable value */

void greedy2( int A[ MAXN ], int n,
              int B[ MAXN ], int *k )
/* A is the set of candidate n elements; B is the set of k elements solution */
{
  int x, v, i;
  *k = 0; /* empty solution set */

  process( A, n );  /* rearrange A */
  for ( i = 0; i < n; i++ )
  {
    x = A[ i ];
    checkIfSolution( B, x, v );
    /* v = 1 if by adding x we get a solution and v = 0 otherwise */
```

```
          if ( v == 1 )
            add2Solution( B, x, *k );
            /* add x to B, specifying the number of elements in B */
        }
      }
```

Greedy Example. Prim's Algorithm

This problem was stated in §8.3. We will briefly remind how this algorithm works:

9.2.1. Initially, include only one node in the tree T. (It does not matter which, so we can safely start with node 1.) The set of arcs in the tree is empty.

9.2.2. Select a minimum cost arc, having one end in the tree and the other in the rest of vertices. Repeat this step $n-1$ times. To avoid passing through all arcs at each step, we can use an n-component vector v defined as:

$$U_i = \begin{cases} 0 & \text{if vertex } i \in T \\ k & \text{if vertex } i \notin T; \\ & k \in T \text{ is a node such that} \\ & (i,k) \text{ is a minimum cost edge} \end{cases}$$

Initially, $v[1] = 0$, and $v[2] = v[3] = \ldots = v[n] = 1$, i.e initially the tree is $A = (\{1\}, \emptyset)$.

Here is an implementation:

```
#include <stdio.h>
#define MAXN 10
#define INFTY 0x7fff

void Prim2( int n, int c[ MAXN ][ MAXN ], int e[ MAXN ][ 2 ], int *cost )
/* n = number of vertices;
   c = cost matrix;
   e = edges of the MST;
   c = cost of MST */
{
  int v[ MAXN ];
  /* v[ i ] = 0 if i is in the MST;
     v[ i ] = j if i is not in the MST;
     j is a node of the tree such that (i, j) is a minimum cost edge */
  int i, j, k, min;

  *cost = 0;
  v[ 1 ]=0;
  for ( i = 2; i <= n; i++ )
    v[ i ] = 1; /* tree is ({1},{}) */
  /* find the rest of edges */
  for ( i = 1; i <= n - 1; i++ )
  { /* find an edge to add to the tree */
    min = INFTY;
    for ( k  = 1; k <= n; k++ )
      if ( v[ k ] != 0 )
        if ( c[ k ][ v[ k] ] < min )
        {
          j = k;
          min = c[ k ][ v[ k ]];
        }
    e[ i ][ 0 ] = v[ j ];
    e[ i ][ 1 ] = j;
    *cost += c[ j ][ v[ j ]];
    /* update vector v */
    v[ j ]= 0;
    for ( k = 1; k <= n; k++ )
      if ( v[ k ] != 0 &&
           c[ k ][ v[ k ]] > c[ k ][ j ])
        v[ k ] = j;
  }
}
int main( int argc, char *argv[] )
{
  int n; /* number of nodes */
  int c[ MAXN ][ MAXN ]; /* costs */
  int e[ MAXN ][ 2 ]; /* tree edges */
  int i, j, k, cost;

  printf( "\nNumber␣of␣nodes␣in␣graph␣G:␣" );
  scanf( "%d", &n );
  while ( '\n' != getchar() );
  for ( i = 1; i <= n; i++ )
    for ( j = 1; j <= n; j++ )
      c[ i ][ j ] = INFTY;
  /* read cost matrix (integers) */
  for ( i = 1; i < n; i++ )
  {
    do
    {
      printf(
      "\nNode␣adjacent␣to␣%2d␣[0=finish]:",
              i );
      scanf( "%d", &j );
      while ( '\n' != getchar() );
      if (j > 0 )
      {
        printf("\nCost␣c[%d][%d]:", i, j);
        scanf( "%d", &c[ i ][ j ] );
        while ( '\n' != getchar() );
        c[ j ][ i ] = c[ i ][ j ];
        /* c is symmetric */
      }
    }
    while ( j > 0 );
  }
  Prim2( n, c, e, &cost );
  printf("\nThe␣cost␣of␣MST␣is␣%d", cost );
  printf("\nTree␣edges\tEdge␣cost\n" );
  for ( i = 1; i <= n - 1; i++ )
    printf("%2.2d␣-␣%2.2d\t%10d\n",
      e[ i ][ 0 ], e[ i ][ 1 ],
      c[ e[ i ][ 0 ] ][ e[ i ][ 1 ] ] );
  return 0;
}
```

Backtracking

Backtracking is used in developing algorithms for the following type of problems where n sets, say $S_1, S_2, \ldots, S_n$ are given, each of n_i components. We are required to find the elements of a vector $X = (x_1, x_2, \ldots, x_n) \in S = S_1 \times S_2 \times \ldots \times S_n$ such that a relation $\varphi(x_1, x_2, \ldots, x_n)$ holds for the elements of vector X. The relation φ is called an *internal* relation, the set $S = S_1 \times S_2 \times \ldots \times S_n$ is called the *space of possible solutions*, and the vector $X = (x_1, x_2, \ldots, x_n)$ is called a *result*.
Backtracking finds all the possible results of the problem. Out of all these, we can pick up one that satisfies an additional condition.
Backtracking eliminates the need to generate all the $\prod_{i=1}^{n} n_S$ possible solutions. To achieve that, in generating vector X we have to obey the following conditions:

9.2.1. x_k gets values only if $x_1, x_2, \ldots, x_{k-1}$ have already got values.
9.2.2. After setting x_k to a value, check the *continuation* relation $\varphi(x_1, x_2, \ldots, x_k)$ establishing if it makes sense to evaluate x_{k+1}. If condition $\varphi(x_1, x_2, \ldots, x_k)$ is not met, then we pick up a new value for $x_k \in S_k$ and check φ again. If the set of choices for x_k becomes empty, we restart by selecting another value for x_{k-1} and so forth. These reduction in the value of k is the origin of the name of the method. It suggests that when no advance is possible, we back-track the sequence of the current solution.

> There is a strong relationship between the internal and the continuation condition. Optimal setting of the continuation condition greatly reduces the number of computations.

A non-recursive backtracking algorithm may be sketched as follows:

```
#define MAXN ? /* suitable value */

void nonRecBackTrack( int n )
/* sets $S_i$ and the corresponding number of elements in each set,
  $n_S$, are assumed global */
{
  int x[ MAXN ];
  int k, v;

  k=1;
  while ( k > 0 )
  {
    v = 0;
    while ( $\exists$ untested $\alpha \in S_k$ && v == 0 )
    {
      x[ k ] = $\alpha$;
      if ( $\varphi$( x[ 1 ], x[ 2 ],..., x[ k ] )
        v = 1;
    }
    if ( v == 0 )
      k--;
    else
    if ( k == n )
      listOrProcess( x, n )
      /* list or process solution */
    else
      k++;
  }
}
```

```
#define MAXN ? /* suitable value */
int x[ MAXN ];

/* $n$, the number of sets $S_k$, sets $S_k$ and the corresponding number of elements
   in each set, $nS[k]$, are assumed global */
void recBackTrack( int k )
{
  int j;

  for ( j = 1; j <= nS[ k ]; j++ )
  {
    x[ k ] = $S_k$[ j ];
    /* the $j^{\text{th}}$ element of set $S_k$ */
    if ( $\varphi$( x[1], x[2],..., x[k] )
      if ( k < n )
        recBackTrack( k + 1 );
      else
        listOrProcess( x, n ) /* list or process solution */
  }
}
```

To process the whole space invoke $recBackTrack(1)$.

Backtracking Example. The Queen Placement Problem

The problem is stated as follows:

> Find all the arrangements of n queens on an $n \times n$ chessboard such that no queen would threaten another, i.e. no two queens are on the same line or diagonal.

As on each line there should be only one queen, the solution may be presented as a vector $X = (x_1, x_2, \ldots, x_n)$, where x_i is the column where a queen is placed on line i.
The continuation conditions are:

9.2.1. No two queens may be on the same column, i.e. $X[i] \neq X[j] \forall i \neq j$

9.2.2. No two queens may be on the same diagonal, i.e. $|k - i| \neq |X[k] - X[i]|$ for $i = 1, 2, \ldots, k - 1$.

A non-recursive solution for this problem is:

```
#include <stdio.h>
#include <stdlib.h>
#define MAXN 10

void nonRecQueens( int n )
/* find all possible arrangements of n queens on a chessboard such that no queen
   threatens another */
{
  int x[ MAXN ];
  int v;
  int i, j, k, solNb;

  solNb = 0;
  k = 1;
  x[ k ] = 0;
  while( k > 0)
  { /* find a valid arrangement on line k */
    v=0;
    while( v==0 && x[ k ] <= n - 1 )
    {
      x[ k ]++;
      v = 1;
      i = 1;
      while ( i <= k - 1 &&
              v == 1 )
        if ( x[ k ] == x[ i ] ||
             abs( k - i ) ==
             abs( x[ k ] - x[ i ] ))
          v=0;
        else
          i++;
    }
    if ( v == 0 )
      k = k - 1;
    else
    {
      if ( k== n )
      { /* display chessboard */
        solNb++;
        printf( "\nSolution %d\n", solNb );
        for ( i = 1; i <= n; i++ )
        {
          for ( j = 1; j <= n; j++ )
            if ( x[ i ] == j )
              printf( "1" );
            else
              printf( "0" );
          printf( "\n" );
        }
        while ( '\n' != getchar() );
      }
      else
      {
        k++;
        x[ k ]=0;
      }
    }
  }
}
int main(void)
```

```
{
  int n;

  printf( "\nNumber␣of␣queens=" );
  scanf( "%d", &n );
  while ( '\n' != getchar() );
  nonRecQueens( n );
  printf( "\nEND\n" );
  return 0;
}
```

A recursive solution for this problem is:

```
#include <stdio.h>
#include <stdlib.h>
#define MAXN 10
int x[ MAXN ];
int n; /* chessboard size */
int solNb; /* solution number */
enum { FALSE=0, TRUE=1 };

int phi( int k )
/* test continuation conditions */
{
  int p;

  for ( p = 1; p <= k - 1; p++ )
    if ( x[ k ] == x[ p ] ||
         abs( k - p ) ==
         abs( x[ k ] - x[ p ] ))
      return FALSE;
  return TRUE;
}
void recQueens( int k )
/* find all possible arrangements of n queens on a chessboard such that no queen
   threatens another */
{
  int i, j, p;

  for ( j = 1; j <= n; j++ )
  {
    x[ k ] = j;
    if ( phi( k ) == TRUE )
      if ( k < n )
        recQueens( k + 1 );
      else
      { /* list solution */
        solNb++;
        printf( "\nSolution␣%d\n", solNb );
        for ( i = 1; i <= n; i++ )
        {
          for ( p = 1; p <= n; p++ )
            if ( x[ i ] == p )
              printf( "1" );
            else
              printf( "0" );
          printf( "\n" );
        }
        while ( '\n' != getchar() );
      }
  }
}
int main(void)
{
  printf( "\nNumber␣of␣queens=" );
  scanf( "%d", &n );
```

```
    while ( '\n' != getchar() );
    solNb = 0;
    recQueens( 1 );
    printf( "\nEND\n" );
    return 0;
}
```

9.3. Lab.09 Assignments

Solve the following problems, with data read from a file and output to a file, in C/C++, using the backtracking approach:

9.9.1. Map coloring. An map of an area of the world contains n countries. Each country is a neighbor of some other country. You are given m colors to color this map. Find all the possible country colorings using all the m colors, such a way that any two neighboring countries are colored differently. .
I/O description. Input: number of countries on one line, followed by the neighboring countries given as name pairs – each on one line, then the number of colors on line line, then color names as strings, e.g.

```
9␣␣␣#␣number␣of␣countries
Romania␣Hungary
Romania␣Serbia
Romania␣Bulgaria
...
5␣␣␣#␣number␣of␣colors
red
green
yellow
...
```

Note that whatever follows the # on an input line is a remark, and thus it is ignored. Output is vertex – color pairs, one on a line, e.g:

```
Romania␣yellow
Hungary␣green
Serbia␣red
Ukraine␣white
...
```

9.9.2. Hamiltonian cycle. A connected graph $G = (V, E)$ is given by its cost matrix, where all costs are positive, and ≤ 65534. Determine a simple cycle passing through all nodes (a Hamiltonian cycle) of minimal cost. .
I/O description. Input: number of nodes on one line, followed by the cost matrix row by row, e.g. (for graph shape see Figure 9.1).

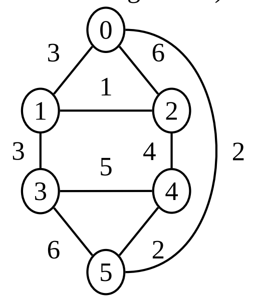


Figure 9.1.: Example Graph for Problem 9.2

```
6
␣␣␣␣␣␣0␣␣␣␣␣1␣␣␣␣␣2␣␣␣␣␣3␣␣␣␣␣4␣␣␣␣␣5
0␣␣␣␣␣0␣␣␣␣␣3␣␣␣␣␣6␣65535␣65535␣␣␣␣␣2
1␣␣␣␣␣3␣␣␣␣␣0␣␣␣␣␣1␣␣␣␣␣3␣65535␣65535
2␣␣␣␣␣6␣␣␣␣␣1␣␣␣␣␣0␣65535␣␣␣␣␣4␣65535
3␣65535␣␣␣␣␣3␣65535␣␣␣␣␣0␣␣␣␣␣5␣␣␣␣␣6
4␣65535␣65535␣␣␣␣␣4␣␣␣␣␣5␣␣␣␣␣0␣␣␣␣␣2
5␣␣␣␣␣2␣65535␣65535␣␣␣␣␣6␣␣␣␣␣2␣␣␣␣␣0
```

Here the value 65535 means no arc ($+\infty$). Nodes are numbered from zero.
Output: sequence of nodes separated by spaces on one line, e.g.[a]

```
0␣2␣1␣3␣4␣5␣0
```

[a]Here, output is just an example of a Hamiltonian cycle, but not a Hamiltonian cycle of minimum length

9.9.3. Find the smallest sum of n integer numbers taken from different diagonals parallel with the main diagonal, and including the main diagonal of a matrix A of size $n \times n$. .
I/O description. Input: number of lines and columns in matrix A, followed by the rows of the matrix, e.g.

```
3
1␣␣␣␣7␣␣␣2
```

```
4␣␣␣␣5␣␣␣3
10␣␣-2␣␣␣0
```

Output:

```
0
```

The elements taken were: -2, 0, and 2.

9.9.4. Find the smallest difference of n integer numbers taken from different diagonals parallel with the secondary diagonal, and including the secondary diagonal of a matrix A of size $n \times n$. .
I/O description. Input: number of lines and columns in matrix A, followed by the rows of the matrix, e.g.

```
3
1␣␣␣␣7␣␣␣2
4␣␣␣␣5␣␣␣3
10␣␣-2␣␣␣0
```

Output:

```
-19
```

The elements taken were: -2, 7, and 10 (i.e. $-2 - 7 - 10$).

9.9.5. A maze is coded using a $n \times m$ matrix with corridors represented by 1s situated in consecutive positions on the same line or column, and the rest of the elements are 0. A person is in position (i, j) inside the labyrinth. List all the exit routes which do not pass the same place twice. .
I/O description. Input: n and m on one line, followed by the rows of matrix A, the coordinates (row, column) of the exit, and the coordinates of the person (row, column), e.g.

```
25␣30
000000000000000000000000000000
001111110111111111011111111100
001000010100000000000001000100
...
```

Output is a sequence of row$-$column pairs indicating the successive position of the person.

9.9.6. A set of integer numbers is given. Generate all the subsets of this set which sum up to exactly S. .
I/O description. Input: enumeration of elements in the set, on one line, then sum on one line e.g.

```
1␣-3␣5␣-7␣2␣6
6
```

Output: enumeration of elements summing up to the given sum, e.g.

```
1␣-3␣2␣6
5␣-7␣2␣6
...
```

9.9.7. A set of natural numbers is given. Generate all the subsets of this set joined by alternating + and - operators which sum up to exactly S. .
I/O description. Input: enumeration of elements in the set, on one line, then sum on one line e.g.

```
1␣3␣5␣7␣2␣6
0
```

Output: enumeration of elements resulting in the given sum, e.g.

```
1-3+2
1+3-6
5-7+2
1+5-6
...
```

9.9.8. A ball is placed on a sand dune of varying height, situated in a plane area. The dune height (natural number ≤ 255 is coded using a $n \times m$ matrix with discrete heights represented by natural numbers. The height of the plane area is 1 less than the lowest dune point. The initial position of the ball is given as a row$-$column pair i, j in the matrix.

List all the possibilities for the ball to climb down the dune out to the plane area, and which do not pass the same place twice. Only if it is moving, the ball will pass through equal height points, otherwise not. The ball can move only on columns or rows. .

I/O description. Input: n and m on one line, followed by the rows of matrix A, and the coordinates of the ball (row, column), e.g.

```
5␣4␣␣#␣dune␣size
15␣15␣11␣22
15␣10␣11␣15
10␣␣2␣16␣16
␣7␣␣8␣15␣33
11␣11␣11␣11
3,3␣␣#␣ball␣position
```

Output is a sequence of row−column pairs indicating the successive position of the ball, e.g.

```
3,3␣2,3␣1,3
3,3␣4,3␣4,2␣4,1
```

10. Algorithm Design. Divide and Conquer and Branch and Bound

10.1. Purpose

In this lab session we will experiment with divide and conquer and branch and bound algorithm development methods.

10.2. Brief Theory Reminder

Branch and Bound

The branch and bound approach is related to backtracking. What makes them somewhat different is the order of state space traversal and also the way they prune the subtrees which cannot lead to a solution.

Branch and bound applies to problems where the initial state, say s_0, and the final state, say s_f are both known. In order to reach s_f from s_0, a series of decisions are made. We are interested in minimizing the number of intermediate states.
Let us assume that these states are nodes in a graph, and an edge indicates that a decision changed state s_i into state s_{i+1}. We have to impose a restriction on the nodes of the graph, that is "‘no two nodes may hold the same state"’, in order to prevent the graph to become infinite. Thus, the graph reduces to a tree. We then generate the tree up to the first occurrence of the node holding the final state.
The graph may also be traversed depth-first or breadth-first, but then the time needed to get a solution is longer. We can get a superior strategy by selecting the closest-to-final node out of the descendants of the current node. We use a cost function c, defined on the tree nodes, to evaluate the distance to the final state. Based on the values supplied by this function, we can select a minimum cost node from the set of descendants of the current node. Such a traversal is called a *least-cost*, or LC traversal.
An ideal function used to measure the distance from one node to the final one is:

$$c(x) = \begin{cases} level(x) - 1 & \text{if } x = \text{final node} \\ +\infty & \text{if } x = \text{terminal node and} \\ & \quad x \neq \text{final node} \\ \min c(y) & \text{if } x \neq \text{final node} \end{cases}$$

where y is a terminal node located in the subtree rooted at node x. But, the function c defined in this way is not applicable, because its computation needs traversing all the nodes in the tree $-$ and this is what we would like to avoid. Still, we have to notice that if we choose to compute c, then descending the tree to the final node involves traversing an already known path, passing through the vertices x which have $c(x) = c(root)$. Because using c is unpractical, we define an approximation for it, in one of the following two ways:

10.2.1. $\hat{c(x)}$=level of vertex x + distance($current\ state$, $final\ state$), or
10.2.2. $\hat{c(x)}$=cost($parent(x)$) + distance($current\ state$, $final\ state$),

where by "‘level of vertex x"’ we mean the number of decision made to reach the current configuration.
The form of the "‘distance"’ function is problem specific. E.g., for the game known as PERSPICO (see laboratory assignments), the distance is the number of tiles which are displaced.
Here are some remarks regarding the algorithm for branch and bound, given below:

- List L holds the nodes storing the state configuration.
- The parent of current vertex i is stored in vector $parent$, which allows rebuilding the path from the root node to the final node.
- $root$ holds a pointer to the start vertex $-$ i.e initial state.

```
NodeT *root;

root = ( NodeT * ) malloc( sizeof( NodeT ));
/* place initial configuration at location pointed to by root */
void BranchAndBound()
{
  NodeT *i, *j;
  ListT L;

  i = root;
  makeNull( L );
  for ( ; ; )
  {
    while ( $\exists\, j$ an unprocessed neighbor of $i$ )
    {
      if ( j == final node )
      {
        print path from $root$ to $j$;
        return;
      }
      else
      {
        append( j, L );
        parent[ j ] = i;
      }
      if ( isEmpty( L ) )
      {
        printf( "No␣solution\n" );
        return;
      }
      else i = LCelement( L );
    }
  }
}
```

The functions used above have the following meanings:

$makeNull(L)$ makes L the empty list;
$isEmpty(L)$ returns 1 if L is empty and 0 otherwise;
$LCelement(L)$ returns an element on list L having the lowest cost $\hat{c}$, to support a LC traversal of the state tree;
$append(j, L)$ appends node j to list L.

Note that when a node i from list L becomes the current node, then all its descendants are generated, and placed on list L. One of these will be selected as current, based on its cost $\hat{c}$, and the process goes on till the final node is reached.

Divide and Conquer

Divide and conquer means repeatedly dividing a problem into two or more subproblems of the same type, and then recombining the solved subproblems in order to get a solution of the problem.
Let $A = (a_1, a_2, \ldots, a_n)$ be a vector whose components are processed. Divide an conquer is applicable if for any p and q, natural numbers such that $1 \leq p < q \leq n$ we can find a number $m \in [p+1, q-1]$ such that processing the sequence $a_p, a_{p+1}, \ldots, a_q$ can be achieved by processing sequences $a_p, a_{p+1}, \ldots, a_m$ and $a_m, a_{m+1}, \ldots, a_q$, and then combining the results. Briefly, divide and conquer may be sketched as follows:

```
void DivideAndConquer( int p, int q, SolutionT $\alpha$ )
/* $p$ and $q$ are indices in the processed sequence; $\alpha$ is the solution */
{
  int $\epsilon$, m;
  SolutionT $\beta$, $\gamma$;

  if ( abs( q - p ) $\leq \epsilon$ ) process( p, q, $\alpha$ );
  else
  {
    Divide( p, q, m );
    DivideAndConquer( p, m, $\beta$ );
    DivideAndConquer( m + 1, q, $\gamma$ );
    Combine( $\beta$, $\gamma$, $\alpha$ );
  }
}
```

Invocation:

```
DivideAndConquer( 1, n, $\alpha$ )
```

Some more things have to be specified:

ϵ is the maximum length of a sequence $a_p, a_{p+1}, \ldots, a_q$ which can be directly processed;
m is the intermediate index where we can split the sequence $a_p, a_{p+1}, \ldots, a_q$;
$beta$ and γ are the intermediate results obtained by processing sequences $a_p, a_{p+1}, \ldots, a_m$ and $a_m, a_{m+1}, \ldots, a_q$, respectively;
α is the result of combining intermediate solutions β and γ;
$split$ splits the sequence $a_p, a_{p+1}, \ldots, a_q$ into sequences $a_p, a_{p+1}, \ldots, a_m$ and $a_m, a_{m+1}, \ldots, a_q$;
$combine$ combines solutions β and γ obtaining the final solution, α.

E.g. Mergesort a vector of n elements:

```
#include <stdio.h>
#define MAXN 100
int A[ MAXN ]; /* vector to sort */

void printVector(int n)
/* print vector elements - 10 on one line */
{
  int i;

  printf( "\n" );
  for( i = 0; i < n; i++ )
  {
     printf("%5d", A[ i ]);
     if( (i + 1) % 10 == 0 )
       printf("\n");
  }
  printf("\n");
}
void merge(int lBound, int mid, int rBound)
{
  int i, j, k, l;
  int B[ MAXN ]; /* B = auxiliary vector */

  i = lBound;
  j = mid + 1;
  k = lBound;
  while( i <= mid && j <= rBound )
  {
    if( A[ i ] <= A[ j ])
    {
      B[ k ] = A[ i ];
      i++;
    }
    else
    {
      B[ k ] = A[ j ];
      j++;
    }
    k++;
  }
  for ( l = i; l <= mid; l++ )
  { /* there are elements on the left */
    B[ k ] = A[ l ];
    k++;
  }
  for ( l = j; l<= rBound; l++ )
  { /* there are elements on the right */
    B[ k ] = A[ l ];
    k++;
  }
  /* sequence from index lBound to rBound is now sorted */
  for( l = lBound; l <= rBound; l++ )
    A[ l ] = B[ l ];
}
```

```
void mergeSort( int lBound, int rBound)
{
  int mid;

  if( lBound < rBound)
  {
    mid= ( lBound + rBound ) / 2;
    mergeSort( lBound, mid );
    mergeSort( mid + 1, rBound);
    merge( lBound, mid, rBound);
  }
}
int main()
{
  int i, n;

  printf("\nNumber␣of␣elements␣in␣vector=");
  scanf( "%d", &n );
  while ( '\n' != getchar() );
  printf("\nPlease␣input␣vector␣elements\n");
  for( i = 0; i < n; i++ )
  {
    printf( "a[%d]=", i );
    scanf( "%d", &A[ i ] );
  }
  printf("\nUnsorted␣vector\n");
  printVector( n );
  mergeSort( 0, n-1 );
  printf("\nSorted␣vector\n");
  printVector( n );
  while ( '\n' != getchar() );
  while ( '\n' != getchar() );
  return 0;
}
```

In the program above, solution combination is achieved by $merge$.

10.3. Lab.10 Assignments

Solve the following problems using branch and bound:

10.3.1. The 15 puzzle. There are 15 tiles, numbered from 1 to 15, enclosed in a rectangular frame of size 4×4, and thus there is one empty position. Any neighboring tile can be moved into this empty position. The game starts with an arbitrary initial distribution of the 15 tiles and the empty position. Figure 10.1 gives an example of initial and final configuration. .

1		3	4
5	2	7	8
9	6	10	11
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 10.1.: Example positions for the 15-puzzle.

I/O description. Input: tile numbers for initial position starting with the top line, e.g. the position in the example should be (actually one space is enough for separation):

```
1␣␣␣0␣␣3␣␣4
5␣␣␣2␣␣7␣␣8
9␣␣␣6␣10␣11
13␣14␣15␣12
```

Output: consecutive board positions in the same format as the input.

10.3.2. There are n cars[1] on a rail track, numbered with distinct values from the set $\{1, 2, \ldots, n\}$. A crane available at that location may pick k cars from the rail track and place them at the end of the sequence of cars (imagine this as being on the right hand side). Then, this cars are pushed to form a continuous line of cars.
Given the initial order of the cars, you are required to find (if possible) the minimum number of operations which the crane must execute to get the cars in sorted order, i.e $1, 2, \ldots, n$.
.
I/O description. Input: sequence of car numbers, separated by spaces all on one line. Output: sequence of configurations, one on a line, each line beginning with the number of operation, a colon and the list of cars. **E.g.** Input:

```
9
9␣2␣5␣8␣1␣5␣7␣6␣3␣4
```

Output (given just for how it looks, unchecked):

```
1:␣9␣5␣8␣1␣5␣7␣6␣3␣4␣2
```

10.3.3. There are $2n$ natives on a river bank. Out of these, n are man eaters. They wish to cross the river using a boat which can take at most k persons in one trip. If on either the bank or in the boat there are more cannibals than normal people, then the cannibals will eat the others. Find a way to take them all on the opposite bank without losing people to the cannibals and without changing numbers (i.e. use other people than the initial ones). .
I/O description. Input: n k, n = number of cannibals, k = boat capacity, e.g. for $n = 12$, and $k = 5$:

```
12␣5
```

Output (initial situation for the same values):

```
L:␣12c␣12n
B:␣0c␣0n
R:␣0c␣0n
```

Here `L` means left bank, `R` means right bank, and `B` means boat. Other letters mean `c`=cannibal, `n`=normal.

10.3.4. There are n goats lined on a bridge, going all in the same direction. From the opposite direction, other n goats are coming towards them. The goats cannot go around each other, but each goat can skip over one single goat of the opposite group and can advance if it there is empty space ahead. Using these two possibilities for a move, make the goat groups cross the bridge.
.
I/O description. Input: number of goats in one line. Output: sequence of configurations, e.g. initially:

```
[0]␣Op:␣none
a5␣a4␣a3␣a2␣a1␣___␣b1␣b2␣b3␣b4␣b5
[1]␣Op:␣advance␣a1
a5␣a4␣a3␣a2␣___␣a1␣b1␣b2␣b3␣b4␣b5
[2]␣Op:␣jump␣b1
a5␣a4␣a3␣a2␣b1␣a1␣___␣b2␣b3␣b4␣b5
```

Here `[0]` is the step number, `a5 a4 a3 a2 a1` is the line of goats going from left to right ___ denotes an empty position, and `Op:` indicates the operation one of `none, advance, jump`

Solve the following problems using divide and conquer:

[1]not automobiles!

10.3.5. The 4 peg version of Towers of Hanoi. There are four pegs: A, B, C and D. Initially peg A has on it a number of disks, the largest at the bottom and the smaller at the top, as the figure 10.2 shows. The task is to move the disks, one at a time from peg to peg, never placing a larger disk on top of a smaller one, ending with all disks on D. .

I/O description. Input: initial number of disks on peg A. Output: consecutive configurations.

E.g. for 7 disks the initial configuration would be:

```
A:␣1␣2␣3␣4␣5␣6␣7
B:
C:
D:
```

Here, the numbers represent disk sizes, the lowest number indicating the smallest disk.

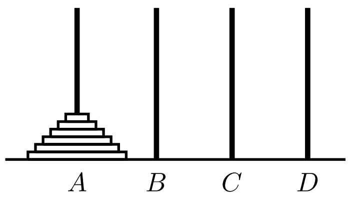


Figure 10.2.: Initial position in "towers of Hanoi".

10.3.6. Consider the design of a round robin chess tournament schedule (works for other contests as well, like volleyball, tennis, etc.), for $n = 2^k$ players. Each player (or team) must play every other player and each of them must play for $n-1$ days the minimum number of days needed to complete the tournament. We get a tournament table of n row per $n-1$ column, whose entry in row i and column j is the player i must contend with on the jth day.

.

I/O description. Input: the number of players. Output. The game table as shown below. Note that line 1 contains player numbers, as well as first column

E.g. For eight players, the input is:

```
8
```

and the output should be:

```
␣␣1␣2␣3␣4␣5␣6␣7
1␣2␣3␣4␣5␣6␣7␣8
2␣1␣4␣3␣6␣7␣8␣5
3␣4␣1␣2␣7␣8␣5␣6
4␣3␣2␣1␣8␣5␣6␣7
5␣6␣7␣8␣1␣4␣3␣2
6␣5␣8␣7␣2␣1␣3␣4
7␣8␣5␣6␣3␣2␣1␣4
8␣7␣6␣5␣4␣3␣2␣1
```

10.3.7. The problem is to tile a board of size $2^k \times 2^k$ with one single tile and $2^{2k}-1$ L–shaped groups of 3 tiles. A divide-and-conquer approach can recursively divide the board into four, and place a L-grouped set of 3 tiles in the center at the parts that have no extra tile, as shown in Figure 10.3.

.

I/O description. Input $k \leq 5$.
Output: board tiling.
Different colors should be symbolized with different lowercase letters. Thus, color 1 sould be marked with ’a’, color 2 with ’b’, etc.

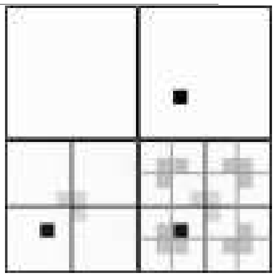


Figure 10.3.: Example tiling.

Given a set of n points (x_i, y_i) the problem asks what is the distance between the two closest points. A brute force approach in which the distances for all the pairs are measured takes $O(n^2)$ time.

A divide-and-conquer algorithm can sort the points along the x-axis, partition the region into two parts R_{left} and R_{right} having equal number of points, recursively apply the algorithm on the sub-regions, and then derive the minimal distance in the original region. The closest pair resides in the left region, the right region, or across the borderline. The last case needs to deal only with points at distance $d = \min(d_{left}, d_{right})$ from the dividing line, where d_{left} and d_{right} are the minimal distances for the left and right regions, respectively. 10.3.8.

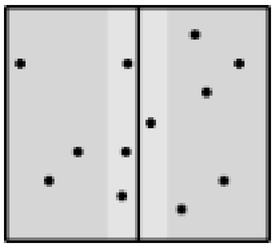


Figure 10.4.: Example closest pair problem.

The points in the region around the boundary line are sorted along the y coordinate, and processed in that order. The processing consists of comparing each of these points with points that are ahead at most d in their y coordinate. Since a window of size $d \times 2d$ can contain at most 6 points, at most five distances need to be evaluated for each of these points (see Figure 10.4).

The sorting of the points along the x and y coordinates can be done before applying the recursive divide-and-conquer algorithm.

.

I/O description. Input: n, the number of points, followed by point coordinates, one point per line (x_i, y_i). Output: coordinates of closest pair $(x_i, y_i)(x_j, y_j)$ on one line. Use parenthesis to mark pairs.

11. Algorithm Design. Dynamic Programming and Heuristics

11.1. Purpose

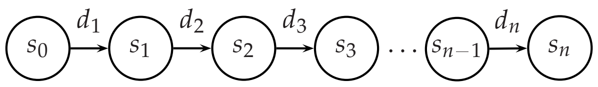
In this lab session we will experiment with dynamic programming and heuristics as algorithm development methods.

11.2. Brief Theory Reminder

Dynamic Programming

Dynamic programming is a method of algorithm development which is suited for finding optimal solutions. The solution is found by making a sequence of decisions which depend on previously made decisions, and all satisfy the principle of optimality. This principle may be stated as follows:

Let $\langle s_0, s_1, \ldots, s_n \rangle$ be a sequence of states, where s_0 is the initial state and s_n is the final state. The final state is reached by making a sequence of decisions, $d_1, d_2, \ldots, d_n$, as suggested by Figure 11.1.



$$\begin{tikzcd} s_0 \arrow[r, "d_1"] & s_1 \arrow[r, "d_2"] & s_2 \arrow[r, "d_3"] & s_3 \ \ldots \ s_{n-1} \arrow[r, "d_n"] & s_n \end{tikzcd}$$

Figure 11.1.: States and decisions.

If the sequence of decisions $\langle d_i, d_{i+1}, \ldots, d_j \rangle$ which changes state s_{i-1} into state s_j (with intermediate states $s_i, s_{i+1}, \ldots, s_{j-1}$) is optimal, and if for all $i \leq k \leq j-1$ both $\langle d_i, d_{i+1}, \ldots, d_k \rangle$ and $\langle d_{k+1}, d_{k+2}, \ldots, d_j \rangle$ are optimal sequences of decisions which transform state s_{i-1} into s_k, and state s_k into s_j respectively, then the principle of optimality is satisfied.

To apply this method we may proceed as follows:

- Check that the principle of optimality is satisfied.
- Write the recurrence equations which result from the rules which govern the state changes and solve them.

E.g. Consider the multiplication of a sequence of matrices

$$R = A_1 \times A_2 \times \ldots \times A_n,$$

where A_i with $1 \leq i \leq n$ is of size $d_i \times d_{i+1}$. The resulting matrix, R, will be of size $d_1 \times d_{n+1}$. It is common knowledge that when multiplying two matrices, say A_i and A_{i+1} we must effect $d_i \times d_{i+1} \times d_{i+2}$ multiplications. If the matrices to be multiplied have different numbers of rows/columns, then the number of operations used to get R depends on the order of performing the multiplications. What we would like to find is the order of performing these multiplications which results in a minimum number of operations.
Let C_{ij} be the minimum number of multiplications for calculating the product $A_i \times A_{i+1} \times \ldots \times A_j$ for $1 \leq j \leq n$. Note that:

- $C_{i,i} = 0$, i.e., multiplication for a chain with only one matrix is done at no cost.
- $C_{i,i+1} = d_i \times d_{i+1} \times d_{i+2}$.
- $C_{1,n}$ will be the minimum value.
- The principle of optimality is observed, i.e.

  $$C_{i,j} = \min[C_{i,k} + C_{k+1,j} + d_i \times d_{k+1} \times d_{j+1},$$

  for $i \leq k < j$, and the associations are $(A_i \times A_{i+1} \times \ldots \times A_k) \times (A_{k+1} \times A_{k+2} \times \ldots \times A_j)$.

We have to compute $C_{i,i+d}$ for each level d till we get $C_{1,n}$. In order to build the binary tree which describes the order of multiplications, we will also retain the value of k which was used in obtaining the minimum, which enables us to show how the matrices are associated. The vertices of this tree will contain the limits of the matrix subsequence for which the matrices are associated. The root will be $(1, n)$, and a subtree rooted at $i, j)$ will have (i, k) and $(k + 1, j)$ as children, where k is the value for which the optimum is obtained.
Here is the C code implementing optimal matrix multiplication:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define  MAXN 10

typedef struct _NodeT
{
   long ind1, ind2;
   struct _NodeT *left,*right;
}
NodeT;

void matMulOrder( long C[ MAXN ][ MAXN ],
                  int D[ MAXN + 1 ], int n )
/* Determines optimal multiplication order for a sequence of matrices
   A1 × A2 × ... × An of sizes D1 × D2, D2 × D3, ..., Dn × Dn+1 */
{
  int i, j, k, l, pos;
  long min, q;

  for( i = 1; i <= n; i++ ) C[ i ][ i ] = 0;
  for(l = 1; l <= n - 1; l++ )
    for( i = 1; i <= n - l; i++ )
    {
      j = i + l;
      min = LONG_MAX;
      for( k = i; k <= j - 1; k++ )
      {
        q = C[ i ][ k ] + C[ k + 1 ][ j ] + (long) D[ i ] * D[ k + 1 ] * D[ j + 1];
        if( q < min)
        {
          min = q;
          pos = k;
        }
      }
      C[ i ][ j ] = min;
      C[ j ][ i ] = pos;
    }
}
NodeT *buildTree( NodeT *p, int i, int j, long C[ MAXN ][ MAXN ] )
{
  p = ( NodeT * )malloc( sizeof( NodeT ));
  p->ind1 = i;
  p->ind2 = j;
  if( i < j )
  {
    p->left = buildTree( p->left, i, C[ j ][ i ], C ) ;
    p->right = buildTree( p->right, C[ j ][ i ] + 1, j, C );
  }
  else
  {
    p->left = p->right= NULL;
  }
  return p;
}
void postOrder( NodeT *p, int level )
{
  int i;
  if( p != NULL )
  {
    postOrder( p->left, level + 1 );
    postOrder( p->right, level + 1 );
    for( i = 0; i <= level; i++) printf("  ");
```

```
      printf( "(%ld,␣%ld)\n", p->ind1, p->ind2 );
   }
}
void main(void)
{
  int i, j, n;
  long C[ MAXN ][ MAXN ];
  int D[MAXN+1]; /* matrix dimensions */
  NodeT *root = NULL;

  printf("\nInput␣no.␣of␣matrices:␣");
  scanf( "%d", &n );
  while ( getchar() != '\n' );
  printf("\nMatrix␣dimensions\n");
  for( i = 1; i <= n + 1; i++ )
  {
     printf("\nEnter␣dimension␣D[%d]=", i );
     scanf( "%d", &D[ i ] );
     while ( getchar() != '\n' );
  }
  matMulOrder( C, D, n );
  /* computed cost matrix C */
  printf("\nCost␣matrix␣C\n");
  for(i = 1; i <= n; i++ )
  {
     for( j = 1; j <= n; j++)
     printf( "%6ld", C[ i ][ j ] );
     printf( "\n" );
  }
  printf("\nMinimum␣no.␣of␣multiplications␣=␣%ld", C[ 1 ][ n ] );
  while ( getchar() != '\n' );
  root = buildTree( root, 1, n, C );
  printf("\nPostorder␣traversal␣of␣tree\n");
  postOrder( root, 0 );
  while ( getchar() != '\n' );
}
```

Heuristics

An *heuristic* algorithm is an algorithm which gives an approximate solution, not necessarily optimal, but which can be easily implemented and has a polynomial growth rate. Heuristics are used in solving problems where a global optimum is not known to exist and where results close to optimum are acceptable. One way to cope with such problems is to decompose the process of finding a solution into successive processes for which the optimum is searched. Generally, this does not lead to a global optimum, as local optima do not imply the global optimum is reached. In practice, a number of approximate solutions are found, and the best of them is then selected.
E.g. Let us now consider the traveling salesman problem. Let $G = (X, \Gamma)$ be an undirected complete[1] graph with a strictly positive cost associated to each edge. We are required to find a cycle, starting with vertex i, which passes through all nodes exactly once, and ends at i. It is possible to find an optimal solution, using backtracking, but that will take exponential time.
The heuristic for which the code is given below, uses the greedy method and requires polynomial time. The steps to obtain the solution are:
If $(v_1, v_2, \ldots, v_k)$ is the already built path then

- If $\{v_1, v_2, \ldots, v_k\} = X$ then add edge (v_k, V_1) and the cycle is completed.
- If $\{v_1, v_2, \ldots, v_k\} \neq X$ then add the minimum cost edge which connects v_k to a vertex in X not included yet.

As a cycle is a closed path, we may start at any one node. So we could pick up each of $1, 2, \ldots, n$, in turn, as a start vertex, find the cycle, and retain the minimum of them all. Here is the code for beginning with node i:

```
#include <stdio.h>
#include <limits.h>
```

[1]remember: complete=all edges are present

```
#define MAXN 10
#define INFTY INT_MAX

void TravSMan( int n, int c[ MAXN ][ MAXN ],
               int i, int tour[ MAXN+1 ],
               int *cost )
/* n = number of nodes
   C = matrix of costs
   i = start vertex
   tour = vector containing the vertices of the tour
   cost = cost of the tour */
{
  int p[ MAXN ];
  int k, v, j, vmin;
  int costmin;

  for( k = 1; k <= n; k++ ) p[k] = 0;
  *cost = 0;
  p[ i ] = 1;
  tour[ 1 ] = i;
  v = i; /* current node */
  for ( k = 1; k < n; k++ )
  { /* add n–1 edges, one by one */
    costmin = INFTY;
    /* find edge of minimum cost which originates at vertex v */
    for ( j = 1; j <= n; j++ )
      if( p[ j ] == 0 && c[ v ][ j ] < costmin )
      {
        costmin = c[ v ][ j ];
        vmin = j;
      }
    *cost = *cost + costmin;
    tour[ k+1 ] = vmin;
    p[ vmin ] = 1;
    v = vmin;
  }
  tour[ n + 1 ] = i;
  *cost = *cost + c[ v ][ i ];
}
void main(void)
{
  int i, j, n;
  int tourCost;
  int tour[ MAXN + 1 ];
  int c[ MAXN ][ MAXN ];

  printf("\nNo.␣of␣nodes␣in␣the␣graph=");
  scanf("%d",&n);
  while ( getchar() != '\n' );
  for(i = 1; i <= n; i++ )
    for( j = 1; j <= n; j++ ) c[ i ][ j ] = INFTY;
  printf( "\nInput␣costs␣for␣edges␣with␣tail" );
  printf( "\nat␣node␣i␣and␣head␣at␣node␣j␣>␣i." );
  printf( "\nEnd␣input␣by␣entering␣0\n");
  for( i = 1; i <= n − 1; i++ )
  {
    for ( ; ; )
    {
      printf("Node␣adjacent␣to␣node␣%d=", i );
      scanf( "%d", &j );
      while ( getchar() != '\n' );
      if( j != 0 )
      {
        printf( "Cost␣of␣edge␣(%d,%d)=", i, j );
        scanf( "%d", &c[ i ][ j ] );
        while ( getchar() != '\n' );
        c[ j ][ i ] = c[ i ][ j ];
      }
      else break;
    }
  }
  i = 1;
  TravSMan( n, c, i, tour, &tourCost );
  printf("\nTour␣cost=%d\n", tourCost );
  printf("\nTour␣is:␣");
  for( i = 1; i <= n + 1; i++ )
    printf("%3d",tour[i]);
  while ( getchar() != '\n' );
}
```

11.3. Lab.11 Assignments

Use dynamic programming to solve the following problems:

11.3.1. The input to this problem is a pair of strings of letters $A = a_1 \dots a_m$ and $B = b_1 \dots b_n$. The goal is to convert A into B as cheaply as possible. The rules are as follows:

- For a cost of 3 you can delete any letter.
- For a cost of 4 you can insert a letter in any position.
- For a cost of 5 you can replace any letter by any other letter.

For example, you can convert A = `abcabc` to B = `abacab` via the following sequence: `abcabc` at a cost of 5 can be converted to `abaabc`, which at cost of 3 can be converted to `ababc`, which at cost of 3 can be converted to `abac`, which at cost of 4 can be converted to `abacb`, which at cost of 4 can be converted to `abacab`. Thus the total cost for this conversion would be 19. This is almost surely not the cheapest possible conversion.
.
I/O description. Input: the two strings on separate consecutive lines.
Output: sequence of transformed strings to turn first string into the second at a minimal cost. Each intermediate line should contain the operation coded as `d`=delete, `i`=insert, `r`=replace, and the changed string. The final line should contain the total cost.
E.g. For the example above we would have, as input:

```
abcabc
abacab
```

Output:

```
r␣abaabc
d␣ababc
d␣abac
i␣abacb
i␣abacab
19
```

11.3.2. The input to this problem consists of an ordered list of n words. The length of the ith word is w_i, that is the ith word takes up w_i spaces.
The goal is to break this ordered list of words into lines, this is called a *layout*. Note that you can not reorder the words. The length of a line is the sum of the lengths of the words on that line. The ideal line length is L. No line may be longer than L, although it may be shorter. The penalty for having a line of length K is $L - K$. The total penalty is the **maximum** of the line penalties. The problem is to find a layout that **minimizes** the total penalty.
HINT: Consider whether how many layouts of the first m words, which have k letters on the last line, you need to remember. .
I/O description. Input: the list of words in order, one word per line. The length of each word it is the actual length of that word plus one (one space). Output: the words on each line, line by line.

11.3.3. The input to this problem is a sequence of n points $p_1, \ldots, p_n$ in the Euclidean plane. You are to find the shortest routes for two taxis to service these requests in order.
The two taxis start at the origin. If a taxi visits a point p_i before p_j then it must be the case that $i < j$. (Think about what this last sentence means.) Each point must be visited by at least one of the two taxis. The cost of a routing is just the total distance traveled by the first taxi plus the total distance traveled by the second taxi. Design and implement an efficient algorithm to find the minimum cost routing.
HINT: Consider exhaustively enumerating the possible tours one point at a time. So after the ith stage you would consider all ways to visit the points $p_1, \ldots, p_i$. Then find a pruning rule that will reduce the number of tours we need to remember down to a polynomial number. .
I/O description. Input: n on one line, followed by n lines of integer x y coordinates. Output: length of route of first taxi, a colon, followed by the sequence of points passed by first taxi, between parenthesis, and separated by a space on one line, followed by the route for the second taxi in an identical format.

11.3.4. The input consists of a sequence $R = \langle R_0, \ldots, R_n \rangle$ of non-negative integers, and an integer k. The number R_i represents the number of users requesting some particular piece of information at time i (say from a www server). If the server broadcasts this information at some time t, the the requests of all the users who requested the information strictly before time t are satisfied. The server can broadcast this information at most k times. The goal is to pick the k times to broadcast in order to minimize the total time (over all requests) that requests/users have to wait in order to have their requests satisfied.
As an example, assume that the input was $R = 3, 4, 0, 5, 2, 7$ (so $n = 6$) and $k = 3$. Then one possible solution (there is no claim that this is the optimal solution) would be to broadcast at times $2, 4$, and 7 (note that it is obvious that in every optimal schedule that there is a broadcast at time $n + 1$ if $R_n \neq 0$). The 3 requests at time 1 would then have to wait 1 time unit. The 4 requests at time 2 would then have to wait 2 time units. The 5 requests at time 4 would then have to wait 3 time units. The 2 requests at time 5 would then have to wait 2 time units. The 7 requests at time 6 would then have to wait 1 time units. Thus the total waiting time for this solution would be $3 \times 1 + 4 \times 2 + 5 \times 3 + 2 \times 2 + 7 \times 1$. .
I/O description. Input: n and k, separated by one space on the first line, then R on second line. Output: the sequence of the k times.

11.3.5. The input to this problem is a sequence S of integers (not necessarily positive). The problem is to find the consecutive subsequence of S with maximum sum. "Consecutive" means that you are not allowed to skip numbers. For example if the input was 12,-14, 1, 23,-6, 22,-34, 13 the output would be 1, 23,-6, 22. Give, if possible, a linear time algorithm for this problem.
HINT: As a first step you might determine why a naive recursive solution is not possible. That is, figure out why knowing the nth number, and the maximum consecutive sum of the first $n-1$ numbers, is not sufficient information to compute the maximum consecutive sum of the first n numbers. There are examples, which show you need to strengthen the inductive hypothesis, should give you a big hint how to strengthen the inductive hypothesis. Then strengthen the inductive hypothesis to compute two different different consecutive sub-sequences, the maximum consecutive sum subsequence, and one other one. .
I/O description. Input integers separated by a single space, on one line. Output: required sequence on one line, numbers separated by one space.

11.3.6. The input to this problem is two sequences of letters $T = \langle t_1, \ldots, t_n \rangle$ and $P = \langle p_1, \ldots, p_k \rangle$ such that $k \leq n$, and a positive integer cost c_i associated with each t_i. The problem is to find a subsequence of of T that matches P with **maximum** aggregate cost. That is, find the sequence $i_1 < \ldots < i_k$ such that for all $j, 1 \leq j \leq k$, we have $t_{i_j} = p_j$ and $\sum_{j=1}^{k} c_{i_j}$ is maximized.
So for example, if $n = 5, T = XYXXY, k = 2, P = XY, c_1 = c_2 = 2, c_3 = 7, c_4 = 1$ and $c_5 = 1$, then the optimal solution is to pick the second X in T and the second Y in T for a cost of 7 + 1 = 8. .
I/O description. Input: sequence T on one line, followed by corresponding costs on the second line, followed by sequence P on the third line. Output: the matched sequence, a space, the optimal cost, a colon, then the list of indices of letters separated by one space.

11.3.7. The input to this problem is a set of n gems. Each gem has a value in €and is either a ruby or an emerald. Let the sum of the values of the gems be L. The problem is to determine if it is possible to partition of the gems into two parts P and Q, such that each part has the same value, the number of rubies in P is equal to the number of rubies in Q, and the number of emeralds in P is equal to the number of emeralds in Q. Note that a partition means that every gem must be in exactly one of P or Q. You algorithm should run in time polynomial in $n + L$. HINT: Start as in the subset sum problem (cf. http://en.wikipedia.org/wiki/Subset_sum_problem. Your pruning rule will have to be less severe. That is, first ask yourself why you may not be able to prune two potential solutions that have the same

aggregate value. .
I/O description. Input: number of gems on the first line; sequence of values of rubies on the second line; sequence of values of emeralds on the third lines. Output: partition P contents on one line, specifying each gem as a letter-value pair, with `r` for ruby and `e` for emerald. Thus, a ruby of value €5 would be specified in output as `r,5`.

11.3.8. A sequence S of integers (not necessarily negative) is given. The problem is to find the consecutive subsequence of S with minimum sum. "Consecutive" means that you are not allowed to skip numbers. For example if the input was 12,-14, 1, -23,-6, 22,-34, 13 the output would be -14, 1, -23,-6, 22, -34. Give, if possible, a linear time algorithm for this problem.
.
I/O description. Input integers separated by a single space, on one line. Output: required sequence on one line, numbers separated by one space.

---

12. Fundamental Sorting Algorithms

12.1. Purpose

In this lab session we will experiment with a number of sorting algorithms. specifically: counting sort, insertion sort (direct insertion and shellsort), sorting by swapping (bubblesort and quicksort), selection sort and mergesort.

12.2. Brief Theory Reminder

To sort means to arrange a sequence of elements, like $\langle a_0, a_1, \dots, a_{n-1}\rangle$ in ascending or descending order. In practice sorting is used to arrange records based on a key – the key is a field in the record.

Counting Sort

Counting sort consists in finding the number of elements smaller than each element a_i in the sequence $\langle a_0, a_1, \dots, a_{n-1}\rangle$. These numbers are stored in another sequence, c. The elements of the sequence to sort are initially duplicated as another sequence, b. Based on the elements of c, the elements of b are arranged in vector a. The sequences may be represented as vectors. A major shortcoming of counting sort is the fact that it uses two working areas of size n. The running tome is $O(N^2)$. An implementation of counting sort is given in Listing 12.1 as function $countingSort$.

Insertion Sort

Insertion sort takes a sorted sequence $a_0 < a_1 < a_2 < \dots < a_{j-1}$ and inserts a new element, a_j, in its rightful position by comparing it with $a_{j-1}, a_{j-2}, \dots$ till it finds an element $a_i < a_j$. Then, it inserts a_j in the position following a_i. To make room for the element to insert, all the elements in positions $j-1, j-2, \dots, i+1$, i.e. $a_{j-1}, a_{j-2}, \dots a_{i+1}$ are shifted one position right in the sequence. This method is called *direct* insertion.
Binary insertion implies executing a binary search for finding the position where a_j is to be inserted, based on the fact that the sequence $\langle a_0, a_1, \dots, a_{j-1}\rangle$ is arranged in ascending order.
Another insertion sort method is known as $shellsort$. To understand how this works, we will use the concept of $h-$sorting. $h-$ sorting means direct insertion of the sequences:

$$\begin{array}{l}\langle a_0, a_h, a_{2h}, \dots\rangle \\ \langle a_1, a_{1+h}, a_{1+2h}, \dots\rangle \\ \vdots \\ \langle a_h, a_{2h}, a_{3h}, \dots\rangle\end{array}$$

h is called an *increment*. Then, shellsort involves the selection of k increments, in descending order, i.e.

$$h_1 > h_2 > h_3 > \dots > h_k = 1$$

and performing a h_1-sort, then a h_2-sort, and so on, and finally a $1-$sort.
The performance of the method is tightly connected to the selection of the increments. In the example code given in Listing 12.1, the function $directInsertSort$ implements sorting by direct insertion, and the function $shellSort$ implements shellsort.
The running time for direct insertion sort is $O(n^2)$, and the growth rate of shellsort is $O(n \log n)$. A similar growth rate – $O(n \log n)$ – is found for binary insertion sort.

Swap Sort

Swap sort means to execute successive swaps of elements, i.e. $a_i \leftrightarrow a_j$ till all the elements of the sequence are arranged in ascending order. Methods based on swaps are bubblesort and quicksort.
Bubblesort is achieved by comparing elements a_i and a_{i+1} and, if $a_i \leq a_{i+1}$ proceeds further to comparing a_{i+1} and a_{i+2}. If $a_i > a_{i+1}$ then a_i and a_{i+1} are swapped. Then a_{i+1} is compared to a_{i+2}. After the first scan of the vector, the element of highest value gets in the last position; after the second scan the second highest gets in second-last position, and so on. The running time is $O(n^2)$.
Quicksort is attributed to C.A.R. Hoare and uses the divide-and-conquer approach. The principle of quicksort is: pick

one element of the array as a *pivot* and re-arrange the array in to sub-arrays such that the left sub-array will have all its elements smaller than the pivot and the right sub-array will have all its elements bigger than the pivot. Then recursively apply this to the left and right sub-arrays. The process stops when the pivot gets to the leftmost and the rightmost positions of the initial array, respectively. The running time for quicksort is $O(n \log n)$. In Listing 12.1, $bubbleSort$ and $quickSort$ implement the corresponding methods.

Selection Sort

Direct selection sort performs the following: find the minimum element, say a_j of $a_0, a_1, \dots a_{n-1}$ and move it to position 0 in the array by swapping $a_0 \leftrightarrow a_j$; then repeat this process choosing $a_1, a_2, \dots a_{n-1}$ as the element to swap with, in turn. The running time is $O(n^2)$.

Merge Sort

Merge sort was already described in §10.2 as an example of divide-and-conquer algorithm.

12.3. Sorting Algorithms Example Implementations

The code which follows implements the algorithms described in §12.2 to §12.2.

Listing 12.1: Sorting Algorithms Examples

```
#include <stdio.h>
#define MAXN 100

void readVector( int n, float a[ MAXN ],float b[ MAXN ] )
/* read the n elements of vector a   and copy the into vector b */
{
  int i;

  printf("\nInput␣elements␣of␣vector␣to␣sort\n");
  for( i = 0; i < n; i++ )
  {
    printf( "a[%d]=", i );
    scanf( "%f", &a[ i ] );
    while( getchar() != '\n' );
    b[ i ] = a[ i ];
  }
}
void reConstruct( int n, float a[ MAXN ],float b[ MAXN ] )
/* reconstruct the n-element vector a */
{
  int i;

  for( i = 0; i < n; i++ ) a[ i ] = b[ i ];
}

void listVector( int n, float a[ MAXN ] )
/* list the n elements of vector a in groups of 10 on each row */
{
  int i;

  for(i = 0;i < n; i++)
  {
    printf( "%8.2f", a[ i ] );
    if( ( i + 1 ) % 10 == 0 ) printf("\n");
  }
}
void  countingSort( int n, float a[ MAXN ] )
/* implements counting sort */
{
  int i, j;
  float b[ MAXN ];
  int c[ MAXN ];
  for( i = 0; i < n; i++ )
  {
    c[ i ] =0; /* initialize counting vector */
    b[ i ] =a[ i ]; /* copy vector a to b */
  }
  /* count */
  for( j = 1; j < n; j++ )
    for( i = 0; i <= j - 1; i++)
      if( a[ i ] < a[ j ] ) c[ j ]++;
      else                  c[ i ]++;
  /* rearrange vector a */
  for( i = 0; i < n; i++ ) a[ c[ i ]] = b[ i ];
}
void directInsertionSort( int n, float a[ MAXN ] ) /* implements direct insertion sort */
{
  int i, j;
  float x;

  for( j = 1; j < n; j++ )
  {
    x = a[ j ];
    i = j - 1;
    while( i >= 0  && x <a[ i ] )
    {
      a[ i + 1 ] = a[ i ];
      i =i - 1;
    }
    a[ i + 1] = x;
  }
}
void shellSort( int n, float a[ MAXN ] ) /* implements shellsort */
{
  int i, j, incr;
  float x;

  incr = 1;
  while( incr < n ) incr = incr * 3 + 1;
  while( incr >= 1 )
  {
    incr /= 3;
    for( i = incr; i < n; i++ )
    {
      x = a[ i ];
      j = i;
      while( a[ j - incr ] > x )
      {
        a[ j ] = a[ j - incr ];
        j = j - incr;
        if( j < incr ) break;
      }
      a[ j ] = x;
    }
  }
}
void bubbleSort( int n, float a[ MAXN ] )
/* implements bubblesort */
{
  int i, j, fin;
  float x;

  j = 0;
  do
  {
    fin = 1;
    j = j + 1;
    for( i = 0; i < n - j; i++ )
      if( a[ i ] > a[ i + 1 ] )
      { /* swap */
        fin = 0;
        x = a[ i ];
        a[ i ] = a[ i + 1 ];
        a[ i + 1 ] = x;
      }
   }
   while ( !fin );
}
void quick( int left, int right, float a[ MAXN ] )
{
  int i, j;
  float pivot, x;

  i = left;
  j = right;
  pivot = a[ ( left + right ) / 2 ];
  do
  {
    while( a[ i ] < pivot ) i++;
    while( a[ j ] > pivot ) j--;
    if( i <= j)
    { /* swap */
      x = a[ i ];
      a[ i ] = a[ j ];
      a[ j ] = x;
      i++;
      j--;
    }
  }
  while( i <= j );
  if( left < j ) quick( left, j, a );
  if( i < right) quick( i, right, a );
}
void quicksort( int n,float a[ MAXN ] ) /* implements quick sort (together with quick) */
{
  quick( 0, n - 1, a );
}
void selectionSort( int n, float a[ MAXN ] ) /* implements selection sort */
{
  int i, j, poz;
  float x;

  for( i = 0; i < n - 1; i++ )
  {
    x = a[ i ];
    poz = i;
    for( j = i + 1; j < n; j++ )
      if( a[ j ] < x )
      {
        x = a[ j ];
        poz = j;
      }
      a[ poz ] = a[ i ];
      a[ i ] = x;
  }
}
void main(void)
{
  int i, n;
  float a[ MAXN ],b[ MAXN ];

  printf( "\nNumber␣of␣elements␣to␣sort=" );
  scanf( "%d", &n );
  while( getchar() != '\n' );
  readVector( n, a, b );
  printf("\nUnsorted␣vector␣is\n");
  listVector( n, a );
  printf( "\nVector␣sorted␣by␣counting:\n" );
  countingSort( n, a );
  listVector( n, a );
  while( getchar() != '\n' );
  printf( "\nSorted␣by␣direct␣insertion:\n" );
  reConstruct( n, a, b );
  directInsertionSort( n, a );
  listVector( n, a );
  while( getchar() != '\n' );
  printf( "\nShellsorted␣vector:\n" );
  reConstruct( n, a, b );
  shellSort( n, a );
  listVector( n, a );
  while( getchar() != '\n' );
  printf( "\nBubblesorted␣vector:\n" );
  reConstruct( n, a, b );
  bubbleSort( n, a );
  listVector( n, a );
  while( getchar() != '\n' );
  printf( "\nQuicksorted␣vector:\n" );
  reConstruct( n, a, b );
  quicksort( n, a );
  listVector( n, a );
  while( getchar() != '\n' );
  printf( "\nSorted␣by␣selection:\n" );
  selectionSort( n, a );
  listVector( n, a );
  while( getchar() != '\n' );
}
```

12.4. Laboratory Assignments

12.1. Given the sequence of integers

5 9 1 7 4 3 2 0

manually arrange this sequence in ascending order using the three "‘elementary"’ sorting methods: insertion sort, bubblesort and selection sort, showing at each step the new configuration of the sequence. How many comparisons and how many element moves were used by each method? Which is the best performing method for sorting this array of integers? Which would be the worst arrangement of this sequence?

12.2. Describe an algorithm for insertion sort, by changing linear search into a binary search performed on the left sub-array of the current array. Calculate the number of steps, number of comparisons and the number of moves for this algorithm (called insertion sort with binary search) operating on the data given in the previous problem. Is this algorithm performing better?

12.3. Given the sequence of reals:

-3.1 0.1 1.2 -5.7 0.3 6

apply shellsort and quicksort showing the new configuration of the sequence, step by step. Count the number of comparisons and element moves. Which of the two algorithms performs better on this data?

12.4. Analyze quicksort and replace the recursive version with a non-recursive one. What variables should be initialized, with what values, and where should $goto$ be placed in order to eliminate recursion?

12.5. Which of the algorithms shown here has the biggest memory requirements?

12.6. Adapt quicksort to find the m^{th} smallest element out of a sequence of n integers.

12.7. Sort n elements, numbered from 1 to n and characterized by precedence relations, given as pairs (j, k) with the meaning j *precedes* k. The sort process must result in a list where element k must be located after its predecessor (topological sort, again).

12.8. Describe an algorithm which performs the following operations encountered when processing data for an admission exam:

a) Calculate the average scores for the candidates
b) Assign places to the accepted candidates, based on m options they may have, and print the list.
c) List all rejected candidates in descending order.

The exam is composed of two tests, graded with reals in the range $[1, 10]$. When average scores (truncated to two decimal positions right of the point) are equal, the score of the first test and then the scores of the second test is used to decide position. If equality persists, increase the number of available positions from a certain option. .
I/O description. Input:

- number of options, m alone on one line
- pairs $option\ maximumnumberofadmittedcandidates$, separated by blanks, each pair on one line
- candidate data, one candidate on each line:

$$Name, score_1, score_2, opt_1, opt_2, \ldots, opt_m$$

E.g. Input

```
4
1␣25
2␣30
3␣35
4␣20
"Doe␣John",9.30,9.80,4,2,1,3
"Doe␣Jane",9.70,9.70,1,2,4,3
...
```

Output:

```
Successful␣candidates␣for␣option␣1
1.␣Doe␣Jane␣ 9.70
...
...
Successful␣candidates␣for␣option␣4
1.␣Doe␣John␣␣␣␣␣␣9.55
Unsuccessful␣candidates
1.␣Jones␣Jim␣␣␣␣␣␣4.99
...
```

Bibliography

[***99] ***. Programming in C. WWW: http://www.lysator.liu.se/c/index.html, 1999. Lynkoeping University, Sweden.

[AHU87] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, 1987.

[AS96] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. MIT Press, USA, second edition, 1996. WWW: http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-1.html.

[Buh02] Jeremy Buhler. CS241 – Algorithms and Data Structures. Handouts. WWW: http://students.cec.wustl.edu/ cs241/handouts, 2002. Washington University in St. Louis, USA, Center for Engineering Computing.

[Del02] DJ Delorie. DJGPP - A free 32-bit development system for DOS. WWW: http://www.delorie.com, 2002.

[Dev97] Luc Devroye. Data Structures and Algorithms. Class Notes Compiled by Students. http://cgm.cs.mcgill.ca/ luc/1997notes.html, 1997. School of Computer Science, McGill University, Montreal, Canada.

[Dow02] Allen B. Downey. How To Think Like A Computer Scientist. Learning with C++. WWW:http://www.ibiblio.org/obp/thinkCScpp/, 2002.

[GBY90] Gaston H. Gonnet and Ricardo Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley Publishing Co. Inc., 1990. WWW: http://www.dcc.uchile.cl/ rbaeza/handbook/hbook.html.

[Haa99] Johan Haastad. Advanced Algorithms. Lecture Notes. WWW: http://www.oopweb.com/Algorithms/Download/algnotes.zip, 1999.

[Hoh02] Robert Hohne. Interactive development environment - RHIDE. WWW: http://www.rhide.com, 2002.

[Hol95] Steve Holmes. C Programming. WWW: http://www.strath.ac.uk/IT/Docs/Ccourse/, 1995. University of Strathclyde Computer Center, Glasgwo, UK.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Fundamental Algorithms. Addison Wesley, Reading, Mass., USA, second edition, 1973.

[Koe89] Andrew Koenig. *C Traps and Pitfalls*. Addison-Wesley Publishing Company, 1989. PDF version: http://www.programmersheaven.com/search/download.asp?FileID=391.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, first edition, 1978.

[Les96] Martin Leslie. C Programming Reference. Release 1.06. WWW: http://mip.ups-tlse.fr/C_ref/C/cref.html, 1996.

[Mor98] John Morris. Data Structures and Algorithms. Lecture Notes. WWW: http://www.oopweb.com/Algorithms/Documents/PLDS210/VolumeFrames.html, 1998. Electrical and Electronic Engineering, University of Western Australia.

[Mou99] David M. Mount. Data Structures – CMSC420. Lecture Notes. WWW: http://www.oopweb.com/Algorithms/Download/420lects.zip, 1999.

[Nav] Jacob Navia. LCC-Win32: a free compiler system for Windows. WWW: http://www.cs.virginia.edu/ lcc-win32/.

[NIS00] NIST. *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology, USA, 2000. WWW: http://www.nist.gov/dads.

[Oua92] Steve Oualline. *C Elements of Style*. M&T Books, 1992. WWW: http://www.oualline.com/style.

[Oua97] Steve Oualline. *Practical C Programming*. O'Reilly & Associates, 1997. Book examples: http://examples.oreilly.com/pcp3.

[PB01] P.J. Plauger and Jim Brodie. *Dinkum C99 Library*. Dinkumware Ltd., Concord MA, USA, 2001. WWW: http://www.dinkumware.com/refxc.html.

[Ski98] Steven S. Skiena. The Stony Brook Algorithm Repository. WWW: http://www.cs.sunysb.edu/ algorithm, 1998. Computer Science, State University of New York.

[Sof02] Bloodshed Software. The Dev-C++ Resource Site. WWW:http://www.bloodshed.net/dev/devcpp.html, 2002.

[Sum95] Steve Summit. `comp.lang.c` Frequently Asked Questions. WWW: http://www.eskimo.com/ scs/C-faq/top.html, 1995.

[Sym99] Antonios Symvonis. Algorithms – COMP 3001. Lecture Notes. WWW: http://www.oopweb.com/Algorithms/Download/Algorithms.zip, 1999.

[Ven02] Andreas Veneris. ECE242: Algorithms and Data Structures. Lecture Notes. WWW: http://www.ecf.utoronto.ca/apsc/courses/ece242f/, 2002. University of Toronto, Canada.

[Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, Englewood Cliffs, NJ, USA, 1976.

A. Programming Style Guide

A.1. Why Need a Style Guide?

This addenda contains excerpts from a *programming style guide* (or just *style guide*). Style guides are usually lengthy documents containing tens or hundreds of guidelines that a programmer is directed to follow in the course of constructing source code.

There are many good reasons for having a style guide. These are three of the most important reasons:

To eliminate flaws. Many guidelines represent good programming practice and help programmers avoid pitfalls in their code. Following these guidelines will raise the quality of your code and reduce the chance of releasing a flawed product.

To promote source code portability. Many guidelines are directed explicitly towards making code portable at many levels: from hardware platform to hardware platform (e.g., Intel to Macintosh), operating system to operating system (e.g., Windows 95 to Windows NT), and from one release of a compiler to the next (e.g., the 16-bit compilers of Windows 3.x to the 32-bit compilers of Windows 95, NT, and 2000). At some point you are almost certainly going to have to port your code; following a good style guide will greatly reduce the effort required to do so.

To promote source code consistency. Having consistently written source code on a project greatly increases the readability of the code and makes it easier for programmers to move from module to module. For example, if you work on the user interface module, you will likely have reason at some point to examine the code in the data base module, possibly because you' re not sure whether an error resides in your code or the in someone else's. If the user interface code and data base code are all written according to consistent guidelines, you will be able to find and evaluate the data base code much more quickly and reliably.

All well-run projects will have a style guide and will expect programmers to follow it. Likewise, the projects that you do as part of this course must conform to the abbreviated style guide found in this addendum.

A.2. Code Documentation

Every file in your project must begin with a *descriptive box* that supplies the following information in the given order:

1. The name of the file.
2. The purpose of the file, briefly stated.
3. The course ID.
4. Your name.
5. Your e-mail address.
6. A module history supplying date of and reason for submission.

Please use the template of listing A.1 for your descriptive box:

Listing A.1: Descriptive box.

```
/*******************************************************************
 *
 * File Name:    list.h
 *
 * Purpose:      To supply prototypes and other
 *               declarations for the list module.
 *
 * Course ID:    DSAL.105.AC.01.04-B2/2003
 * Student Name: I. V. Ionescu
 * E-mail:       iionescu@asterix.obs.utcluj.ro
 *
 *******************************************************************
 * Module History
 *
 * Date        Description
 * -------     ------------------------
 * 06-mar-03   First submission of Project #1
 * 08-mar-03   Resubmit to correct error in prototype for
 *             LIST_createNew().
 ******************************************************************/
```

A.3. General Naming Conventions

Variable and parameter names will always begin with a lower-case letter.
Types defined will begin with an uppercase letter, and will end in the word "Type", e.g. `ElementType`
Defined constants will be all uppercase.

A.4. Preprocessor Statements

A.5. Portability Issues

Preprocessors are notoriously different in operation from one environment to another, which can make writing portable code difficult. To avoid the most common pitfalls, write your code according to the following guidelines:

- The pound sign in a preprocessor statements must always begin in column one.
- The preprocessor directive (e.g., define, include) must always follow the pound sign in column two.
- A preprocessor directive must always be followed by exactly one space.
- Identifiers in preprocessor statements must conform to ANSI requirements for construction; that is, they are a maximum of 32 characters, contain only letters, numerals, and underscores, and begin with a letter or underscore.

These are examples of *bad* preprocessor statements:

```
/* Do NOT write your code this way: */
#ifndef LIST.H  /* Illegal character in identifier */

#ifdef DEBUG
    #define TRACE (1)  /* # must be in column 1 */
#else
    #define TRACE (0)  /* # must be in column 1 */
#endif
    ...
#ifdef DEBUG
#   define TRACE (1) /* define must begin in column 2 */
#else
#   define TRACE (0) /* define must begin in column 2 */
#endif
```

The preceding code would be correctly written like this:

```
#ifndef LIST_H

#ifdef DEBUG
#define TRACE (1)
#else
#define TRACE (0)
#endif
```

A.6. Macros

- Macro names will always be in uppercase.
- Whenever possible, the body of a macro must be enclosed in parentheses, like this:

  ```
  #define MAX_SIZE      (200)
  ```

- An instance of an argument in a macro expansion must be enclosed in parentheses whenever possible, as in the following:

  ```
  #define SQ_ROOT( n ) (pow( (n), .5 ))
  ```

A.7. General Construction Conventions

- Compound statements will always begin with the opening brace on a line by itself, flush with the left margin of the block within which it is declared. Here's an example:

```
/* Instead of this: */
static void processFile( FILE *file ){/*WRONG!*/
    while ( !feof( file ) ) {\{}      /*WRONG!*/
/* or this: */
static void processFile( FILE *file ) /*WRONG!*/
{
  while ( !feof( file ) ) /*WRONG!*/
  {

/* Use this: */
static void processFile( FILE *file )
{
  while ( !feof( file ) )
  {
```

- Compound statements will be terminated with the closing brace on a line by itself, aligned directly underneath the opening brace. This is an example:

```
static void processFile( FILE *file )
{
  while ( !feof( file ) )
  {
       . . .
  }
  . . .
}
```

- Initialize a variable in its declaration whenever possible, like this:

```
/* Instead of this: */
int inx;

/* Use this: */
int inx = 0;
```

- Leave one space from the parentheses as you probably notices in all the supplied examples. This improves readability.

Listing A.3 shows the template to use for `*.h` files, and listing A.2 shows the template to use for `*.c` files:

Listing A.2: Template for *.c files

```
/*******************************************************************
*
* File Name:    <file name here>
*
* Purpose:      <purpose here>
*
* Course ID:    DSAL.105.AC.01.04-B2/2002
* Student Name: <your name here>
* E-mail:       <your email here>
*
********************************************************************
* Module History
*
* Date        Description
* -------     -------------------------
* dd-lll-aa   First submission of Project #1
*******************************************************************/
#include <system includes>
#include "<project includes specified relative
           to project base directory
           eg. inc/list.h>
```

```
/*
Global variables
----------------
*/
<variable declarations.  Same as in .h except without extern and with
 initializers>
/*
Module types
------------
*/
<declarations for types which are only used within the module.
 Should be storage class static>
/*
Module variables
----------------
*/
<declarations for variables which are global to this module only.
 Should be storage class static>
/*
Module functions
----------------
*/
<definitions for functions which are used in this module only.
 Should be storage class static>
/*
Global functions
----------------
*/
<definitions for the functions prototyped in .h>
```

Listing A.3: Template for *.h files

```
/*******************************************************************
*
* File Name:    <file name here>
*
* Purpose:      <purpose here>
*
* Course ID:    DSAL.105.AC.01.04-B2/2003
* Student Name: <your name here>
* E-mail:       <your email here>
*
**********************************************
* Module History
*
* Date        Description
* -------     -------------------------
* dd-lll-aa   First submission of Project #1
*********************************************/
#if !defined (<module name in capitals>_H)
#define <module name in capitals>_H

#include <system includes>
#include "<project includes specified relative to project base
           directory e.g. inc/list.h>
/*
Global types
------------
*/
<type definitions>
/*
Global variables
----------------
```

```
*/
<variable declarations. Should all be storage class extern>
/*
Global functions
----------------
*/
<prototypes for the functions that can be used by other modules>
#endif
```

B. C Language Reference

Program Structure and Functions

type $fnc(type_1, \ldots)$;	function declarations
type name;	external variable declarations
`main()`	main routine
`{`	
$declarations$	local variable declarations
$statements$	
`}`	
type $fnc(arg_1, \ldots)$	function definition
`{`	
$declarations$	local variable declarations
$statements$	
`return` $value$;	
`}`	
`/* */`	comments
`main(int args, char *argv[])`	main with args
`exit(`arg`)`	terminate execution

C Preprocessor

include library file	`#include` $< filename >$
include user file	`#include` $"filename"$
replacement text	`#define$` *name text*
replacement macro	`#define$` $name(var)$ *text*

E.g. `#define max(A,B) ((A)>(B) ? (A) : (B))`

undefine	`#undef` $name$
quoted string replace	`#`
concatenate args and rescan	`##`
conditional execution	`#if, #else, #elif, #endif`
is $name$ defined, not defined?	`#ifdef, #ifndef`
$name$ defined?	`defined(`$name$`)`
line continuation char	`\`

Data Types and Their Declaration

character (1 byte)	`char`
integer	`int`
float (single precision)	`float`
float (double precision)	`double`
short (16 bit integer)	`short`
long (32 bit integer)	`long`
positive and negative	`signed`
only positive	`unsigned`
pointer to `int, float,...`	`*int, *float,...`
enumeration constant	`enum`
constant (unchanging) value	`const`
declare external variable	`extern`
register variable	`register`
loal to source file	`static`
no value	`void`
structure	`struct`
create name by data type	`typedef` *typename*
size of an object (type is `size_t`)	`sizeof`$(object)$
size of a data type (type is `size_t`)	`sizeof`(*type name*)

Initialization

initialize variable	$type\ name = value;$
initialize array	$type\ name[] = \{value_1, \ldots\};$
initialize char string	`char` $name[] = "string";$

Constants

long (suffix)	`L` or `l`
float (suffix)	`F` or `f`
exponential form	`e`
octal (prefix 0)	`0`
hexadecimal (prefix zero-ex)	`0x` or `0X`
character constant (char, octal, hex)	`'a'`, `'\`*ooo*`'`, `\x`*hh*`'`
newline, cr, tab, backspace	`\n`, `\r`, `\t`, `\b`
special characters	`\\`, `\?`, `\'`, `\"`

Pointers, Arrays and Structures

declare pointer to $type$	$type\ *name$
declare function returning pointer to $type$	$type$ `*f()`
declare pointer to function returning $type$	$type$ `(*pf)()`
generic pointer type	`void *`
null pointer	`NULL`
object pointed to by $pointer$	$*pointer$
address of object $name$	$\&name$
array	$name[dim]$
multi-dimensional array	$name[dim_1][dim_2]\ldots$
Structures	
`struct` tag	structure tempalte
`{`	
$declarations$	declaration of members
`}`	
create strcuture	`struct` $tag\ name$
member of structure from template	$name.member$
member of pointed to structure	$pointer-> member$
	E.g. `(*p).x` and `p->x` are the same
single value, multiple type structure	`union`
bit field with b bits	$member : b$

Operators (grouped by precedence)

structure member operator	$name.member$
structure pointer	$pointer-> member$
increment, decrement	`++`, `--`
plus, minus, logical not, bitwise not	`+`, `-`, `!`, `~`
indirection via pointer	$*pointer$, $\&name$
cast expression $expr$ to $type$	$(type)\ expr$
size of an object	`sizeof`
multiply, divide, modulus (remainder)	`*`, `/`, `%`
add, subtract	`+`, `-`
left, right shift [bit ops]	`<<`, `>>`
comparisons	`>`, `>=`, `<`, `<=`
comparisons	`==`, `!=`
bitwise and	`&`
bitwise exclusive-or	`^`
bitwise or (inclusive-or)	`|`
logical and	`&&`
logical or	`||`
conditional expression	$expr_1 ?\ expr_2 :\ expr_3$
assignement operators	`+=`, `-=`, `*=`, `/=`, `...`
expression evaluation separator	`,`

Flow of Control

statement terminator	`;`
block delimiters	`{ }`
exit from `switch, while, do, for`	`break`
next iteration of `while, do, for`	`continue`
goto	`goto`
label	*label* :
return value from function	`return` *expr*
Flow Constructions	
`if` statement	`if (`*expr*`)` *statement* `else if (`*expr*`)` *statement* `else` *statement*

`while` statement	`while (`*expr*`)`
	  statement

`for` statement	`for (`$expr_1$; $expr_2$; $expr_3$`)`
	  statement

`do` statement	`do` *statement* `while (`*expr*`)`

`switch` statement	`switch (`*expr*`)`
	`{`
	  `case` $const_1$: $statement_1$ `break`
	  `case` $const_2$: $statement_2$ `break`
	  `default:` *statement*
	`}`

ANSI Standard Libraries

```
<assert.h>  <ctype.h>  <errno.h>   <float.h>   <limits.h>
<locale.h>  <math.h>   <setjmp.h>  <signal.h>  <stdarg.h>
<stddef.h>  <stdio.h>  <stdlib.h>  <string.h>  <time.h>
```

Character Class Tests <**`ctype.h`**>

alphanumeric?	`isalnum(c)`
alphabetic?	`isalpha(c)`
control char?	`iscntrl(c)`
decimal digit?	`isdigit(c)`
printing char (w/o space)	`isgraph(c)`
lower case letter	`islower(c)`
printing char (w/ space)	`isprint(c)`
printing char (except space, letter, digit)?	`ispunct(c)`
space, formfeed, newline, cr, tab, vtab?	`isspace(c)`
upper case letter?	`isupper(c)`
hexadecimal digit?	`isxdigit(c)`
convert to lower case	`tolower(c)`
convert to upper case	`toupper(c)`

String Operations <string.h>

`s,t` are strings `cs,ct` are constant strings

length of `s`	`strlen(s)`
copy `ct` to `s`	`strcpy(s,ct)`
up to `n` chars	`strncpy(s,ct,n)`
concatenate `ct` after `s`	`strcat(s,ct)`
up to `n` chars	`strncat(s,ct,n)`
compare `cs` to `ct`	`strcmp(s,ct)`
only first `n` chars	`strncmp(s,ct,n)`
pointer to first `c` in `cs`	`strchr(cs,c)`
pointer last `c` in `cs`	`strrchr(cs,c)`
copy `n` chars from `ct` to `s`	`memcpy(s.ct,n)`
copy `n` chars from `ct` to `s` (may overlap)	`memmove(s.ct,n)`

compare `n` chars of `cs` with `ct`	`memcmp(s.ct,n)`
pointer to first `c` in first `n` of`cs`	`memchr(cs,c,n)`
put `c` into first `n` chars of `s`	`memset(s.c,n)`

Input/Output <`stdio.h`>

Standard I/O

standard input stream	`stdin`
standard output stream	`stdout`
standard error stream	`stderr`
end of file	`EOF`
get a character	`getchar()`
print a character	`putchar(`chr`)`
print formatted data	`printf("`$format$`",`$arg_1, \ldots$`)`
print to string `s`	`sprintf(s,"`$format$`",`$arg_1, \ldots$`)`
read formatted data	`scanf("`$format$`",`$\&name_1, \ldots$`)`
read from string `s`	`sscanf(s,"`$format$`",`$\&name_1, \ldots$`)`
read line to string `s`(<`max` chars)	`gets(s,max)`
print string `s`	`puts(s)`

File I/O

declare file pointer	`FILE *`fp
pointer to named file	`fopen("`$name$`","`$mode$`")`
	modes: `r` (read), `w` (write), `a` (append)
get a character	`getc(`fp`)`
write a character	`getc(`chr, fp`)`
write to file	`fprintf(`fp`,"`$format$`",`$arg_1, \ldots$`)`
read from file	`fscanf(`fp`,"`$format$`",`$\&name_1, \ldots$`)`
close file	`fclose(`fp`)`
non-zero if error	`ferror(`fp`)`
non-zero if EOF	`feof(`fp`)`
read line to string `s`(<`max` chars)	`fgets(s,max,`fp`)`

Codes for Formatted I/O: `"%-+ 0`$w.pmc$

`-`	left justify
`+`	print with sign
$space$	print space if no sign
`0`	pad with leading zeros
w	min field width
p	precision
m	conversion character:
	`h` short, `l` long, `L` long double

`d,i`	integer	`u`	unsigned
`c`	single char	`s`	char string
`f`	float	`e,E`	exponential
`o`	octal	`x,X`	hexadecimal
`p`	pointer	`n`	number of chars written

`g,G` same as `f` or `e,E` depending on exponent

Variable Argument Lists <`stdarg.h`>

declaration of pointer to arguments	`va_list` $name$`;`
initialization of argument pointer	`va_start(`$name, lastarg$`)`
	$lastarg$ is the last named parameter of the function
access next unnamed arg, update pointer	`va_arg(`$name, type$`)`
call before exiting function	`va_end(`$name$`)`

Standard Utility functions <`stdlib.h`>

absolute value of `int n`	`abs(n)`

absolute value of `long n`	`labs(n)`
quotient and remainder of `int`s `n,d`	`div(n,d)`
returns structure with `div_t.quot` and `div_t.rem`	
quotient and remainder of `long`s `n,d`	`div(n,d)`
returns structure with `ldiv_t.quot` and `ldiv_t.rem`	
pseudo-random integer `[0,RAND_MAX]`	`rand()`
set random seed to `n`	`srand(n)`
terminate program execution	`exit(status)`
pass string `s` to system for execution	`system(s)`

Conversions

convert string `s` to double	`atof(s)`
convert string `s` to integer	`atoi(s)`
convert string `s` to long	`atol(s)`
convert prefix of `s` to double	`strtod(s,endp)`
convert prefix of `s` (base `b`) to `long`	`strtol(s,endp)`
same, but `unsigned long`	`strtol(s,endp)`

Storage Allocation

allocate storage	`malloc(size), calloc(nobj,size)`
change size of object	`realloc(pts,size)`
deallocate space	`free(ptr)`

Array Functions

search unsorted`array` for `key`	`lsearch(key,array,n,size,cmp())`
search sorted`array` for `key`	`bsearch(key,array,n,size,cmp())`
sort `array` ascending order	`qsort(array,n,size,cmp())`

Time and Date Functions

processor time used by program	`clock()`
E.g. `clock()/CLOCKS_PER_SEC` is time in seconds	
current calendar time	`time()`
`time`$_2$−`time`$_1$	`difftime(time`$_2$`,time`$_1$`)`
arithmetic types representing times	`clock_t,time_t`
structure type for calendar time	`tm`

`tm_sec`	seconds after minute
`tm_min`	minutes after hour
`tm_hour`	hours since midnight
`tm_mday`	day of month
`tm_mon`	months since January
`tm_year`	years since 1900
`tm_wday`	days since Sunday
`tm_yday`	days since January 1
`tm_istdst`	Daylight Savings Time flag

convert local time to calendar time	`mktime(tp)`
convert time in `tp` to string	`asctime(tp)`
convert calendar time in `tp` to local time	`ctime(tp)`

Mathematical Functions <`math.h`>

trig functions	`sin(x), cos(x), tan(x)`
inverse trig functions	`asin(x), acos(x), atan(x)`
arctan(y/x)	`atan2(y,x)`
hyperbolic trig functions	`sinh(x), cosh(x), tanh(x)`
exponentials and logarithms	`exp(x), log(x), log10(x)`
exponentials and logarithms (2 power)	`ldexp(x,n), frexp(x,*e)`
division and remainder	`modf(x,*ip), fmod(x,y)`
powers	`pow(x,y), sqrt(x)`
rounding	`ceil(x), floor(x), fabs(x)`

Integer Type Limits <`limits.h`>

`CHAR_BIT`	bits in `char`	(8)
`CHAR_MAX`	max value of `char`	$(127\ textrmor\ 255)$
`CHAR_MIN`	min value of `char`	$(-128$ or $0)$
`INT_MAX`	max value of `int`	$(+32,767)$
`INT_MIN`	min value of `int`	$(-32,768)$
`LONG_MAX`	max value of `long`	$(+2,147,483,647)$
`LONG_MIN`	min value of `long`	$(-2,147,483,648)$
`SCHAR_MAX`	max value of `signed char`	$(+127)$
`SCHAR_MIN`	min value of `signed char`	(-128)
`SHRT_MAX`	max value of `signed int`	$(+32,767)$
`SHRT_MIN`	min value of `signed int`	$(-32,768)$
`UCHAR_MAX`	max value of `unsigned char`	(255)
`UINT_MAX`	max value of `unsigned int`	$(65,536)$
`ULONG_MAX`	max value of `unsigned long`	$(4,294,967,295)$
`USHRT_MAX`	max value of `unsigned short`	$(65,536)$

Float Type Limits <`float.h`>

`FLT_RADIX`	radix of exponent representation	(2)
`FLT_ROUNDS`	floating point rounding mode	
`FLT_DIG`	decimal digits of precision	(6)
`FLT_EPSILON`	smallest x so that $1.0 + x \neq 1.0$	(10^{-5})
`FLT_MANT_DIG`	number of digits in mantissa	
`FLT_MAX`	maximum floating point number	(10^{37})
`FLT_MAX_EXP`	maximum exponent	
`FLT_MIN`	minimum floating point number	(10^{-37})
`FLT_MIN_EXP`	minimum exponent	
`DBL_DIG`	decimal digits of precision in `double`	(10)
`DBL_EPSILON`	smallest x so that $1.0 + x \neq 1.0$	(10^{-9})
`DBL_MANT_DIG`	number of digits in mantissa	
`DBL_MAX`	max `double` floating point number	(10^{307})
`DBL_MAX_EXP`	maximum exponent	
`DBL_MIN`	minimum `double` floating point number	(10^{-307})
`DBL_MIN_EXP`	minimum exponent	

Index