Algorithms and Data Structures in C

Gerard R. Renardel de Lavalette et al.

2020–2021

Realization

This is the twelfth version of the lecture notes Algorithms and Data Structures in C. It was originally written by Gerard Renardel and contains several clarifications and improvements made by Malvin Gattinger.
The author thanks Wim Hesselink, Yuri Meiburg, Arnold Meijster and Mark IJbema for their contributions. They provided earlier versions of the sections on expressions and expression trees (Wim), the practical assignments (Mark and Arnold) and the appendix about C (Yuri).

Layout

These lecture notes have been produced with the lecture notes style from L-Space, developed by Mark IJbema and Thomas ten Cate.

Compilation date: 8th February 2022

Meaning of the icons


exercise, to be worked out on paper


practical assignment, leading to a computer program


a report is required, written with LaTeX


useful tip


Alert! common pitfall

Contents

Chapter 0

Introduction

These lecture notes are written for the course *Algorithms and Data Structures in C*. This is the second programming course in the first year of the bachelor programmes Computing Science and Artificial Intelligence at the University of Groningen. The course presumes programming skills with and knowledge about the programming language C, including integers, floating point numbers, characters, arrays, strings and pointers. These subjects are treated in the course *Imperative Programming*.

These lecture notes contain the subject material and the tutorial exercises for the course.

What is an algorithm? It is an abstract description of a computation, abstracting away from implementation details. A computer program can be an implementation of an algorithm. In these notes, C is used to implement algorithms. We also introduce pseudocode, a semi-formal notation for algorithms.

In Chapter 1 we begin with linear data structures: stacks, queues, linked lists. They are presented as *abstract data structures (ADTs)*, abstracting away from implementation details. As an application, we use linked lists in the development of a parser and an evaluator for simple arithmetical expressions. In Chapter 2 trees are presented, mainly binary trees. They have several applications: search trees, heaps, tries, and expression trees. Chapter 3 is devoted to graphs and several graph algorithms: depth-first search, breadth-first search, Dijkstra's shortest path algorithm and the A* algorithm.

The appendices contain additional information about C and the programming environment, the definition of pseudocode and the $\mathcal{O}$ notation, naming and layout conventions for programs, and guidelines for writing a programming report.

For C issues, we will refer to the lecture notes of the course Imperative Programming by Arnold Meijster, and to *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, a standard text about C. We will denote it as *Kernighan & Ritchie*.

Chapter 1

Linear data structures

In this chapter we present linear structures: stacks, queues and lists. As an application, we use lists in functions for recognizing and evaluating arithmetical expressions.

1.1 Stacks

A *stack* is a data structure in which we can insert and remove items. We use the function `push` for storing an item, and `pop` for removing an item. The *last in, first out* principle is valid here: the item removed by a `pop` is the item that was stored last. So after execution of `push(1); push(2); push(3);`, the function `pop` yields the number 3. Again `pop` yields 2, another time `pop` yields 1, and doing `pop` another time gives an error message, e.g. *stack empty*. A stack is sometimes called a LIFO-queue (*Last in, First out*).

This is a rather abstract description of the stack. It only indicates what you can do with a stack, not how to realize it with an implementation. How to make a stack? There are several ways. Now we work with an integer array. Later on, we present another way to realize a stack.

The idea is to implement `push` by placing an item (in this case an integer) at the first free position in the array of the stack, and `pop` by retrieving the integer from the highest nonempty position in the array. To realize this, we define a struct type for stacks, consisting of an integer array, an index to the first free position, and the size of the array. Structures have been discussed shortly in the lecture notes *Imperative Programming* 4.2.1. See also *Kernighan & Ritchie*, Ch. 6.

```
typedef struct Stack {
  int *array;
  int top;
  int size;
} Stack;
```

We have a function for creating an empty stack, with the initial size of the array as a parameter:

```
Stack newStack (int s) {
  Stack st;
  st.array = malloc(s*sizeof(int));
  assert(st.array != NULL);
  st.top = 0;
  st.size = s;
  return st;
}
```

Before giving the implementation of `push`, we anticipate the situation of overflow that occurs when we try to add an integer when the array is completely filled. We do not want to

overwrite (and hence lose) any integer in the stack, so we have to extend the array. For this, we use the following function, which doubles the size of the array while leaving the contents intact.

```
void doubleStackSize (Stack *stp) {
  int newSize = 2 * stp->size;
  stp->array = realloc(stp->array, newSize * sizeof(int));
  assert(stp->array != NULL);
  stp->size = newSize;
}
```

Recall that `stp->array` is shorthand for `(*stp).array`.

Now we can define `push`:

```
void push (int value, Stack *stp) {
  if (stp->top == stp->size) {
    doubleStackSize(stp);
  }
  stp->array[stp->top] = value;
  stp->top++;
}
```

Observe that the second parameter in `push` is a reference parameter: a pointer to a stack. When `push` is executed, this pointer points to the new stack. This is the procedural style of value transfer — see section A.3 for details. We might have followed the functional style, with the function prototype `Stack push (int value, Stack st)`. However, this would not work for `pop`, because it already has an output value, viz. the popped item. So using a reference parameter in `pop` is quite natural. We decide to follow the same approach in the definition of `push`.

When implementing `pop`, we have to deal with the case that the stack is empty, so there is nothing to pop. For that purpose, we define two functions:

```
int isEmptyStack (Stack st) {
  return (st.top == 0);
}

void stackEmptyError () {
  printf("stack empty\n");
  abort();
}
```

Now we can define `pop`:

```
int pop (Stack *stp) {
  if (isEmptyStack(*stp)) {
    stackEmptyError();
  }
  (stp->top)--;
  return (stp->array)[stp->top];
}
```

Note that we do not have to actually empty `(stp->array)[stp->top]`.

Finally, we define a function for freeing a stack:

```
void freeStack (Stack st) {
  free(st.array);
}
```

We inspect the time complexity of `push` and `pop`. It is quite obvious that `pop` requires a bounded number of computation steps: perform a simple test, give an error message or decrement a variable and return a value from an array. In other words: the time complexity of `pop` is in $\mathcal{O}(1)$.

The situation for `push` is somewhat more complicated. In most cases, the execution of `push` only requires a bounded number of computation steps, but this is not always the case. When the stack is full, `doubleStackSize` is executed, and this leads to the execution of `realloc` to double the memory allocation of the array. When the required memory is available next to the present memory location, this is done in $\mathcal{O}(1)$ time. But it may occur that the array has to move to another location in memory, and in that case $\mathcal{O}(\texttt{size})$ computation steps are required. We conclude: the time complexity of `push` is often in $\mathcal{O}(1)$ and sometimes in $\mathcal{O}(\texttt{size})$.

As an upper bound, we have no better than $\mathcal{O}(\texttt{size})$. This may look pretty bad, but often we are interested in *average* time complexity. How is that for `push`? Suppose we start with an array of length 1000, and we perform `push` 1 000 000 times. What is the average time complexity? Well, we will perform `doubleStackSize` 10 times to double the memory size of the array 10 times, to $2^{10} \cdot 1000 = 1\,024\,000$. In every doubling step, `realloc` is applied to the array, and this may involve copying the contents of the array to a new memory location. The number of computation steps required for this is of the order $1000 + 2000 + \cdots + 512\,000 = 1\,023\,000$, i.e. of the order 1 000 000. So on average $\mathcal{O}(1)$ per execution of `push`.

Conclusion: the *average* time complexity of `push` is in $\mathcal{O}(1)$, but for an individual execution of `push` there is a relatively small probability that it will take $\mathcal{O}(\texttt{size})$ steps.

1.2 Queues

We now consider another way to store and retrieve items, where the retrieved item is the item that was stored first (instead of last, as in a stack). We call this structure a *queue*. There are two functions: with `enqueue` we store an item at the back end of the queue, and with `dequeue` we return the item at the front end of the queue. So after performing

```
enqueue(1); enqueue(2); enqueue(3);
```

the call `dequeue` returns 1. Calling `dequeue` again yields 2, doing it another time yields 3, and when we try `dequeue` again we get an error message, e.g. `queue empty`. Sometimes a queue is called a FIFO-queue (*First in, First out*).

A queue can be implemented with an array, too. We define an appropriate type:

```
typedef struct Queue {
  int *array;
  int back;
  int front;
  int size;
} Queue;
```

Compare this with the type `Stack`: `top` is replaced by two fields, `back` and `front`, both acting as array indices. They are used to indicate the back end and the front end of the queue, as represented in the array. `back` points to the first free position, `front` to the position with the 'oldest' item when the queue is not empty, otherwise to the same position as `back`.

We define a function `newQueue`:

```
Queue newQueue (int s) {
  Queue q;
  q.array = malloc(s*sizeof(int));
  assert(q.array != NULL);
  q.back = 0;
  q.front = 0;
  q.size = s;
  return q;
}
```

The implementation of `enqueue` and `dequeue` is more involved than that of `push` and `pop`. Let us analyse the situation first. When filling queue `q`, `enqueue` will increment `q.back`; when emptying the queue with `dequeue`, `q.front` will be incremented, too. When `q.back` gets the value `q.size`, the end of `q.array` has been reached. But it is very well possible that several positions are free at the beginning of the array, because `dequeue` has been performed one or more times. To use these free positions, we let `q.back` jump back from `q.size` to `0`. With `q.front` we do the same. This is realized by computing *modulo* `q.size`. As a consequence, it may happen that `q.back` $<$ `q.front`. We call this a *split configuration*. See Figure 1.1.

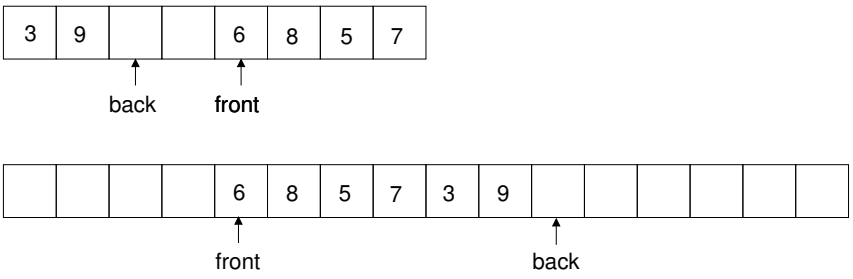

Figure 1.1: Above a queue of size 8 with split configuration. Below the result of applying doubleQueueSize from page 9.

How to check that the queue is empty? From the description of the meaning of `q.back` and `q.front` it follows that this is the case if `q.back` and `q.front` are equal. So we define

```
int isEmptyQueue (Queue q) {
  return (q.back == q.front);
}

void queueEmptyError() {
  printf("queue empty\n");
  abort();
}
```

And when do we know that the queue is full? We can see this when, after incrementing `q.back` directly after adding an item, `q.back` has become equal to `q.front`. Now it is time for direct action: for if we do nothing and look at a later moment at the queue, we see that `q.back` and `q.front` are equal, so we think wrongly that the queue is empty. We use the function `doubleQueueSize` to deal with a full queue.

The implementation of the functions `enqueue` and `dequeue`, based on the principles discussed above, is as follows:

```
void enqueue (int item, Queue *qp) {
  qp->array[qp->back] = item;
  /* assume that this spot is free */
  qp->back = (qp->back + 1) % qp->size;
  if (qp->back == qp->front) {
    doubleQueueSize(qp);
  }
}
```

```
int dequeue (Queue *qp) {
  int item;
  if (isEmptyQueue(*qp)) {
    queueEmptyError();
  }
  item = qp->array[qp->front];
  qp->front = (qp->front + 1) % qp->size;
  return item;
}
```

Observe that we use a queue pointer as reference parameter.

Now the implementation of `doubleQueueSize`. There is more to do than for the stack. In case of a split configuration, we must restructure the configuration. This is done by moving the items in the positions 0, 1, ..., `q.back - 1` to the new positions `oldSize`, `oldSize + 1`, ..., `oldSize + q.back - 1`, followed by an update of `q.back`. Here `oldSize` is the initial value of `q.size`. Observe that, after moving the items, the configuration is no longer split. See Figure 1.1.

```
void doubleQueueSize (Queue *qp) {
  int i;
  int oldSize = qp->size;
  qp->size = 2 * oldSize;
  qp->array = realloc(qp->array, qp->size * sizeof(int));
  assert(qp->array != NULL);
  for (i=0; i < qp->back ; i++) {
    /* eliminate the split configuration */
    qp->array[oldSize + i] = qp->array[i];
  }
  qp->back = qp->back + oldSize;  /* update qp.back */
}
```

Finally, a function for freeing a queue:

```
void freeQueue (Queue q) {
  free(q.array);
}
```

Thus we obtain an implementation of a queue.

As to the time complexity of `enqueue` and `dequeue`, the same remarks apply as for `push` and `pop`. So the *average* time complexity of `enqueue` is in $\mathcal{O}(1)$. For an individual execution of `enqueue` there is a relatively small probability that it will take $\mathcal{O}(\texttt{q.size})$ steps. The time complexity of `dequeue` is in $\mathcal{O}(1)$.

1.2.1 Priority queues

The *priority queue* is a different type of queue. Every item in it is associated with a number that indicates its priority. When removing an item from the priority queue with the function `removeMax`, the element with the highest priority value is chosen, not the item that was stored first. Later on in these lecture notes, we will present two implementations of the priority queue. The first implementation is rather straightforward and uses ordered lists (see Section 1.3.4). The time complexity of `enqueue` then becomes $\mathcal{O}(n)$, where n is the length (i.e. number of elements) of the priority queue. This is not optimal. The second implementation uses *heaps*, a kind of binary trees (see Section 2.3). In this implementation, the time complexity of `enqueue` is $\mathcal{O}(\log(n))$, which is definitely better than $\mathcal{O}(n)$, and in a certain sense optimal.

1.2.2 Stacks and queues as abstract data structures

We have seen how stacks and queues can be implemented with help of arrays. We might be tempted to think that stacks and queues are nothing else but arrays that we manipulate in a certain way. However, that would ignore the *essence* of stacks and queues. What is the essence of a stack? It is contained in the behaviour of the functions `push` and `pop`. Let us try to describe this essence in the following definition.

A stack is a dynamic data structure that can contain items.
`push(item)` adds an item to the stack.
`pop()` removes and returns the item in the stack that was added last (provided the stack is not empty).

This is an example of an *abstract data structure (ADT)*. It talks only about the functional behaviour of the stack, abstracting away from implementation issues. Besides the essential functions (`push` and `pop` in the case of the stack) an ADT may contain auxiliary functions, e.g. `isEmptyStack`.

Not surprisingly, the definition of the queue closely resembles that of the stack:

A queue is a dynamic data structure that can contain items.
`enqueue(item)` adds an item to the queue.
`dequeue()` removes and returns the item in the queue that was added first (provided the queue is not empty).

1.2.3 Making a queue from two stacks

We will now do an exercise in elementary programming on the level of abstract data types. Is it possible to make a queue from two stacks `stack0` and `stack1`?

Yes, it is. The idea is as follows. The function `enqueue` is simply realized by `stack0`. But how to implement `dequeue`, i.e. how to obtain the first item from the queue? With `pop(stack0)` we only obtain the last item from `stack0`, while we need the first item. Now it is time to use `stack1`. We take all items in `stack0` and put them one after the other in `stack1`: this is done with `push(pop(&stack0),&stack1)` repeatedly, until `stack0` is empty. Now the first item in the queue is the last item in `stack1`, and it can be obtained by `pop(stack1)`.

This can be worked out as follows:

```
typedef struct Queue {
  Stack stack0;
  Stack stack1;
} Queue;

void enqueue (int value, Queue *qp) {
  push(value,&(qp->stack0)); /* push on stack 0 */
}

int dequeue(Queue *qp) {
  if (isEmptyStack(qp->stack1)) {
    while (! isEmptyStack(qp->stack0) ) {
      /* transfer contents of stack0 to stack1 */
      push(pop(&(qp->stack0)),&(qp->stack1));
    }
  }
  return pop(&(qp->stack1));
}
```

The time complexity of `dequeue` is $\mathcal{O}(n)$ in the worst case (n is the number of elements in the queue). This happens when `stack1` is empty, and the n items have to be transferred from `stack0` to `stack1`. However, the *average* time complexity of `dequeue` is $\mathcal{O}(1)$: for every item is first placed in `stack0`, once transferred to `stack1` and in the end removed from `stack1`.

We leave it as an exercise to make a stack from two queues.

1.3 Lists

Stacks and queues are data structures with limited access: adding and removing only happens at the front or at the back side. We will now present another way to order items linearly that offers more access: the (linked) *list*. A list contains *nodes*: a node not only contains an item, but also a pointer to the next node. For now we assume that all items are of type `int`. We therefore define

```
typedef struct ListNode* List;

struct ListNode {
  int item;
  List next;
};
```

In the first line, we define the type name `List` for the type `struct listNode*`. Then we define `struct listNode` as a structure with a field `item` and a field `next`.

Observe that this is a definition with *mutual recursion*: `List` is used in the definition of `ListNode`, `ListNode` in the definition of `List`. An alternative way of defining lists uses 'plain' recursion and goes as follows:

```
typedef struct ListNode {
  int item;
  struct ListNode* next;
}* List;
```

In these lecture notes, we shall use the first method with mutual recursion when defining recursive types, as it is more readable.

Now we can define the functions `newEmptyList` that creates a new list, and `addItem` that adds an item (in this case a number) in a new node at the beginning of the list.

```
List newEmptyList() {
  return NULL;
}

List addItem(int n, List li) {
  List newList = malloc(sizeof(struct ListNode));
  assert(newList!=NULL);
  newList->item = n;
  newList->next = li;
  return newList;
}
```

We can make a list with one node containing the value 3:

```
li = addItem(3,newEmptyList());
```

Now `li` points to the node that contains 3 and the pointer `NULL`. We may add a node to the beginning of `li`:

```
li = addItem(2,li);
```

We now have a list with two nodes: the first node contains 2 and a pointer to the second node, which contains 3 and the pointer `NULL`. In this way, we may e.g. make the list displayed in Figure 1.2.

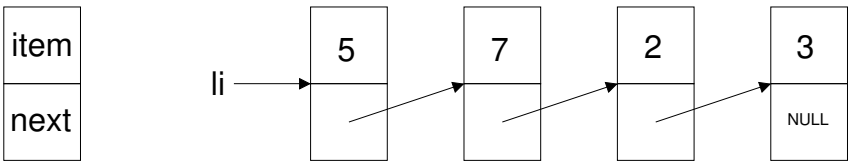

Figure 1.2: Graphical representation of a list li with 4 nodes.

Then we define the function `firstItem` that returns the item from the first item in the list, and an error message if the list is empty.

```
void listEmptyError() {
  printf("list empty\n");
  abort();
}

int firstItem(List li) {
  if (li == NULL) {
    listEmptyError();
  }
  return li->item;
}
```

We define the function `removeFirstNode`. It returns its argument without the first item.

```
List removeFirstNode(List li) {
  List returnList;
  if (li == NULL) {
    listEmptyError();
  }
  returnList = li->next;
  free(li);
  return returnList;
}
```

Observe that we use the auxiliary variable `returnList`, which enables us to free the memory used for the first node before we return the rest of the list.

Finally we define the function `freeList`, which frees the memory used for a list.

```
void freeList(List li) {
  List li1;
  while (li != NULL) {
    li1 = li->next;
    free(li);
    li = li1;
  }
  return;
}
```

The following toy example shows how our little stack library can be used.

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

int main(int argc, char *argv[]) {
  Stack stack;
  stack = newStack(2);
  push(42, &stack);
  push(17, &stack);
  push(23, &stack);
  pop(&stack);
  printf("%d\n", pop(&stack));
  freeStack(stack);
  return 0;
}
```

Question: Can you predict the output of the program above?

1.3.1 Stacks and queues implemented with lists

The list function `addItem` closely resembles the stack function `push`, and the stack function `pop` can be seen as a combination of the list functions `firstItem` and `removeFirstNode`. As a consequence, the implementation of a stack with a list is quite straightforward.

```
typedef struct Stack {
  List list;
} Stack;
```

```
Stack newStack () {
  Stack st;
  st.list = newEmptyList();
  return st;
}

int isEmptyStack (Stack st) {
  return isEmptyList(st.list);
}

void stackEmptyError() {
  printf("stack empty\n");
  exit(0);
}

void push (int n, Stack *stp) {
  stp->list = addItem(n,stp->list);
  return;
}

int pop (Stack *stp) {
  int n;
  if (isEmptyStack(*stp)) {
    stackEmptyError();
  }
  n = firstItem(stp->list);
  stp->list = removeFirstNode(stp->list);
  return n;
}

void freeStack (Stack st) {
  freeList(st.list);
}
```

The implementation of a queue with a list is a bit more involved, because adding an item and removing an item takes place at different ends of the list. Therefore, we use a field `lastNode` to point to the last node in the list. It is used in the definition of the function `enqueue` that adds a new node at the end of the list.

```
typedef struct Queue {
  List list;
  List lastNode;
} Queue;

Queue newEmptyQueue() {
  Queue q;
  q.list = newEmptyList();
  q.lastNode = NULL;
  return q;
}

int isEmptyQueue(Queue q) {
  return isEmptyList(q.list);
}
```

```
void queueEmptyError() {
  printf("queue empty\n");
  exit(0);
}

void enqueue(int n, Queue *qp) {
  if (isEmptyList(qp->list)) {
    qp->list = addItem(n,NULL);
    qp->lastNode = qp->list;
  } else { /* *qp not empty, so qp->lastNode != NULL */
    (qp->lastNode)->next = addItem(n,NULL);
    qp->lastNode = (qp->lastNode)->next;
  }
  return;
}
```

The definition of `dequeue` resembles that of `pop` for the stack. However, we should not forget here to give the `lastNode` field the value `NULL` as soon as the list is empty.

```
int dequeue(Queue *qp) {
  int n;
  if (isEmptyQueue(*qp)) {
    queueEmptyError();
  }
  n = firstItem(qp->list);
  qp->list = removeFirstNode(qp->list);
  if (isEmptyList(qp->list))  {
    qp->lastNode = NULL;
  }
  return n;
}

void freeQueue (Queue q) {
  freeList(q.list);
}
```

1.3.2 Traversing a list

When you have a list, you may want to traverse it systematically and to perform some action at each node. Suppose we have a function `visit` that should be applied to each node. For example this might be a function which prints the item in the node.

We can then traverse the list with the following iterative function:

```
void visitList(List li) {
  while (li != NULL) {
    visit(li);
    li = li->next;
  }
}
```

Traversing a list can also be done recursively:

```
void visitListRec(List li) {
```

```
  if (li == NULL) {
    return;
  }
  visit(li);
  visitListRec(li->next);
}
```

1.3.3 Operations on an arbitrary position in a list

Up to now, all operations on lists act on the beginning of the list, with the exception of `enqueue` that adds a node at the end of a list. We will now define some functions that act on an *arbitrary* position in the list. This is accomplished by walking through the list until we have reached the node where the action has to take place. Like for arrays, we start counting positions at 0.

We begin with generalizing `firstItem` to a function `itemAtPos` that returns the item at position p. Since we start counting at 0, `itemAtPos(li,0)` will do the same as `firstItem(li)`. When $p > 0$, we solve the problem recursively by decrementing p by 1 and making a step in the list, from `li` to `li->next`. If the indicated position is not in the list, an error message follows via the function `listTooShort`.

```
void listTooShort() {
  printf("List too short\n");
  abort();
}

int itemAtPos(List li, int p) {
  if (li == NULL) {
    listTooShort();
  }
  if (p == 0) {
    return firstItem(li);
  } else {
    return itemAtPos(li->next,p-1);
  }
}
```

Along the same line we generalize `addItem` to `addItemAtPos`. The basic cases are dealt with in a different order, for now `li==NULL` is no problem when `p==0`.

```
List addItemAtPos(List li, int n, int p) {
  if (p == 0) {
    return addItem(n,li);
  }
  if (li == NULL) {
    listTooShort();
  }
  li->next = addItemAtPos(li->next,n,p-1);
  return li;
}
```

Of course, `itemAtPos` and `addItemAtPos` can be defined without recursion. For `itemAtPos` this is an exercise, but for `addItemAtPos` it is a bit harder and we do it here.

Suppose we want to add an item at position 3. It seems straightforward to walk 3 steps from the beginning of the list to the node on position 3, and then to create a new node with

`addItem`. But now we have a problem: how are we to tell the node on position 2 that there is a new node at position 3? We solve this problem by walking 2 instead of 3 steps, so that we end up in the node on position 2 where we can make the `next` field point to the new node on position 3. See Figure 1.3 and the C code below.

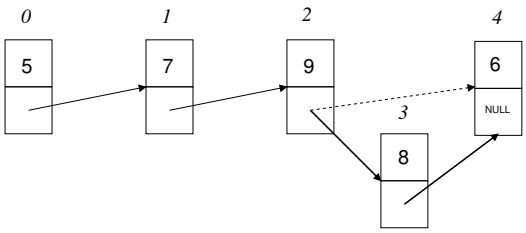

Figure 1.3: Adding a node at position 3. Italic numbers indicate the new positions.

```
List addItemAtPosIt(List li, int n, int p) {
  List li1;
  if (p == 0) {
    return addItem(n,li);
  }
  li1 = li;
  while (li1 != NULL && p > 1) {
    li1 = li1->next;
    p--;
  }
  if (li1 == NULL) {
    listTooShort();
  }
  /* now li1 points to the node on position p-1 */
  li1->next = addItem(n,li1->next);
  return li;
}
```

There is another way to solve the problem mentioned above. Just walk up to node w on position p, replace item m in w by n, and now create a new node for item m and place it between w and its successor (if any). This works, but it is slightly less elegant as it generalizes not well to the removal of an item on a given position (which will be asked in an exercise).

Finally we consider the removal of a node with a given item. More precisely: of the first node containing a given item, if it exists. With recursion this goes as follows:

```
List removeItem(List li, int n) {
  if (li == NULL) {
    return li;
  }
  if (li->item == n) {
    return removeFirstNode(li);
  }
  li->next = removeItem(li->next,n);
  return li;
}
```

We follow the same strategy as for `addItemAtPos` in the conversion to an iterative definition: when walking through the list, stop just before the position where action is required. This is accomplished by looking ahead in the condition of the `while` statement.

```
List removeItemIt(List li, int n) {
  List li1;
  if (li == NULL) {
    return li;
  }
  if (li->item == n) {
    return removeFirstNode(li);
  }
  li1 = li;
  while (li1->next != NULL && (li1->next)->item != n) {
    li1 = li1->next;
  }
  if (li1->next!=NULL) {  /* so (li1->next)->item == n */
    li1->next = removeFirstNode(li1->next);
  }
  return li;
}
```

1.3.4 Ordered lists

An *ordered list* is a list where the items in the nodes are ordered. With an ordered list we can implement the priority queue, with the operations `enqueue` and `removeMax`. When we choose a decreasing order, `removeMax` can be implemented with `removeFirstNode`. For the implementation of `enqueue` a new function `insertInOrder` is required that traverses an ordered list and inserts the item at the right position. The definition of `insertInOrder` is left as an exercise.

1.4 Application: recognize and evaluate arithmetical expressions

As an application of lists, we will now focus on an important task for computers: the processing of expressions. We restrict ourselves to rather simple mathematical expressions: formulas and equations with natural numbers, identifiers and arithmetical operations.

1.4.1 Grammars

Expressions are generated by a *grammar*, i.e. a collection of rewrite rules. Such rules are written down in the so-called Backus-Naur form, short BNF. As an example, here is a grammar for the usual notation of integers:

⟨*integer*⟩ ::= '0' | ['-'] ⟨*posint*⟩ .

⟨*posint*⟩ ::= ⟨*posdigit*⟩ { ⟨*digit*⟩ } .

⟨*posdigit*⟩ ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .

⟨*digit*⟩ ::= '0' | ⟨*posdigit*⟩ .

We use the following notation for rewrite rules:

- every rewrite rule has the form ⟨*nonterminal*⟩ ::= ... . ;

- terminals (the symbols produced by the rules) are written between single quotation marks;
- the vertical bar `|` is used as a separation mark between alternatives;
- text between square brackets `[ ]` is optional;
- text between braces `{ }` is to be repeated zero or more times.

We may paraphrase the contents of the grammar as follows:

- An ⟨*integer*⟩ is 0, or a ⟨*posint*⟩ possibly preceded by - (the minus sign).
- A ⟨*posint*⟩ is a ⟨*posdigit*⟩ followed by zero or more times a ⟨*digit*⟩.
- A ⟨*posdigit*⟩ is one of the digits 1 up to 9.
- A ⟨*digit*⟩ is 0 or a ⟨*posdigit*⟩.

The expressions generated by the grammar are called the *productions* of the grammar. Observe that this grammar indeed produces the usual notation for integers: 0, 5, 2324, –718 are productions of the grammar, while –0, +3, 042 are not.

We give a linguistic example. A strongly simplified grammar for English can be written as follows:

⟨*sentence*⟩ ::= ⟨*subject*⟩ ⟨*verb*⟩ `[` ⟨*object*⟩ `]` .

⟨*subject*⟩ ::= ⟨*thing*⟩ .

⟨*object*⟩ ::= ⟨*thing*⟩ .

⟨*thing*⟩ ::= `[` '`the`' | '`a`' | '`an`' `]` `[` ⟨*adjective*⟩ `]` ⟨*noun*⟩ .

⟨*verb*⟩ ::= '`sees`' | '`speaks`' | '`gives`' .

⟨*noun*⟩ ::= '`man`' | '`woman`' | '`hand`' | '`chair`' | '`Santa Claus`' .

⟨*adjective*⟩ ::= '`big`' | '`small`' | '`old`' | '`young`' .

This grammar produces e.g. the sentences *a young woman speaks* and *old Santa Claus gives a big hand*, but also nonsense sentences like *chair sees an man*.

1.4.2 Arithmetical expressions

Now we will consider an important class of expressions: the arithmetical expressions. They are built from natural numbers, identifiers, the arithmetical operators '`+`', '`-`', '`*`', '`/`', and the parentheses '`(`' and '`)`'. First we present the grammar for natural numbers (i.e. non-negative integers). It is a slight variant of the grammar for integers given above.

⟨*nat*⟩ ::= '`0`' | ⟨*posdigit*⟩ `{` ⟨*digit*⟩ `}` .

⟨*posdigit*⟩ ::= '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`' .

⟨*digit*⟩ ::= '`0`' | ⟨*posdigit*⟩ .

An identifier is a nonempty sequence of letters and digits starting with a letter. We can describe them with a grammar:

⟨*identifier*⟩ ::= ⟨*letter*⟩ `{` ⟨*letter*⟩ | ⟨*digit*⟩ `}` .

⟨*letter*⟩ ::= '`A`' | . . . | '`Z`' | '`a`' | . . . | '`z`' .

Now we can give the following grammar for arithmetical expressions:

⟨*expression*⟩ ::= ⟨*nat*⟩
| ⟨*identifier*⟩
| '`(`' ⟨*expression*⟩ '`)`'
| ⟨*expression*⟩ '`+`' ⟨*expression*⟩
| ⟨*expression*⟩ '`-`' ⟨*expression*⟩
| ⟨*expression*⟩ '`*`' ⟨*expression*⟩
| ⟨*expression*⟩ '`/`' ⟨*expression*⟩ .

This grammar produces arithmetical expressions. However, it has an important disadvantage: it is *ambiguous*. This means that some expressions can be produced in more than one way. Consider e.g. the production `5+3*2`: this expression can be obtained by conjoining the expressions `5+3` and `2` using `*`, but also by conjoining `5` and `3*2` using `+`. If we were only interested in the production of expressions, this would not be a problem, but we want more than that. We want to use the grammar as an instrument for the evaluation of expressions. For the evaluation we must know how the expression is constructed. If that can be done in more than one way, there is also more than one way to compute the value of the expression and we fail to have a unique evaluation result.

In order to disambiguate the grammar, we have a look at the priority rules for arithmetic.

- `*` and `/` have priority over + and -, so `2*3+4/5 = (2*3) + (4/5)`;
- `*` and `/` are executed from left to right,[1] so `2/3*4/5 = ((2/3)*4)/5`;
- `+` and `-` are executed from left to right, so `2-3+4-5 = ((2-3)+4)-5`.

The ambiguity can be eliminated by adding two syntactical categories: terms and factors. We then get the following grammar

⟨*expression*⟩ ::= ⟨*term*⟩ { '`+`' ⟨*term*⟩ | '`-`' ⟨*term*⟩ } .

⟨*term*⟩ ::= ⟨*factor*⟩ { '`*`' ⟨*factor*⟩ | '`/`' ⟨*factor*⟩ } .

⟨*factor*⟩ ::= ⟨*nat*⟩ | ⟨*identifier*⟩ | '`(`' ⟨*expression*⟩ '`)`' .

In words:

- an expression consists of a non-empty sequence of terms separated by `+` or `-` ;
- a term consists of a non-empty sequence of factors separated by `*` or `/`;
- a factor is a natural number, an identifier or an expression between parentheses.

This grammar is not ambiguous and respects the priority rules of arithmetic. As an example, we consider the expression `2*3+4/5`. It can be obtained by conjoining the terms `2*3` and `4/5` with `+`, which is in line with the priority rules. And is it also possible to obtain `2*3+4/5` by conjoining the factors `2` and `3+4/5` with `*`? No, that is not possible, for `3+4/5` is not a factor (it only becomes one when it is put between parentheses).

1.4.3 The interpretation of expressions

We have seen how a grammar can be used to generate expressions. Expressions are used to express meaning. How can an automated system determine the meaning of an expression? In the rest of the chapter, we will present a method to do this. It uses the grammar to reconstruct the structure of an expression generated by the grammar, and with this structure we are able to find the meaning of the expression. As an example, we will focus on the grammar for arithmetical expressions given above.

[1] This may be different from what you learned at school, where often `*` is given priority over `/`.

We begin with a rather simple problem, viz. the *recognition* of arithmetical expressions: given a sequence of characters, check whether it is an expression generated by the grammar. A first step is *scanning*. That is: splitting the sequence of characters in the parts of an expression (non-negative numbers, identifiers and symbols), and putting these parts in a list called the *token list*. Scanning is also called *lexing*, and the program that does it can be called a *scanner* or *lexer*.

The second step is *parsing* to check whether and how the list of parts can be produced by the grammar. When the parser succeeds, the expression is recognized as a production of the grammar.

Finally we will *evaluate* expressions, i.e. compute their value. That will only succeed for arithmetical expressions not containing identifiers: the *numerical* expressions. The evaluator is obtained from the recognizer by adding functionality.

This will all be worked out in the rest of the chapter. The code described here is available on Themis and can be used for the practical assignments.

1.4.4 Scanning

We now present the scanner that can be found in the files `scanner.h` and `scanner.c`. It can be used for the practical assignments.

The header file `scanner.h` contains the following definitions:

```
#define MAXINPUT 100  /* maximal length of the input */
#define MAXIDENT 10   /* maximal length of an identifier */

typedef enum TokenType {
  Number,
  Identifier,
  Symbol
} TokenType;

typedef union Token {
  int number;
  char *identifier;
  char symbol;
} Token;

typedef struct ListNode *List;

typedef struct ListNode {
  TokenType tt;
  Token t;
  List next;
} ListNode;

char *readInput();
List tokenList(char *array);
int valueNumber(List *lp, double *wp);
void printList(List l);
void freeTokenList(List l);
void scanExpressions();
```

`MAXINPUT` and `MAXIDENT` are the maximum length of the input and identifiers. These maxima are not absolute, as we shall see later in the functions `readInput` and `matchIdentifier`.

The first `typedef` defines the type `TokenType`. This is an enumeration type containing three constants: `Number` (with value 0), `Identifier` (with value 1), and `Symbol` (with value

2). More about enumeration types in *Kernighan & Ritchie* 2.3, p. 39.

The second `typedef` defines the type `Token`. That is a union type: an object of type `token` is an `int`, a pointer to `char` or a `char`. See *Kernighan & Ritchie* 6.8, p. 147.

The third and fourth `typedef` are not new for us: they form the definition of a list, in this case the token list. The nodes in a token list contain three fields: `tt` indicates the type of the token in `t`, and `next` contains as usual a pointer to the next node. See Figure 1.4.

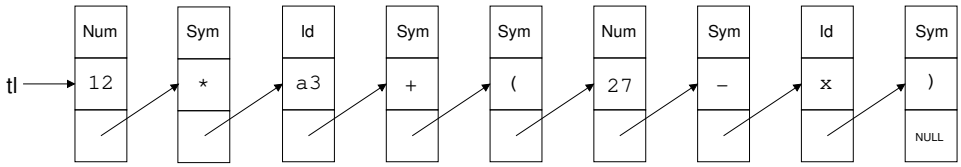

Figure 1.4: Token list for the string `12*a3+(27-x)`.

The definitions of the functions declared in `scanner.h` are in `scanner.c`, which we investigate now. We also show the `#include` lines here, with comments to clarify which function comes from which library.

```
#include <stdio.h>  /* getchar, printf */
#include <stdlib.h> /* NULL, malloc, free */
#include <string.h> /* strcpy */
#include <ctype.h>  /* isspace, isdigit, isalpha, isalnum */
#include <assert.h> /* assert */
#include "scanner.h"

char *readInput() {
  int strLen = MAXINPUT;
  int c = getchar();
  int i = 0;
  char *s = malloc((strLen+1)*sizeof(char));
  assert(s != NULL);
  while (c != '\n') {
    s[i] = c;
    i++;
    if (i >= strLen) {
      /* s is not large enough, double its length */
      strLen = 2*strLen;
      s = realloc(s,(strLen+1)*sizeof(char));
      assert(s != NULL);
    }
    c = getchar();
  }
  s[i] = '\0';
  return s;
}
```

`readInput()` reads the characters from the input using the function `getchar` defined in `stdio.h` (see *Kernighan & Ritchie* B1.4, p. 247). Each character is put in the string `s`, unless `\n` (newline) is read: in that case reading stops and the *null character* `\0` is added to `s` to indicate the end of the string. Furthermore the length of string `s` is doubled when necessary.

Now that we can read in a string, how do we obtain a list of tokens from it? First we define three auxiliary functions for reading numbers, identifiers and symbols.

```
int matchNumber(char *ar, int *ip) {
```

```
  int n = 0;
  while (isdigit(ar[*ip])) {
    n = 10*n + (ar[*ip] - '0');
    (*ip)++;
  }
  return n;
}

char matchCharacter(char *ar, int *ip) {
  char s = ar[*ip];
  (*ip)++;
  return s;
}

char *matchIdentifier(char *ar, int *ip) {
  int j = 0;
  int strLen = MAXIDENT;
  char *s = malloc((strLen+1)*sizeof(char));
  assert(s != NULL);
  while (isalnum(ar[*ip+j])) {
    s[j] = ar[*ip+j];
    j++;
    if (j >= strLen) {
      /* s is not large enough, double the length */
      strLen = 2*strLen;
      s = realloc(s,(strLen+1)*sizeof(char));
      assert(s != NULL);
    }
  }
  s[j] = '\0';
  *ip = *ip + j;
  return s;
}
```

These functions each have two parameters: the array from which they read, and a pointer to a position in the array. This last parameter is a *call by reference* that is not only used for the initial position in the array to be read, but also to point to the first unread position in the array after the function has ended. See the lecture notes *Imperative Programming* 4.1.6.

For the computation of the numerical value of a sequence of digits we use the fact that in the ASCII table the digits have consecutive codes, so the numerical value of digit `c` equals `c-'0'`. In the function `matchIdentifier` the length of the string `s` is doubled when necessary, as in `readInput`.

Now we define a function to construct the nodes for the token list.

```
List newNode(char *ar, int *ip) {
  /* precondition: !isspace(a[*ip]) */
  List node = malloc(sizeof(struct ListNode));
  assert(node!=NULL);
  node->next = NULL;
  if (isdigit(ar[*ip])) {
    /* we see a digit, so a number starts here */
    node->tt = Number;
    (node->t).number = matchNumber(ar,ip);
    return node;
  }
```

```
  if (isalpha(ar[*ip])) {
    /* we see a letter, so an identifier starts here */
    node->tt = Identifier;
    (node->t).identifier = matchIdentifier(ar,ip);
    return node;
  }
  /* no space, no number, no identifier: we call it a symbol */
  node->tt = Symbol;
  (node->t).symbol = matchCharacter(ar,ip);
  return node;
}
```

Like the previous three functions, the function `newNode` has two parameters: the array `ar` to read from, and a pointer `ip` to a position in the array. The precondition is that `ar[*ip]` is not a space: so it is a digit, a letter, or another character. In the last case, we call it a `Symbol`. We use the functions `isdigit` and `isalpha` from `ctype.h` to determine which case applies. Based on its outcome, one of the previous three functions is called. The result is put in a new node.

Now we can compose the token list.

```
List tokenList(char *ar) {
  List lastNode = NULL;
  List node = NULL;
  List tl = NULL;
  int i = 0;
  int length = strlen(ar);
  while (i < length) {
    if (isspace(ar[i])) { /* spaces are skipped */
      i++;
    } else {
      node = newNode(ar,&i);
      if (lastNode == NULL) {
        /* there is no list yet; start it with node */
        tl = node;
      } else {
        /* there is already a list; add node at the end */
        (lastNode)->next = node;
      }
      lastNode = node;
    }
  }
  return tl;
}
```

The function `tokenList` processes a string (a sequence of characters) and builds a list of the tokens in the string. Spaces are considered as separation marks between the tokens and are not included in the token list. Whenever a non-space is found, `newNode` creates a new node `new`. `new` is to be put at the end of the list in construction: we use `lastNode` to know the end of the list.

Observe that `newNode` has the pointer `&i` to index `i` as second argument. The content of this pointer is adapted by `newNode`, so that reading in `ar` can continue.

We also have two auxiliary functions for printing and freeing the token list.

```
void printList(List li) {
  while (li != NULL) {
```

```
    switch (li->tt) { /* distinguish by TokenType */
    case Number:
      printf("%d ",(li->t).number);
      break;
    case Identifier:
      printf("%s ",(li->t).identifier);
      break;
    case Symbol:
      printf("%c ",(li->t).symbol);
      break;
    }
    li = li->next;
  }
  printf("\n");
}
```

The function `printList` prints the tokens of a token list, separated by spaces. We use a `switch` statement to distinguish the three types of tokens. Observe how a token from a node is addressed: by `(li->t).number` when it is a number, by `(li->t).identifier` when it is an identifier and by `(li->t).symbol` when it is a symbol. Here `li->t` is the token field of the node pointed to by `li`, and the addition `.number` (or `.identifier` or `.symbol`) is required because the type of `token` is a union type.

```
void freeTokenList(List li) {
  if (li == NULL) {
    return;
  }
  if (li->tt == Identifier) {
    free((li->t).identifier);
  }
  freeTokenList(li->next);
  free(li);
}
```

The function `freeTokenList` frees the memory that is used by a token list. Observe that it is a recursive function that works backwards — we first free `li->next` and then `li` itself. Moreover, whenever we encounter a node that contains an identifier as token, we first free the memory used for that identifier.

1.4.5 Recognition

With the function `tokenList` we can thus convert a sequence of characters into a token list. Now we will determine whether the token list can be generated by the grammar for arithmetical expressions. This recognizer is in the files `recognizeExp.h` and `recognizeExp.c`. The file `recognizeExp.h` contains the following function declarations:

```
int acceptNumber(List *lp);
int acceptIdentifier(List *lp);
int acceptCharacter(List *lp, char c);
int acceptExpression(List *lp);
void recognizeExpressions();
```

These functions are defined in `recognizeExp.c` together with some other functions.

```
int acceptNumber(List *lp) {
```

```
  if (*lp != NULL && (*lp)->tt == Number) {
    *lp = (*lp)->next;
    return 1;
  }
  return 0;
}

int acceptIdentifier(List *lp) {
  if (*lp != NULL && (*lp)->tt == Identifier ) {
    *lp = (*lp)->next;
    return 1;
  }
  return 0;
}

int acceptCharacter(List *lp, char c) {
  if (*lp != NULL && (*lp)->tt == Symbol
                  && ((*lp)->t).symbol == c ) {
    *lp = (*lp)->next;
    return 1;
  }
  return 0;
}
```

The functions `acceptNumber`, `acceptIdentifier` and `acceptCharacter` have as (first) argument a pointer to a token list: observe that this is a *call by reference*. Furthermore `acceptCharacter` has a character as second argument. The functions check whether the first token in the token list is a natural number, an identifier or the character that was given as the second argument. When this is the case, the function in question yields the value 1 and moves the pointer to the next item in the token list. Otherwise the value 0 is returned and the pointer remains unchanged.

Now we can recognize expressions. The idea is the same as for the previous accept functions: inspect the tokens in the token list one by one and check whether they correspond to the definition of ⟨*expression*⟩ according to the grammar. A close look at the grammar we defined on page 20 shows us that:

> the rewrite rule for ⟨*expression*⟩ uses ⟨*term*⟩,
>
> the rewrite rule for ⟨*term*⟩ uses ⟨*factor*⟩,
>
> the rewrite rule for ⟨*factor*⟩ uses ⟨*expression*⟩.

This is an example of *mutual recursion*. Fortunately, C can deal with mutually recursive definitions. We shall give mutually recursive definitions of the functions `acceptExpression`, `acceptTerm` and `acceptFactor`. This technique of using mutually recursive functions for the parsing and recognition of expressions is called *recursive descent*.

We begin with `acceptFactor`. It uses `acceptExpression` which will be defined later. In order to keep the C compiler happy, we have to declare `acceptExpression` before we use it in the definition of `acceptFactor`. This declaration is in the header file `recognizeExp.h`.

The argument of the function `acceptFactor` is a pointer to a token list. It is checked whether the token list has an initial segment that can be recognized as a factor, as defined by the grammar. We repeat the rewrite rule for ⟨*factor*⟩:

⟨*factor*⟩ ::= ⟨*nat*⟩ | ⟨*identifier*⟩ | '(' ⟨*expression*⟩ ')' .

```
int acceptFactor(List *lp) {
```

```
  return
    (  acceptNumber(lp)
    || acceptIdentifier(lp)
    || (  acceptCharacter(lp,'(')
       && acceptExpression(lp)
       && acceptCharacter(lp,')')
       )
    );
}
```

So `acceptFactor` yields 1 whenever a production of ⟨*factor*⟩ has been identified as initial segment of the token list, and the pointer points to the first item in the remaining token list. Otherwise `acceptFactor` yields the value 0.

Observe that we use the function `acceptExpression` in the definition of `acceptFactor`. The function `acceptExpression` is not yet *defined*, but it suffices that it has been *declared* in `recognizeExp.h`.

```
int acceptTerm(List *lp) {
  if (!acceptFactor(lp)) {
    return 0;  /* no first factor at the start, so it cannot be
       ➥  a term */
  }
  while (acceptCharacter(lp,'*') || acceptCharacter(lp,'/')) {
    if (!acceptFactor(lp)) {
      return 0;
    }
  }   /* no * or /, so we reached the end of the term */
  return 1;
}
```

The function `acceptTerm` recognizes the productions of ⟨*term*⟩, with the rewrite rule

⟨*term*⟩ ::= ⟨*factor*⟩ `{` '`*`' ⟨*factor*⟩ | '`/`' ⟨*factor*⟩ `}` .

`acceptTerm` does the following:

- when we do not see a factor, it cannot be a term and we return 0;
- when we see a factor, we check whether we see '∗' or '/'; when it is not followed by a factor we return 0, otherwise we repeat this step;
- when we no longer see '∗' or '/', we know that we reached the end of the term and return 1.

Conclusion: `acceptTerm` yields 1 when a production of ⟨*term*⟩ is found as initial segment of the token list, and the pointer points to the first item in the remainder of the token list. Otherwise `acceptTerm` yields the value 0.

```
int acceptExpression(List *lp) {
  if (!acceptTerm(lp)) {
    return 0;
  }
  while (acceptCharacter(lp,'+') || acceptCharacter(lp,'-')) {
    if (!acceptTerm(lp)) {
      return 0;
    }
  } /* no + or -, so we reached the end of the expression */
```

```
  return 1;
}
```

The function `acceptExpression` recognizes the productions of ⟨*expression*⟩, with the rewrite rule

⟨*expression*⟩ ::= ⟨*term*⟩ { '+' ⟨*term*⟩ | '-' ⟨*term*⟩ } .

The working of `acceptExpression` closely resembles that of `acceptTerm`.

We demonstrate the recognizer with help of the function `recognizeExpressions`:

```
void recognizeExpressions() {
  char *ar;
  List tl, tl1;
  printf("give an expression: ");
  ar = readInput();
  while (ar[0] != '!') {
    tl = tokenList(ar);
    printf("the token list is ");
    printList(tl);
    tl1 = tl;
    if (acceptExpression(&tl1) && tl1 == NULL) {
      printf("this is an expression\n");
    } else {
      printf("this is not an expression\n");
    }
    free(ar);
    freeTokenList(tl);
    printf("\ngive an expression: ");
    ar = readInput();
  }
  free(ar);
  printf("good bye\n");
}
```

The function `recognizeExpressions` repeatedly asks for an expression, reads the input, transforms it into a token list, prints it and checks whether the token list represents an expression. It stops when the input starts with '!'.

Observe that we use two variables `tl` and `tl1` of type `List`. `tl` refers to the first node of the token list, and this variable is passed as parameter to `printList`. However, `acceptExpression` does not receive `&tl` as argument, but `&tl1` instead, which is a copy of `tl`. Why?

Recall that the parameter in `acceptExpression` is a *call by reference*: its value can be modified by the function. If we would pass `&tl` instead of `&tl1` as a parameter, then we would no longer have access to the begin of the token list, so we would be unable to free the memory used for the token list.

Also observe that, for recognizing an expression, we not only have to check whether `acceptExpression` holds, but also whether we have reached the end of the token list. That is why the condition `tl1 == NULL` is added to the condition in the `if` statement.

A session with the function `recognizeExpressions` may look as follows:

```
give an expression: 2+3*4-5
the token list is 2 + 3 * 4 - 5
this is an expression

give an expression: -3/7
```

```
the token list is - 3 / 7
this is not an expression

give an expression: 0 - 3/7
the token list is 0 - 3 / 7
this is an expression

give an expression: 3*x
the token list is 3 * x
this is an expression

give an expression: 3**x
the token list is 3 * * x
this is not an expression

give an expression: !
good bye
```

1.4.6 Evaluation

Now we go one step further: not only check whether a sequence of characters is an expression, but also evaluate it (i.e. determine its value). Of course, evaluation is only possible when the sequence of characters does not contain identifiers. Therefore we will restrict evaluation to *numerical* expressions, i.e. expressions not containing identifiers. So we simplify the rewrite rule for ⟨*factor*⟩ to

⟨*factor*⟩ ::= ⟨*nat*⟩ | '(' ⟨*expression*⟩ ')' .

How do we evaluate? Along the same lines as recognition, complemented by keeping track of intermediate evaluation results. The result is in `evalExp.h` and `evalExp.c`. In `evalExp.h` the following functions are declared:

```
int valueExpression(List *lp, double *vp);
void evaluateExpressions();
```

These functions are defined in `evalExp.c`, together with the functions `valueNumber`, `valueFactor` and `valueTerm`. All these functions are extensions of the functions in `recognizeExp.c`. Besides the return value telling us whether an expression has been recognized, they also write to a second *reference* parameter a pointer that after successful execution points to the value of the subexpression that has been recognized.

The functions `valueXXX(lp,vp)` do two things:

1. they check whether the token list `lp` points to something that can be recognized as `XXX`: if so, they yield the value 1, otherwise the value 0;
2. when they have recognized an `XXX`, `vp` points to its value.

Here `XXX` stands for `Number`, `Factor`, `Term`, `Expression`.

We begin with `valueNumber`:

```
int valueNumber(List *lp, double *vp) {
  if (*lp != NULL && (*lp)->tt == Number ) {
    *vp = ((*lp)->t).number;
    *lp = (*lp)->next;
    return 1;
  }
  return 0;
```

```
}
```

Comparing `valueNumber` with `acceptNumber`, we see that the line

```
*vp = ((*lp)->t).number;
```

has been added.

The function `valueFactor` is obtained from `acceptFactor` (see page 27) by

skipping the recognition of the identifier;
replacing `acceptNumber` by `valueNumber`;
replacing `acceptExpression` by `valueExpression`.

```
int valueFactor(List *lp, double *vp) {
  return
    (  valueNumber(lp,vp)
    || (  acceptCharacter(lp,'(')
       && valueExpression(lp,vp)
       && acceptCharacter(lp,')')
       )
    );
}
```

The adaptation of `acceptTerm` to `valueTerm` is more involved. We introduce the local variable `v` of type `double` for the computations. The test

```
(acceptCharacter(lp,'*')|| acceptCharacter(lp,'/'))
```

is split into two parts since the computation step (multiplication or division) depends on it.

```
int valueTerm(List *lp, double *vp) {
  double v;
  if (!valueFactor(lp,vp)) {
    return 0;
  }
  v = *vp;
  while (*lp != NULL) {
    if (acceptCharacter(lp,'*')) {
      if (valueFactor(lp,vp)) {
        v = v * (*vp);
      } else {
        return 0;
      }
    } else if (acceptCharacter(lp,'/')) {
      if (valueFactor(lp,vp)) {
        v = v / (*vp);
      } else {
        return 0;
      }
    } else {
      *vp = v;
      return 1;
    }
  }
  *vp = v;
  return 1;
}
```

In a similar way we adapt `acceptExpression` and obtain `valueExpression`:

```
int valueExpression(List *lp, double *vp) {
  double v;
  if (!valueTerm(lp,vp)) {
    return 0;
  }
  v = *vp;
  while (*lp != NULL) {
    if (acceptCharacter(lp,'+')) {
      if (valueTerm(lp,vp)) {
        v = v+(*vp);
      } else {
        return 0;
      }
    } else if (acceptCharacter(lp,'-')) {
      if (valueTerm(lp,vp)) {
        v = v-(*vp);
      } else {
        return 0;
      }
    } else {
      *vp = v;
      return 1;
    }
  }
  *vp = v;
  return 1;
}
```

Finally we have the function `evaluateExpressions` to repeatedly read and process input, using the functions defined above. The structure of the definition of `evaluateExpressions` closely resembles that of `recognizeExpressions` in the previous section.

```
void evaluateExpressions() {
  char *ar;
  List tl, tl1;
  double v;
  printf("give an expression: ");
  ar = readInput();
  while (ar[0] != '!') {
    tl = tokenList(ar);
    printf("\nthe token list is ");
    printList(tl);
    tl1 = tl;
    if (valueExpression(&tl1,&v) && tl1==NULL) {
      /* there may be no tokens left */
      printf("this is a numerical expression with value %g\n",
             v);
    } else {
      tl1 = tl;
      if (acceptExpression(&tl1) && tl1 == NULL) {
        printf("this is an arithmetical expression\n");
      } else {
        printf("this is not an expression\n");
```

```
      }
    }
    free(ar);
    freeTokenList(tl);
    printf("\ngive an expression: ");
    ar = readInput();
  }
  free(ar);
  printf("good bye\n");
}
```

A session with the function `evaluateExpressions` may look as follows:

```
give an expression: 2+3*4-5
the token list is 2 + 3 * 4 - 5
this is a numerical expression with value 9

give an expression: 0 - 3/7
the token list is 0 - 3 / 7
this is a numerical expression with value -0.428571

give an expression: 3*x
the token list is 3 * x
this is an arithmetical expression

give an expression: 3**x
the token list is 3 * * x
this is not an expression

give an expression: !
good bye
```

This ends the definition of the evaluation program.

1.5 Exercises

Exercise 1.1. The function `doubleStackSize` doubles the size of the array when the stack gets full. Define a function `extendStackSize` that extends the array with 1000 positions. What can be said about the average time complexity of `push` when `doubleStackSize` is replaced by `extendStackSize`?

Exercise 1.2. Make a stack (i.e. the functions `push` and `pop`) from two queues. What can you say about the time complexity of these functions?

Exercise 1.3. **a.** We consider strings (i.e. arrays of characters) consisting of parentheses `'('` and `')'`, and square brackets `'['` and `']'`. Such a string is called *balanced* if every opening parenthesis has a matching closing parenthesis, every opening bracket has a matching closing bracket, and all matchings are well nested. A more precise, inductive definition reads as follows:

- `()` and `[]` are balanced;
- if s and t are balanced, then st is balanced;
- if s is balanced, then `(`s`)` and `[`s`]` are balanced.

So `([])`, `([]())`, `([()])` and `(())[]` are balanced, but `(()` and `([)]` are not.
Define a function `int balanced (char *str)` that determines whether the input is balanced. Hint: use a stack.
b. Of course, the solution for **a** also works for strings containing only parentheses, no square brackets. But for these strings there is an easier way to check balance. How?

Exercise 1.4. **a.** The function `doubleQueueSize` is only called when the queue is full. Why is this important? What could go wrong if the function is called on a non-full queue? (Hint: consider the part where a split configuration is eliminated.)
b. Modify `doubleQueueSize` so that it can safely be called on non-full queues.

Exercise 1.5. Implement the function `pop` in the queue data type. That is, include functionality in the queue to retrieve and remove the item that was added last.

Exercise 1.6. Create an implementation of the stack data type that implements a priority queue. That is, when `pop` is called, the highest number on the stack is returned rather than the number that was added last. *Tip: you may make use of algorithms learned during Imperative Programming*

Exercise 1.7. The function `copyStack` is intended to make a copy of a stack. It is defined by

```
Stack copyStack(Stack st) {
  Stack copy = newStack(1);
  while (!isEmptyStack(st)) {
    push(pop(&st));
  }
  return copy;
}
```

a. What is wrong with this definition? Hint: compute an example by hand.
b. Give an improved definition of `copyStack`, so that it indeed yields a copy of the input stack, without accessing the underlying array. *Hint: use recursion.*

Exercise 1.8. Define a function `findInList` that checks whether a number occurs in a list. Give a recursive and an iterative solution.

Exercise 1.9. Define a function `removeAllFromList` that removes *all* nodes from a list that contain a given number. Give a recursive and an iterative solution.

Exercise 1.10. Define a function `removeNodeAtPos` that removes a node at a given position from a list. See Figure 1.5.

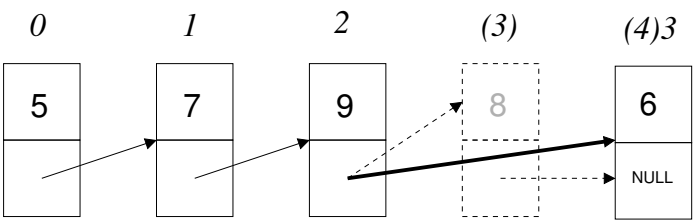

Figure 1.5: Removing a node from a list, at position 3.

Exercise 1.11. Define a function `insertInOrder` that inserts a given number at the correct position in a strictly ordered list. When the number already occurs in the list, the function does nothing.

Exercise 1.12. Define a function `removeLastOccurrence` that, given a list and a number, removes the node with the last occurrence of that number.

Exercise 1.13. Define a function `removeDuplicates` that removes all elements from a list that are preceded by an identical element. Example: when applied to the list `(1 1 1 3 4 4 3 4 1 2 2 2 2 1)`, `removeDuplicates` returns the list `(1 3 4 3 4 1 2 1)`.

Exercise 1.14. **a.** Give a grammar for the formulae of propositional logic, constructed from the constants T (true) and F (false), identifiers for atomic propositions, the connectives $\neg$, $\wedge$, $\vee$ and $\rightarrow$ (in decreasing priority: $\neg$ binds strongest, $\rightarrow$ weakest) and parentheses. Ensure that parsing respects the indicated priorities.
b. Define a function that recognizes the expressions generated by this grammar.
c. Define a function that recognizes and evaluates an expression without atomic propositions to return its truth value according to the definitions you learned in *Introduction to Logic*.

Exercise 1.15.
a. Extend the grammar for expressions so that exponentiation is also allowed. Exponentiation is denoted with `^`, so e.g. `(2+3)^(8/5)`. Ensure that exponentiation binds stronger than the other operators.
b. Adapt the recognizer and the evaluator so that they also can deal with expressions from the extended grammar.

Exercise 1.16. Compare the definition of the function `matchNumber` on p. 23 with the grammar definition of ⟨*nat*⟩ on p 19. Observe that `matchNumber` is in fact too liberal: it accepts expressions that are not produced by ⟨*nat*⟩.
a. Which are these expressions?
b. Give an alternative grammar for ⟨*altnat*⟩ that exactly produces the number representations that are accepted by `matchNumber`.
c. Indicate how the scanner and the recognizer can be brought into correspondence with the original grammar for ⟨*nat*⟩ given on p 19.

Exercise 1.17. **a.** Consider the following grammar for ⟨*sterm*⟩, simple terms:

⟨*sterm*⟩ ::= ⟨*nat*⟩ | `[` ⟨*nat*⟩ `]` ⟨*identifier*⟩ .

Examples of simple terms are `42`, `5 x` and `size`.
Define a function `acceptSterm` with prototype `int acceptSterm(List *lp)`. It tries to identify a *maximal* initial segment of token list `lp` as a production of ⟨*sterm*⟩. When such a production of ⟨*sterm*⟩ has been identified, the function returns 1 and `lp` points to the first item in the remaining token list. Otherwise, the return value is 0 and `lp` is unchanged.
Example: when `lp` corresponds with `15 x + 23`, the return value is 1 and the remaining token list is `+ 23` (and not `x + 23`), for `15 x` is the maximal initial segment of `15 x + 23` that is a production of ⟨*sterm*⟩.
You may use the functions `acceptNumber` and `acceptIdentifier`.
b. We extend ⟨*sterm*⟩ to ⟨*eterm*⟩, terms with exponents:

⟨*eterm*⟩ ::= ⟨*nat*⟩ | `[` ⟨*nat*⟩ `]` ⟨*identifier*⟩ `[` '^' ⟨*num*⟩ `]` .

Typical examples of terms with exponents are `x^3` and `5 y^2`.
Now define a function `acceptEterm` that tries to identify a *maximal* initial segment of token list `lp` as a production of ⟨*eterm*⟩.
You may also use the function `acceptCharacter`.

Chapter 2

Trees

Stacks, queues and lists are linear data structures. In many situations, it is more practical to structure data hierarchically, i.e. in a tree structure. This is the case when the data have an inherent hierarchical structure, e.g. persons in a family tree, or divisions/departments/etc. in a large organization. Additionally, often there is another reason for adopting a tree structure: fast access. Figure 2.1 shows a tree.

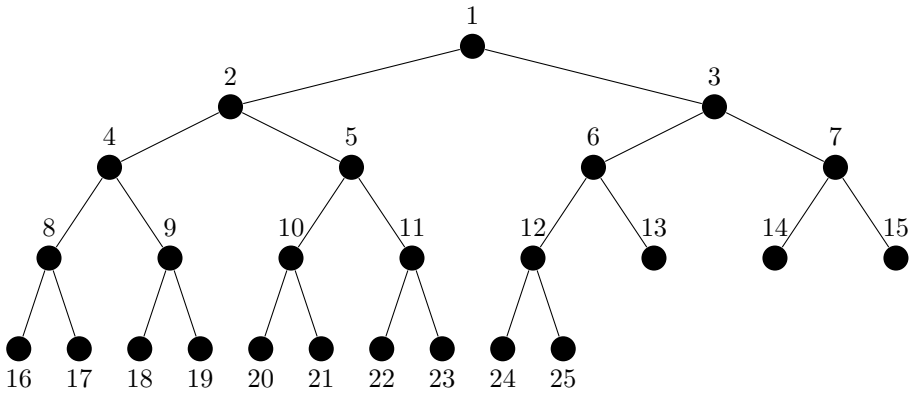

Figure 2.1: A binary tree with 25 nodes. Node 1 is the *root*, the nodes 13 up to 25 are *leaves*. It is also a complete binary tree.

First some terminology. A *tree* consists of *nodes* and *edges*. A tree starts at a node that we call the *root*. Slightly unnaturally, the root of a tree is usually placed on top in illustrations, so

---

trees grow upside down in Computing Science.

Every edge connects two nodes, one above the other. When node u is above node v and an edge connects u and v, we call u the *parent* of v, and v a *child* of u. Every node except the root has exactly one parent. Nodes without children are called *leaves*. Nodes with the same parent are called *siblings*. An *ancestor* of a node v is v itself or an ancestor of the parent of v (observe that this is a recursive definition). A node u is a *descendant* of v when v is an ancestor of u. The root is ancestor of all nodes in a tree. In other words: all nodes in a tree are descendants of the root.

A *branch* is a sequence of nodes $(u_0,u_1,\ldots,u_n)$ such that there is an edge connecting u_i with u_{i+1} for every $i < n$, where all edges point in the same direction. So in Figure 2.1 we have e.g. the branches $(1,3,6,12)$ and $(10,5,2)$, while $(8,4,9,19)$ is not a branch. The length of a branch is its number of edges (i.e. the number of nodes minus one).

The *depth* of a node in a tree is the length of the (unique) branch from that node to the root. So the depth of the root is 0, the children of the root have depth 1, their children have depth 2, and so on. The *height* of a tree is the maximum depth of a node of the tree. The n-th *level* of a tree is the collection of nodes with depth n.

We observe the following property of trees:

the number of edges equals the number of nodes − 1.

Why is this the case? Well, every edge connects a child with its unique parent, so the number of edges equals the number of children; and all nodes except the root are children.

2.1 Binary trees

A *binary* tree is a tree where every node has at most two children: a left child and a right child. This is in essence the simplest kind of trees: when we restrict the number of children of the nodes further to ≤ 1, we obtain a so-called unary tree, which is just a list.

2.1.1 Relation between height and number of nodes

Let us have a look at the relation between the height of a binary tree and the number of nodes it consists of. First one extreme: the height is h, all leaves have depth h and all non-leaves have two children. This is a 'fat' binary tree:

1 node (the root) on level 0,
2 nodes on level 1,
4 nodes on level 2,
8 nodes on level 3,
in general: 2^i nodes on level $i \leq h$.

In total $1 + 2 + 4 + \ldots 2^h = 2^{h+1} - 1$ nodes. We call this the *perfect* binary tree with height h. So we have

for every number h there is a binary tree with height h and $2^{h+1} - 1$ nodes.

Thanks to this property, it is possible to store $2^{h+1} - 1$ items in such a way that every item is accessible in at most h steps. This makes trees so very useful: they enable us to store $\mathcal{O}(2^h)$ items with every item accessible in h steps.

Now the other extreme: the height of the tree is h and there is only one branch. This is the unary tree with $h + 1$ nodes. Slightly less extreme is the tree with depth h where every right child is a leaf. This is a very lean binary tree with only $2h + 1$ nodes. In this kind of lean trees we no longer have the 'tree advantage': only $\mathcal{O}(h)$ nodes are accessible in h steps.

We conclude: in a binary tree with n nodes, the height can vary between $\mathcal{O}(h)$ and $\mathcal{O}(\log(h))$. An interesting question is: How can we ensure that the binary trees that we use as data structures are and remain as thick as possible, so that we can always access many items in few steps? We will come back to this issue later on.

2.1.2 Numbering the node positions

The node positions in a binary tree can be numbered systematically:

the root has position 1;
when a node has position n, its left child has position $2n$
and its right child has position $2n + 1$.

This is the standard numbering of node positions in a binary tree. See Figure 2.1. Observe that 0 is not used in this numbering.

In a perfect tree with height h, the root gets position 1, its children in level 2 the positions 2 and 3, the nodes in level 2 the positions 4, 5, 6, and 7, and so on. In general: in level $i < h$

the nodes get the positions 2^i up to $2^{i+1} - 1$. So the node positions of a perfect tree form a sequence without gaps (i.e. if there is a node with position n and $m < n$, then there is also a node with position m).

But the perfect trees are not the only binary trees where the positions form a sequence without gaps. This property is shared by all trees where every level except the last is fully filled (i.e. 2^i nodes in level i) and where all nodes in the last level are maximally to the left. We call such a tree a *complete* tree. The tree in Figure 2.1 is an example of a complete tree.

How many nodes are there in a complete tree with height h? Every level $i < h$ contains 2^i nodes, and level h contains between 1 and 2^h nodes. So the total number of node positions in the tree lies between $1 + 2 + 4 + \ldots 2^{h-1} + 1$ and $1 + 2 + 4 + \ldots 2^{h-1} + 2^h$, i.e. between 2^h and $2^{h+1} - 1$. In other words: a complete binary tree with n nodes has height $\lfloor {}^2\log(n) \rfloor$. Here $\lfloor \cdot \rfloor : \mathbb{R} \to \mathbb{Z}$ is the *floor* function: $\lfloor x \rfloor$ is the greatest integer n with $n \leq x$.

2.1.3 Two representations of binary trees

We present two representations of binary trees. The first uses pointers and is related to the list representation of the previous chapter, the second is more surprising and uses an array.

Pointer representation. The only difference with lists is that a node has *two* pointers to other nodes. We define the following type:

```
typedef struct TreeNode *Tree;

struct TreeNode {
  int item;
  Tree leftChild, rightChild;
};
```

After this definition `TreeNode` is a composite type, with a field `item`, a field `leftChild` and a field `rightChild`. Furthermore `Tree` is the type of pointers to structures of type `TreeNode`. Compare this with the definition of lists in Section 1.3.

We define two functions to create trees:

```
Tree emptyTree() {
  return NULL;
}

Tree newTree(int n, Tree tL, Tree tR) {
  Tree new = malloc(sizeof(struct TreeNode));
  assert(new != NULL);
  new->item = n;
  new->leftChild = tL;
  new->rightChild = tR;
  return new;
}
```

With these functions, we can make the empty tree and two one-node trees, containing the values 5 and 8:

```
Tree t0 = emptyTree();
Tree t1 = newTree(5,t0,t0);
Tree t2 = newTree(8,t0,t0);
```

Now we can use these to make a three-node tree:

```
Tree t3 = newTree(14,t1,t2)
```

We show another binary tree in Figure 2.2.

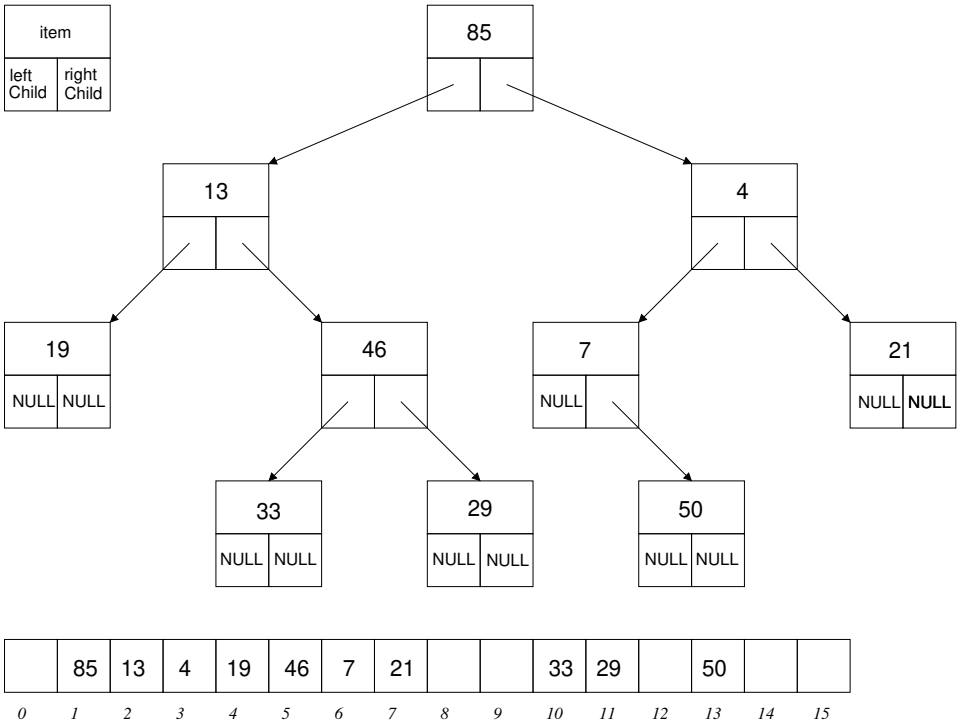

Figure 2.2: The pointer representation and the array representation of a binary tree.

Array representation. For this representation we use the numbering of node positions in a binary tree. The positions of the nodes are the indices in the array, and the first field (with index 0) remains empty. Given a node with index n, its parent has index $n/2$ (unless $n = 1$) and its children have indices $2n$ and $2n + 1$.

In general, there will be holes in the array representation of a binary tree, i.e. empty fields between nonempty fields. As a consequence, it may happen that a large array is needed to store a tree with a small number of nodes. To illustrate this, we consider the tree with depth n where all left children are leaves. The positions of the nodes are $1, 2, 3, 6, 7, 14, 15, \ldots, 2^{n+1} - 2, 2^{n+1} - 1$. In order to represent this with an array as described above, we need an array of length 2^{n+1} in which $2^{n+1} - 2n - 1$ fields will be empty! This is not very efficient, and a pointer representation is preferred here. However, the array representation is easy to implement and efficient for certain trees, e.g. for complete binary trees. We will use the array representation in Section 2.3 for the heap structure.

There is a small problem with the representation of a binary tree by an array `b[]`: how can we see which node positions occur in the tree, and which not? For a complete tree, it suffices to store the number of nodes `n`, for then `b[i]` contains a node exactly when `0 < i <= n`. But this does not work for the general case. We come back to this in an exercise.

2.1.4 Traversing a binary tree

Like for lists, it is possible to traverse a binary tree systematically and to perform an action in each node. We call this action `visit`. There are several ways to do this, depending on the order of performing `visit` and going to the left and right child of the node in question.

We use the pointer representation of binary trees. When we first perform `visit` before we go to the children, we get the *preorder traversal*:

```
void preOrder(Tree t) {
  if (t == NULL) {
    return;
  }
  visit(t);
  preOrder(t->leftChild);
  preOrder(t->rightChild);
}
```

Alternatives are *postorder traversal*:

```
void postOrder(Tree t) {
  if (t == NULL) {
    return;
  }
  postOrder(t->leftChild);
  postOrder(t->rightChild);
  visit(t);
}
```

and *inorder traversal*:

```
void inOrder(Tree t) {
  if (t == NULL) {
    return;
  }
  inOrder(t->leftChild);
  visit(t);
  inOrder(t->rightChild);
}
```

The names of the three `...Order` functions will become more clear in the next section. For now we illustrate the effect of these functions when applied to the binary tree shown in Figure 2.1. We interpret `visit` as `printf("%d ",t->item)`.

The function `preOrder` leads to

```
1 2 4 8 16 17 9 18 19 5 10 20 21 11 22 23 3 6 12 24 25 13 7 14 15
```

while `postOrder` yields

```
16 17 8 18 19 9 4 20 21 10 22 23 11 5 2 24 25 12 13 6 14 15 7 3 1
```

and with `inOrder` we get this:

```
16 8 17 4 18 9 19 2 20 10 21 5 22 11 23 1 24 12 25 6 13 3 14 7 15
```

2.2 Search trees

Now we will use binary trees as structures to store and retrieve items efficiently. We assume that the items have a *linear order*: if x and y are items, then $x \leq y$ or $y \leq x$. Examples of sets with linear orders are:

- `int`, the integers with the usual order $\leq$;
- `char`, the characters with the order determined by the ASCII table index:

  `0 < 1 ... < 8 < 9 < A < B < ... < Y < Z < a < b < ... < y < z`

- the set of strings of characters with the *lexicographical* order. The lexicographical order is used to order words in a dictionary. It is defined recursively by

$$a_0a_1\ldots a_{m-1} \leq b_0b_1\ldots b_{n-1} \text{ iff } \begin{array}{l} m = 0 \\ \text{or } (m,n > 0 \text{ and } a_0 < b_0) \\ \text{or } (m,n > 0 \text{ and } a_0 = b_0 \text{ and } a_1\ldots a_m \leq b_1\ldots b_n) \end{array}$$

In the examples we will use integers as items.

A *search tree* is a binary tree where every node contains a value from a linearly ordered set of values, and which satisfies the *search tree property* which reads as follows.

All nodes k with a value x satisfy:
all values in the left subtree of k are smaller than x,
and all values in the right subtree of k are greater than x.

Observe that the search tree property does not admit that a value occurs more than once in the search tree. Occasionally, another definition of search tree is used where a value may occur more than once. We will not do so.

Figure 2.3 shows a search tree. Note that the search tree property guarantees that the function `inOrder` from the previous section yields a sorted list.

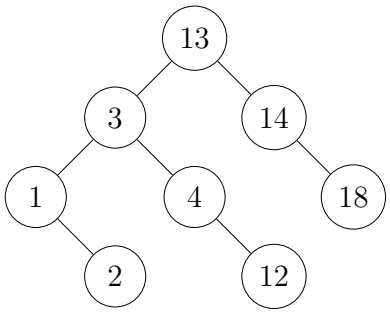

Figure 2.3: A search tree.

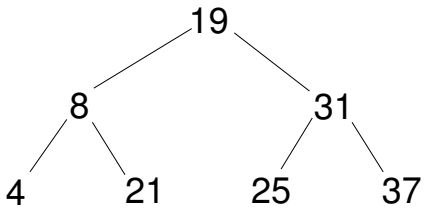

Figure 2.4: This is not a search tree. Why?

The search tree property is very useful when we search for a value n in the search tree. When the search tree is empty, we are done. If not, we compare the value x in the root w with n. When $x = n$, we have found n in the search tree and we are done. If $n < x$, we know that we will not find n in the right subtree of r, for all values there are greater than x. So we go to the left subtree of r. Similarly, if $n > x$, we go to the right subtree of r. Now we repeat the procedure until we reach a leaf.

The above is an informal recursive description of the algorithm for searching in a search tree. We will present a C program that implements this description. But first we will give a description of the algorithm somewhere in between the textual description and the implementation in a programming language, in so-called *pseudocode*.

algorithm SearchInSearchTree(T, n)
    input : search tree T containing numbers, and number n
    output : a node in T that contains n if it exists, otherwise `not found`
    if T empty **then**
        return `not found`
    r ← the root of T
    x ← the value in r
    if n = x **then**
        return r
    if n < x **then**
        return SearchInSearchTree(left subtree of T, n)
    /∗ now we have: n > x ∗/
    return SearchInSearchTree(right subtree of T, n)

What can we say about this kind of description? It looks a bit like a computer program because of the programming constructs if/then and return, and the use of a (recursive) function with arguments. But at the same time it deviates from programming languages:

- it contains English text,
- we often omit type declarations,
- there are no semicolons ';',
- the arrow '← ' is used to assign value to variables (instead of the equality symbol `=`),
- the equality symbol '=' is used to compare two values (instead of `==`),
- the block structure is only indicated by indentation (instead of `{`curly brackets`}`).

Pseudocode is a useful tool to describe an algorithm in a structured and somewhat precise way. Not precise enough for a computer: it is intended for human users, not for computers. An algorithm described in pseudocode is often more compact and more elegant than a computer program. There is no exact definition of pseudocode: textbooks and research papers use many variants, all based on the principles listed above. Appendix B contains a description of the pseudocode that we use in this course.

We return to search trees. The search algorithm can be implemented in C as follows:

```
Tree searchNode(Tree t, int n) {
  if (t == NULL) {
    return NULL;
  }
  if (n == t->item) {
    return t;
  }
  if (n < t->item) {
    t = t->leftChild;
  } else {
    t = t->rightChild;
  }
  return searchNode(t,n);
}
```

We can add a value n to a search tree as follows. When the tree is empty, we replace it by a tree with only a root containing n. Otherwise, we compare n with the value x in the root of the tree. If $n < x$ we go left, and if $n > x$ we go right, and in both cases we repeat the procedure. When $n = x$ we do nothing. This can be described in pseudocode as follows:

algorithm AddToSearchTree(T, n)
input : search tree T, number n
output : T if it contains n, otherwise T extended with a node containing n,
in such a way that the search tree property is preserved
if T empty **then**
return tree with only a root containing n
x ← the value of the root of T
if n < x **then**
Tleft ← left subtree of T
return T with Tleft replaced by AddToSearchTree(Tleft, n)
if x < n **then**
Tright ← right subtree of T
return T with Tright replaced by AddToSearchTree(Tright, n)
/∗ now n = x holds and we do nothing ∗/
return T

This is implemented in C as follows:

```
Tree addInSearchTree(Tree t, int n) {
  if (t == NULL) {
    return newTree(n,emptyTree(), emptyTree());
  }
  if (n < t->item){
    t->leftChild = addInSearchTree(t->leftChild,n);
  } else if (n > t->item) {
    t->rightChild = addInSearchTree(t->rightChild,n);
  }
  return t;
}
```

Removing a node v from a search tree is somewhat harder. When v is a leaf, it is quite simple. When v has only one child, it is not so difficult: that child will take the position of v. The difficult case is when v has two children: what to do with these children? In general, we cannot give them to the parent of v: that parent may have another child beside v, so it would end up with three children.

We take recourse to a trick here: we will not remove v, but its successor w in the inorder traversal of the search tree. First we observe that v has indeed a inorder successor, for otherwise it would not have a right child. We shall show that w, the inorder successor of v, has at most one child, so removing w is simple. We then put the value of w in v. Since v and w are next to each other in the inorder traversal of the the search tree, the search tree property is not disturbed by this operation. See Figure 2.5.

How do we find w and why has it at most one child? We obtain w as follows: go to the right child u of v, and follow the branch from u downwards that turns left as long as possible. This branch stops in the first node below u where there is no left turn (this may be u itself), and we call this node w. Because of the search tree property, the value of w is greater than the value of v. Moreover, there are no nodes with a value between the values of v and w: any such node should be left below w, but there is nothing there since w has no left child. And that is why w has at most one child. When we write this down in pseudocode, we get:

algorithm InorderSuccessor(T,v)
input : search tree T with node v having two children
output : the successor v in T according to inorder traversal
u ← the right child of v
w ← the lowest left descendant of u
return w

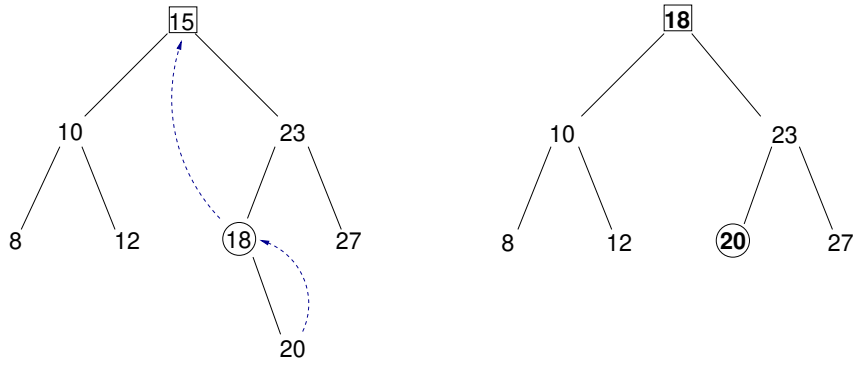

Figure 2.5: Removing item 15, in the root, from a search tree. The inorder successor of the root is the encircled node, containing item 18. This value moves to the root, while the item 20 in the only child of the inorder successor replaces item 18.

```
algorithm RemoveFromSearchTree(T,n)
    input : search tree T, number n
    output : T with value n removed (when present)
    if there is no node with value n in T then
        return T
    v ← the node in T with value n
    if v is a leaf then
        return T with v removed
    else if v has 1 child then
        return T with the subtree from v replaced by the subtree from the child of v
    else /* the difficult case: v has two children */
        w ← InorderSuccessor(T,v)
        (value of v) ← (value of w)
        /* now we use that w has no left child */
        if w has a right child then
            return T with w replaced by its right child
        else /* w has no children, so it is a leaf */
            return T with w removed
```

We leave the implementation of these algorithms in C as an exercise.

What can we say about the time complexity of the search tree algorithms we just presented? In all these algorithms (search, add, remove), we start in a node and traverse a branch. In the nodes that we pass, we do simple things that only require constant time. Therefore, an upper bound for the time complexity is $\mathcal{O}(h)$ with h the maximum length of a branch, i.e. the height of the search tree. We have seen that h may vary between $\mathcal{O}(n)$ and $\mathcal{O}(\log(n))$ with n the number of nodes in the search tree.

Conclusion: the algorithms on search trees presented here are fast (i.e. $\mathcal{O}(\log(n))$) provided the search tree is balanced enough and the height h is in $\mathcal{O}(\log(n))$. However, for less balanced trees, they may be $\mathcal{O}(n)$, which is slow when n gets really big. So for a search tree to be efficient, care has to be taken that it is balanced and remains so.

Experience shows that this is not automatically the case. When a search tree is used statically (i.e. only search, no adding or removing nodes), it suffices to construct a balanced search tree, which is not very difficult. Problems arise when the search tree is used dynamically: adding e.g. larger and larger values will lead to a lopsided, unbalanced search tree.

Many solutions for this problem have been (and are being) developed in the course of time. Some solutions are based on reconstruction of the search tree to restore the balance. Other solutions work with alternative data structures. We will have a closer look at some of these solutions in *Advanced Algorithms and Data Structures*.

2.3 Heaps

Like a search tree, a *heap* is a binary tree where every node contains a value from a linearly ordered set of values. Moreover, a heap is always a complete binary tree (see page 37) and satisfies the *heap property*.

For each node v, its descendants have a value $\leq$ the value in v.

As a consequence, the sequence of values that you encounter on a branch from the root is weakly descending, and the largest value of the heap is in the root. So, in some sense, the order in a heap is vertical, while the order in a search tree is horizontal. Observe that different nodes in a heap may contain the same value, in contrast to the situation in search tree where we do not allow this. An example of a heap is given in Figure 2.6.

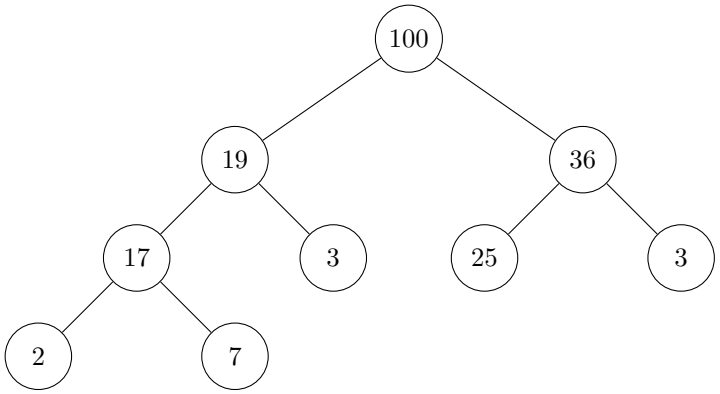

Figure 2.6: A heap

We shall show that a heap implements a *priority queue*. This is a variant of the normal queue we discussed in Section 1.2, with the following definition.

A priority queue contains items from a linearly ordered set of values.
`enqueue` adds an item to the priority queue.
`removeMax` yields and removes the largest item of the priority queue (provided the priority queue is not empty).

Usually, there is also a function `getMax` which yields the largest item of the priority queue without removing it.

With a heap, the first action in the implementation of `removeMax` is easy: take the value in the root. But then we have to restore the heap property. Similarly, the implementation of `enqueue` starts with a straightforward action: create a new node such that the tree remains complete, an put the item in that node. After that, we again must restore the heap property.

We proceed as follows. First we describe the algorithms for Enqueue and RemoveMax in pseudocode. Then we describe the auxiliary algorithms Upheap and Downheap to restore 'heapness'. Adding an item to a heap is done as follows:

algorithm Enqueue(n)
  input : number n
  result : a node with value n has been added to the heap
  add a new node v to the heap so that it remains a complete tree
  put value n in v
  Upheap(v) /* to restore heap order */

Removing the largest value from a heap is done as follows:

algorithm RemoveMax()
    output : the maximum value a in the heap
    result : the value in the root has been removed, and the heap order has been restored
    w ← the root of the heap
    n ← value of w
    v ← last node of the heap
    value of w ← value of v
    remove v from the heap
    Downheap(w) /∗ to restore heap order ∗/
    return n

Now we still need the algorithms Upheap and Downheap.

algorithm Upheap(v)
    input : node v in a heap, with possibly a conflict
        with the heap order between v and its parent
    result : heap order has been restored
    if v ≠ the root of the heap **then**
        u ← the parent of v
        if (value of u) < (value of v) **then**
            swap the values of u and v
            Upheap(u)

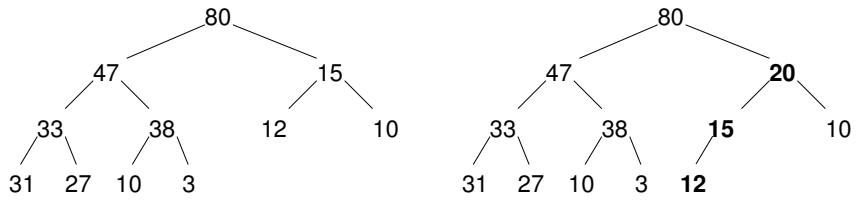

Figure 2.7: Addding 20 to a heap. 20 is initially placed at the first free position, i.e. as a left child of 12; then, with two Upheap steps, heap order is restored.

algorithm Downheap(v)
    input : node v in a heap, with possibly a conflict
        with the heap order between v and its children
    result : heap order has been restored
    if v has at least one child **then**
        lc ← the left child of v
        rc ← the right child of v (or lc, when v has no right child)
        if (value of lc) > (value of v) and (value of lc) > (value of rc) **then**
            swap the values of lc and v
            Downheap(lc)
        else if (value of rc) > (value of v) **then**
            /∗ now also (value of rc) ≥ (value of lc) ∗/
            swap the values of rc and v
            Downheap(rc)

We have a look at the time complexity of these algorithms. First we observe that Upheap and Downheap have time complexity $\mathcal{O}(h)$, with h the height of the heap. As a consequence, the time complexity of Enqueue and RemoveMax is also $\mathcal{O}(h)$. But a heap is a complete binary tree, so its height h is $\leq \log(n)$, so the time complexity of these algorithms is $\mathcal{O}(\log(n))$.

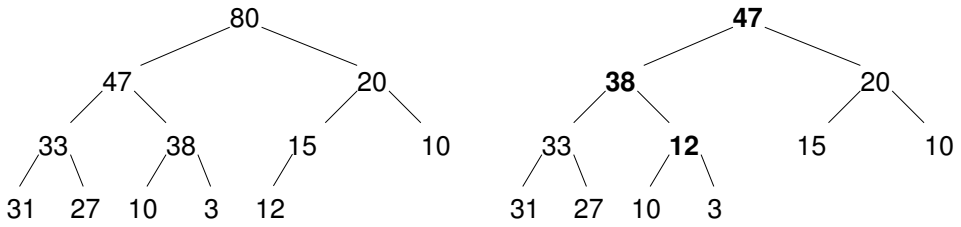

Figure 2.8: A heap before and after removing the largest value, yielding 80. Note initially we move 12, the value at the last position to the root. After that we need two Downheap steps in which we take a left and a right branch, to restore the heap order.

2.3.1 Implementation in C

We will now implement a heap in C. A first decision we need to make is whether to use the pointer or the array representation for the tree.

Note that the Upheap algorithm needs access to the parent of a given node. But this is not available in the pointer representation of binary trees: A node only has pointers to its children, not to its parent. Hence we will now use the array representation of a binary tree for the implementation of the heap algorithms in C.

We define a type `Heap`:

```
typedef struct Heap {
  int *array;
  int front;
  int size;
} Heap;
```

For the creation of a heap, we have the function `makeHeap`:

```
Heap makeHeap () {
  Heap h;
  h.array = malloc(1*sizeof(int));
  assert(h.array != NULL);
  h.front = 1;
  h.size = 1;
  return h;
}
```

The field `h.front` indicates the first free position in the array. Observe that `makeHeap()` yields an empty heap with size 1 (and not 0). Recall that node position 0 does not occur in binary trees.

To deal with empty heaps, we define

```
int isEmptyHeap (Heap h) {
  return (h.front == 1);
}

void heapEmptyError() {
  printf("heap empty\n");
  abort();
}
```

The function `enqueue` is defined by

```
void enqueue (int n, Heap *hp) {
  int fr = hp->front;
  if (fr == hp->size) {
    doubleHeapSize(hp);
  }
  hp->array[fr] = n;
  upheap(hp,fr);
  hp->front = fr + 1;
}
```

Observe that `enqueue` uses the functions `doubleHeapSize` and `upheap`. Their definition is asked for in two exercises.

We define `removeMax` by

```
int removeMax(Heap *hp) {
  int n;
  if (isEmptyHeap(*hp)) {
    heapEmptyError();
  }
  n = hp->array[1];
  hp->front--;
  hp->array[1] = hp->array[hp->front];
  downheap(hp,1);
  return n;
}
```

The function `downheap` used in `removeMax` is the subject of another exercise.

2.3.2 Remark about priority queues with unique elements

Sometimes a priority queue is desired in which all priorities are different. This means that an item is not enqueued in the priority queue when the queue already contains an item with the same priority. When we assume that the priority of an element is just its value, this comes down to: a priority queue without duplicates.

Avoiding duplicates is rather easy with the ordered list implementation (see Section 1.3.4). Here `enqueue` is implemented by `insertInOrder` which traverses the ordered list: it will find a duplicate of the item to be inserted whenever it occurs in the ordered list. However, for the heap implementation this does not hold: `enqueue` only visits the items in the branch that runs from the insertion point to the root, and it will miss duplicates elsewhere in the heap.

2.4 Tries

After having done several things with binary trees, we now turn to trees with higher branching degrees. We will use them to represent texts and strings in such a way that we can search remarkably fast in them. More precisely, we can realize the following strong feat.

Let a text T with length n be given. There is an auxiliary structure with size in $\mathcal{O}(n)$ so that we can check in $O(k)$ time (!) whether T contains an arbitrary pattern p with length k.

The remarkable fact is that the search speed does not depend on n, the length of the text T. First we observe that a straightforward approach does not work here. It would run as follows: first check whether p occurs at the beginning of T, i.e. from position 0. That

will take $0(k)$ time: check whether $p[0] = T[0]$, if so check whether $p[1] = T[1]$, if so check whether $p[2] = T[2]$, and so on, until we find inequality or reach the end of p or T. When the result is positive, we are done. When it is negative, we do the same from position 1 in T: compare $p[0]$ with $T[1]$, $p[1]$ with $T[2]$, and so on. This may continue up to position $n-k-1$ of T. So in the ultimate case we have to do $\mathcal{O}(n)$ often something that takes $\mathcal{O}(k)$ time, in total $\mathcal{O}(kn)$ time. To see that this simple-minded approach may take that long, consider $T = aaa \ldots aaa$ with length $2n$ and $p = aaa \ldots aab$ with length n.

This is indeed a very simple approach, and with some cleverness we can improve the search for p in T to $\mathcal{O}(n)$ time. But that is still far away from the claimed $\mathcal{O}(k)$. To realize it, we will construct an auxiliary structure based on text T. This structure is a *suffix trie*[1]: it takes $\mathcal{O}(n)$ time to build it, and it fits in $\mathcal{O}(n)$ memory. With the suffix trie we can check in $\mathcal{O}(k)$ time (k being the length of p) whether and where pattern p occurs in text T.

In a few steps, we shall introduce the notion of suffix trie. We start with standard tries.

2.4.1 Standard tries

The standard trie is a data structure for the efficient solution of the following (simpler) problem: given a collection W of words, find out whether the word p occurs in W. For a smooth treatment, we assume that W satisfies the *no-initial-segment property*: it does not contain words v and w such that v is an initial segment of w. If W does not satisfy this property, it is not difficult to change it into W' that does satisfy it: take a character that does not occur in W and add it at the end of every word in W.

How to represent W so as to enable fast search? The idea is:

> build a tree where all nodes except the root contain a letter; nodes that are siblings contain different letters, and every word in W corresponds with a branch from the root.

We call this type of tree a *trie*. See Figure 2.9.

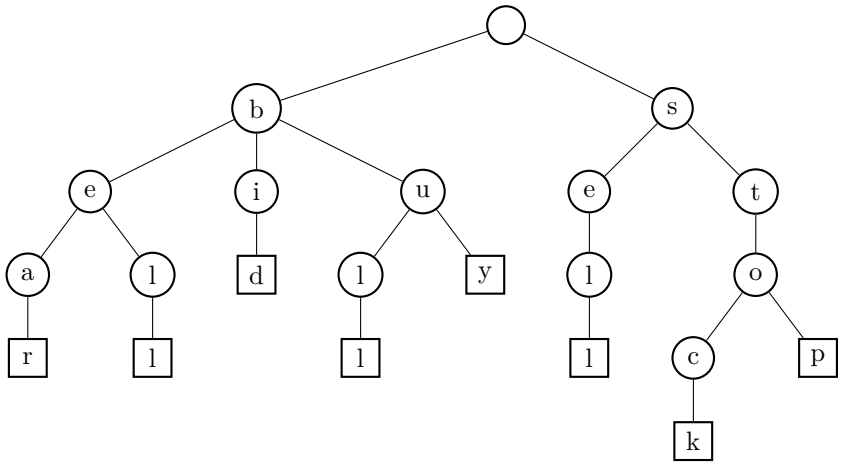

Figure 2.9: A standard trie for the set {`bear`, `bell`, `bid`, `bull`, `buy`, `sell`, `stock`, `stop`}.

The definition is as follows. A *standard trie* T for the collection W of words is a tree with the following properties:

- the root of T is empty, and every other node contains a letter;
- the children of a node of T contain different letters and are in alphabetical order;
- the branches in T from the root correspond exactly with the words in W.

[1] 'trie' is pronounced as 'try': the term is derived from *retrieval*.

How much memory is required for a standard trie? Let n be the sum of the lengths of the words in W. There are at most $n+1$ nodes in T: this maximum is attained when all words start with a different letter. In general, there is overlap between the words and hence less nodes, but $n+1$ is the upper limit. Every node contains one letter, which requires a fixed amount of memory. Furthermore, there are maximally n edges. So the memory required for T is in $\mathcal{O}(n)$ with n the total length of all words in W. (In the next section, we shall reduce this to $\mathcal{O}(m)$ with m the number of words.)

Searching whether pattern p occurs in W comes down to trying to follow from the root in T the branch that corresponds with p. During the search, we successively go from a node to the next node to match the next letter in p. When there is no such node, the search stops with a negative result. When we reach the end of p, we check whether we are in a leaf of T. If so, we have found that p occurs in W. Otherwise the search ends negatively (thanks to the no-initial-segment property of W). In pseudocode:

```
algorithm SearchInTrie(T,w)
    input standard trie T, word w
    output Yes if w occurs in T, otherwise No
    k ← root of T
    while w not empty do
        x ← first letter of w
        w ← w minus x
        if k has no child containing x then
            return No
        k ← child of k that contains x
    if k is a leaf then
        return Yes
    else
        return No
```

2.4.2 The compressed trie

A *compressed trie* is obtained from a standard trie by compressing the non-branching parts of a branch in a single node. The compressed trie has the following properties:

- the root is empty, and every other node contains *a nonempty string*;
- the children of a node contain strings with different initial letters and are ordered alphabetically *on the initial letter of the string*;
- *there are no nodes with branching degree 1* (if W contains at least two words);
- the branches from the root correspond exactly with the words in W.

Searching in a compressed trie is not much different from searching in a standard trie. Adapting the search algorithm is an exercise.

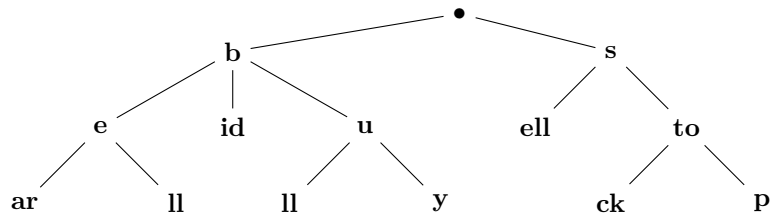

Figure 2.10: The compressed trie of the standard trie in Figure 2.9.

What is the effect of the compression of the trie? We claim: the compressed trie contains at most $2m$ nodes (m is the number of words). We can see this as follows. There are m words, so m branches in the tree and m leaves. We shall show that there at most m non-leaves by giving an *injection* $f : \mathit{NonLeaves} \to \mathit{Leaves}$. Given a non-leaf v, the leaf $f(v)$ is found as follows. First go down to the leftmost child of v, then continue with downward steps to the rightmost child until you reach a leaf. (Check that this is an injection, i.e. different non-leaves yield different leaves.) This demonstrates that $\#\mathit{NonLeaves} \leq \#\mathit{Leaves}$ ($\#X$ denotes the number of elements of X), so the compressed trie contains at most $2m$ nodes.

This reduces the number of nodes from $\mathcal{O}(n)$ to $\mathcal{O}(m)$. However, this does not have the desired effect on the memory use, for the size of the nodes has increased. It was one letter, now it is a string. The total length of all strings in the nodes has $\mathcal{O}(n)$ as an upper bound. So the upper bound for the memory use is not improved. High time for a new idea.

2.4.3 The compact trie

The compact trie is obtained from the compressed trie by replacing the strings in the nodes by their coordinates. For this purpose, we work with an array A that represents the collection W of words. For the trie in Figure 2.9, the array A is

$$A = \{\texttt{b,e,a,r,b,e,l,l,b,i,d,b,u,l,l,b,u,y,s,e,l,l,s,t,o,c,k,s,t,o,p}\}$$

which we index starting with 0 as usual:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
 b  e  a  r  b  e  l  l  b  i  d  b  u  l  l  b  u  y  s  e  l  l  s  t  o  c  k  s  t  o  p
```

Finally, the compact trie for our example is shown in Figure 2.11:

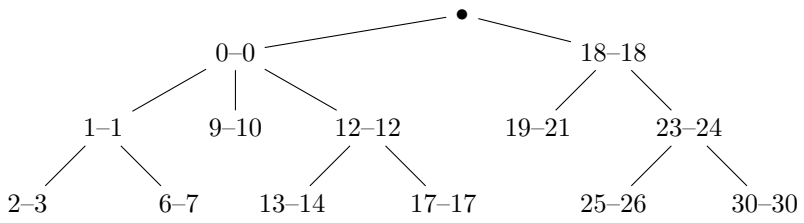

Figure 2.11: The compact trie of the compressed trie in Figure 2.10.

The compact trie has the following properties:

- every node except the root contains *two numbers referring to a string*;
- the children of a node are ordered alphabetically on initial letter;
- there are no nodes with branching degree 1;
- the branches from the root correspond exactly with the words in W.

We have reduced the size of every node to a fixed value, so the total memory use of a compact trie is $\mathcal{O}(m)$ with m the number of words.

So we have a data structure with size in $\mathcal{O}(m)$ (m the number of words in W) with which we can search in $\mathcal{O}(k)$ time for a pattern p of size k in W. Recall that we are out for more: searching in a text. This requires a new idea. As a starter we observe: with a (compact) trie, we can efficiently check whether a pattern occurs as a *prefix* (i.e. an initial segment) of one of the words in W. This is similar to searching for a word, with the only difference that we do not have to check whether we have reached the end of the word when we found a pattern match.

2.4.4 Suffix tries

We might try to solve the problem with a compact trie that contains *all* substrings of a given text T. There are n substrings with length 1, $n-1$ substrings with length 2 $\ldots$, $n-i$ substrings with length i, $\ldots$, and finally 1 substring with length n. In total $n+(n-1)+\cdots+1 = n(n+1)/2$, ie. $O(n^2)$ many. That is quite a lot, especially when n is large. But, luckily, it suffices to work with only the n *suffixes* (end segments) of T. For we have:

---

every substring of a string is the prefix of a suffix.

We observe that the compact *suffix trie* of a text T of length n fits in $\mathcal{O}(n)$ memory and enables us to search for a pattern with length k in $\mathcal{O}(k)$ time.

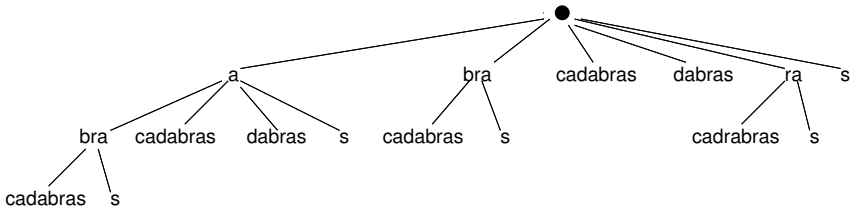

Figure 2.12: The suffix trie for the string `abracadabras`.

Finally about the construction of a compact suffix trie of a string. All straightforward approaches, e.g. first build a trie with all suffixes, followed by compressing it and making it compact, require $\mathcal{O}(n^2)$ time. There are algorithms to do this in $\mathcal{O}(n)$ time, but they are rather complicated and are therefore not treated here.

2.5 Application: expression trees

2.5.1 Expression trees

With binary trees, we can make the structure of (arithmetical) expressions explicit. The non-leaves contain operators, the leaves contain numbers or variables. The structure of the tree indicates in which order the computations are to be performed. So parentheses are not required in an expression tree. See Figure 2.13.

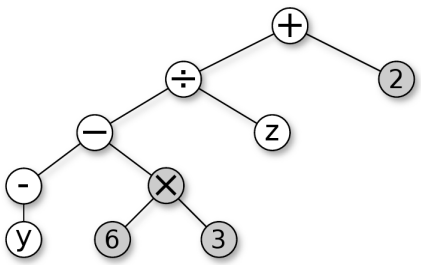

Figure 2.13: An expression tree of the expression $(((-y)-(6\times 3))/z)+2$.

The code described in this section is available on Themis, and can be useful for the practical assignment.

2.5.2 Prefix expressions

We now present some functions to transform expressions in expression trees. To keep things simple, we use *prefix expressions*, where the operator is placed before its two operands, instead of between them (as in the usual infix expressions). As a consequence, the language is no longer ambiguous, so parentheses are not needed.

Example. $+3 * t\ 7$ stands for $3 + (t * 7)$, and $+ - z\ 33\ 3$ for $(z - 33) + 3$.

A prefix expression is a number, or an identifier, or an operator followed by two prefix expressions. This is formalized in the following grammar:

⟨*prefixExp*⟩ ::= ⟨*number*⟩
| ⟨*identifier*⟩
| '+' ⟨*prefixExp*⟩ ⟨*prefixExp*⟩
| '-' ⟨*prefixExp*⟩ ⟨*prefixExp*⟩
| '*' ⟨*prefixExp*⟩ ⟨*prefixExp*⟩
| '/' ⟨*prefixExp*⟩ ⟨*prefixExp*⟩

We use the scanner from Section 1.4.4 to scan the input and transform it in a token list. From the token list we want to build an expression tree. We follow the approach for the recognizer in Section 1.4.5 and extend the functionality of the functions defined there in order to build expression trees. First the definition of the type of nodes in an expression tree:

```
typedef struct ExpTreeNode *ExpTree;

typedef struct ExpTreeNode {
  TokenType tt;
  Token t;
  ExpTree left;
  ExpTree right;
} ExpTreeNode;
```

The types `TokenType` and `Token` are defined in Section 1.4.4.

The only, rather obvious, difference with the nodes in a token list is the presence of a second pointer to another node. For creating a tree node we have the following function:

```
ExpTree newExpTreeNode(TokenType tt, Token t, ExpTree tL,
    ➥ ExpTree tR) {
  ExpTree new = malloc(sizeof(ExpTreeNode));
  assert (new!=NULL);
  new->tt = tt;
  new->t = t;
  new->left = tL;
  new->right = tR;
  return new;
}
```

For the evaluation of numerical expressions we used the function `valueNumber` from `evalExp.c` to compute the value of a number in the token list. We now define two functions for processing the value of an identifier and an operator. The auxiliary function `isOperator` is used to check whether an operator is arithmetical.

```
int valueIdentifier(List *lp, char **sp) {
  if (*lp != NULL && (*lp)->tt == Identifier) {
    *sp = ((*lp)->t).identifier;
    *lp = (*lp)->next;
    return 1;
```

```
  }
  return 0;
}

int isOperator(char c) {
  return (c == '+' || c == '-' || c == '*' || c == '/');
}

int valueOperator(List *lp, char *cp) {
  if (*lp != NULL && (*lp)->tt == Symbol
                  && isOperator(((*lp)->t).symbol) ) {
    *cp = ((*lp)->t).symbol;
    *lp = (*lp)->next;
    return 1;
  }
  return 0;
}
```

For freeing the memory used for an expression tree, we define the following function.

```
void freeExpTree(ExpTree tr) {
  if (tr==NULL) {
    return;
  }
  freeExpTree(tr->left);
  freeExpTree(tr->right);
  free(tr);
}
```

Observe that here, unlike in `freeTokenList` in `scanner.c`, the strings in identifier nodes are not freed. The reason is that the function `newExpTreeNode` does not allocate memory for strings in nodes, but only a pointer to a string in a node in the token list.

Now we can build the expression tree. As we have only expressions and no terms and factors, the function `treePrefixExpression` is defined with 'normal' recursion:

```
int treePrefixExpression(List *lp, ExpTree *tp) {
  double w;
  char *s;
  char c;
  Token t;
  ExpTree tL, tR;
  if (valueNumber(lp,&w)) {
    t.number = (int)w;
    *tp = newExpTreeNode(Number, t, NULL, NULL);
    return 1;
  }
  if (valueIdentifier(lp,&s)) {
    t.identifier = s;
    *tp = newExpTreeNode(Identifier, t, NULL, NULL);
    return 1;
  }
  if (valueOperator(lp,&c) && treePrefixExpression(lp,&tL)) {
    if (treePrefixExpression(lp,&tR)) {
      t.symbol = c;
      *tp = newExpTreeNode(Symbol, t, tL, tR);
```

```
      return 1;
    } else { /* without 'else' there is a memory leak */
      freeExpTree(tL);
      return 0;
    }
  }
  return 0;
}
```

Given an expression tree, we can print its expression in infix notation as follows.

```
void printExpTreeInfix(ExpTree tr) {
  if (tr == NULL) {
    return;
  }
  switch (tr->tt) {
  case Number:
    printf("%d",(tr->t).number);
   break;
  case Identifier:
    printf("%s",(tr->t).identifier);
    break;
  case Symbol:
    printf("(");
    printExpTreeInfix(tr->left);
    printf(" %c ",(tr->t).symbol);
    printExpTreeInfix(tr->right);
    printf(")");
    break;
  }
}
```

Now we want to compute the numerical value of an expression tree. First we have to check whether the expression is numerical, i.e. contains no identifiers. This is what the function `isNumerical` does.

```
int isNumerical(ExpTree tr) {
  assert(tr!=NULL);
  if (tr->tt==Number) {
    return 1;
  }
  if (tr->tt==Identifier) {
    return 0;
  }
  return (isNumerical(tr->left) && isNumerical(tr->right));
}
```

Then we can compute the value of a numerical expression:

```
/* precondition: isNumerical(tr)) */
double valueExpTree(ExpTree tr) {
  double lval, rval;
  assert(tr!=NULL);
  if (tr->tt==Number) {
    return (tr->t).number;
```

```
  }
  lval = valueExpTree(tr->left);
  rval = valueExpTree(tr->right);
  switch ((tr->t).symbol) {
  case '+':
    return (lval + rval);
  case '-':
    return (lval - rval);
  case '*':
    return (lval * rval);
  case '/':
    assert(rval != 0);
    return (lval / rval);
  default:
    abort();
  }
}
```

Finally we can demonstrate the workings of what we have done:

```
void prefExpTrees() {
  char *ar;
  List tl, tl1;
  ExpTree t = NULL;
  printf("give a prefix expression: ");
  ar = readInput();
  while (ar[0] != '!') {
    tl = tokenList(ar);
    printf("the token list is ");
    printList(tl);
    tl1 = tl;
    if (treePrefixExpression(&tl1,&t) && tl1 == NULL) {
         /* there should be no tokens left */
      printf("in infix notation: ");
      printExpTreeInfix(t);
      printf("\n");
      if (isNumerical(t)) {
        printf("the value is %g\n",valueExpTree(t));
      } else {
        printf("this is not a numerical prefix expression\n");
      }
    } else {
      printf("this is not a prefix expression\n");
    }
    freeExpTree(t);
    t = NULL;
    freeTokenList(tl);
    free(ar);
    printf("\ngive a prefix expression: ");
    ar = readInput();
  }
  free(ar);
  printf("good bye\n");
}
```

A dialogue with the function `prefExpTrees`:

```
give a prefix expression:  +-*/2 3 4 5   6
the token list is + - * / 2 3 4 5 6
in infix notation: ((((2 / 3) * 4) - 5) + 6)
the value is 3.66667

give a prefix expression: 2 + 3 - 4
the token list is 2 + 3 - 4
this is not a prefix expression

give a prefix expression:  + a * 3 x
the token list is + a * 3 x
in infix notation: (a + (3 * x))
this is not a numerical prefix expression

give a prefix expression: !
good bye
```

2.6 Exercises

Exercise 2.1. Perform `preOrder`, `postOrder` and `inOrder` on the binary search tree in Figure 2.3 and on the heap in Figure 2.6. When doing so, read `printf("%d",t->item)` for `visit(t)`.

Exercise 2.2. Below several possible actions on a tree are given. Indicate which of the three traversals (preorder, postorder or inorder) is to be used.

1. compute the value of an arithmetical expression that is given as an expression tree;
2. print an arithmetical expression that is given as an expression tree;
3. compute the storage use of all folders in a hierarchical file directory;
4. print a document consisting of sections and subsections that are stored in a tree structure;
5. remove a tree and free the memory used.

Exercise 2.3. Define the function `void freeTree(Tree t)` to free the memory used by tree `t`.

Exercise 2.4. Define C functions that have the following result when executed on a binary tree in pointer representation containing integers. You are not supposed to define a single function that can do everything, but several functions.
a. a copy of the tree;
b. the height of the tree;
c. the number of nodes in the tree;
d. the highest position that contains a node;
e. an answer to the question whether it is a search tree.

Exercise 2.5. Define a function `Tree subtree(Tree t, int n)` that, given a binary tree in pointer representation and a number n, yields the subtree that has the node on position `n` as its root. You may assume that tree `t` contains a node on position `n`.
Hint. Focus on a recursive approach. How may the node with number $n/2$ help you in finding the node with number n?

Exercise 2.6. Rewrite the definitions of `preOrder`, `postOrder`, `inOrder` for binary trees in array representation. You may assume that the nodes contain non-negative numbers; the arrays contain the value -1 in the fields that correspond with non-existing nodes.

Exercise 2.7. The function `addInSearchTree` is defined in Section 2.2. Consider the following alternative definition:

```
Tree addInSearchTree(Tree t, int n) {
  if (t == NULL) {
    return newTree(n,emptyTree(), emptyTree());
  } else if (t->item == n) {
    return t;
  }
  return addInSearchTree( ( n <= t->item ? t->leftChild
                                         : t->rightChild ), n);
}
```

What is wrong with this alternative definition?

Exercise 2.8. **a.** Define a function `treeToArray` that converts a binary tree in pointer representation into a binary tree in array representation, and conversely a function `arrayToTree` that converts a binary tree in array representation into a binary tree in pointer representation. You may assume that the nodes contain only non-negative numbers.
b. How would you adapt your solution when nothing is given about the values in the nodes (except the type `int`)?

Exercise 2.9. Convert the pseudocode for RemoveFromSearchTree and Successor in Section 2.2 to functions in C.

Exercise 2.10. Given a sorted integer array `ar`, construct a balanced search tree that contains the integers in `ar`. You may assume that all integers in `ar` are different.

Exercise 2.11. Define a C function `void doubleHeapSize (Heap *hp)` that doubles the size of a heap. Use the type `Heap` given in Section!2.3.

Exercise 2.12. Define a C function `void upheap (Heap *hp, int n)` that can be used in the function `enqueue` given in Section 2.3. You may use the function `swap` as defined in 4.1.6 in the lecture notes on *Imperative Programming*:

```
void swap(int *pa, int *pb) {
  int h = *pa;
  *pa = *pb;
  *pb = h;
}
```

Exercise 2.13. Define a C function `void downheap (Heap *hp, int n)` that can be used in the function `removeMax` given in Section 2.3. You may use the function `swap` as given in the previous exercise.

Exercise 2.14. Define a C function `void heapSort(int n, int ar[])` that uses a heap to sort array `ar` with length `n`.

Exercise 2.15. **a.** Adapt the algorithm SearchInTrie in Section 2.4 so that it checks whether w occurs as a prefix of a word in T.
b. Adapt the algorithm SearchInTrie so that it checks whether a prefix of w occurs in T.

Exercise 2.16. Describe in pseudocode an algorithm that converts a standard trie into a compressed trie.

Exercise 2.17. Describe in pseudocode an algorithm that builds a standard suffix trie from a given string S. You may assume that the last character in S does not occur elsewhere in S: as a consequence, a suffix of S cannot be the prefix of another suffix of S.

Exercise 2.18. Describe in pseudocode an algorithm that simplifies expression trees according to the following rules:

$0 * E$ and $E * 0$ are simplified to 0;
$0 + E$, $E + 0$, $E - 0$, $1 * E$, $E * 1$ and $E/1$ are simplified to E.
Here E is an arbitrary expression.

Exercise 2.19. Define a C type that is appropriate for standard tries.

Chapter 3

Graphs

In the previous chapters we have presented the data structures lists and trees. We now continue with a more general data structure: graphs. Like a tree, a graph consists of nodes and edges. Unlike a tree, a graph has no root that serves as a starting point; moreover, a graph may contain *cycles*, paths that return to their starting point. See Figure 3.1.

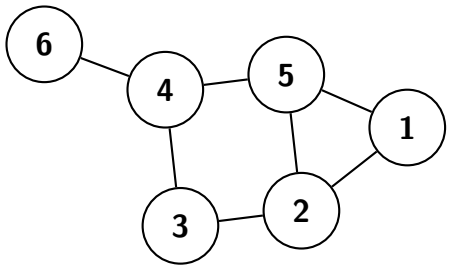

Figure 3.1: A simple undirected graph with 6 nodes and 7 edges. 1 and 2 are adjacent, 1 and 3 are not adjacent. (1,2,3,2,5) is a path. (1,2,3,4,6) is a simple path, (1,2,5,4,3,2,5,1) is a cycle, (1,2,5,1) is a simple cycle. This graph is simple and connected.

We start with introducing some notions. A *graph* $G = (V,E)$ consists of a collection V of *nodes*[1] and a collection E of *edges* that connect nodes. When edge e connects the nodes v and w, we say that e is *incident* with v and w, and that v and w are *neighbours* (or: are *adjacent*). Two edges are called *parallel* when they are incident with the same nodes. An edge is called a *loop* when it connects a node with itself. A graph is *simple* when it contains no loops and no parallel edges.

A *path* is a sequence $(v_0,v_1,\ldots,v_n)$ of nodes where every two subsequent nodes v_i, v_{i+1} are neighbours (i.e. are connected by an edge). The length of a path is the number of its nodes minus 1, so (v) is a path with length 0. A path with length > 0 is called a *cycle* when the first and the last node are equal. A path is called *simple* when all nodes are different. A cycle is called simple when all nodes except the first and the last are different, and when it contains at least three nodes. The last condition excludes trivial cycles of the form (v,w,v).

A graph is called *connected* when every pair of nodes is connected by a path.

3.1 The start of graph theory

Before we continue, it is interesting to look how graphs and graph theory were 'invented'. It is an instructive example of how an abstraction, developed for the solution of a single problem, can lead to a completely new theory which can be used for many more problems.

Around 1730, the great Swiss mathematician Leonard Euler (1707 - 1783) was professor at the University of St. Petersburg. He heard about the *Problem of the Seven Bridges of Königsberg*. At that time, Königsberg was a German town and the capital of Prussia.

[1] Nodes in a graph are also called *vertices*, with the singular form *vertex*. This is Latin for *peak*.

Nowadays it is called Kaliningrad, situated in the Russian exclave between Poland and Lithuania. Figure 3.2 shows the city.

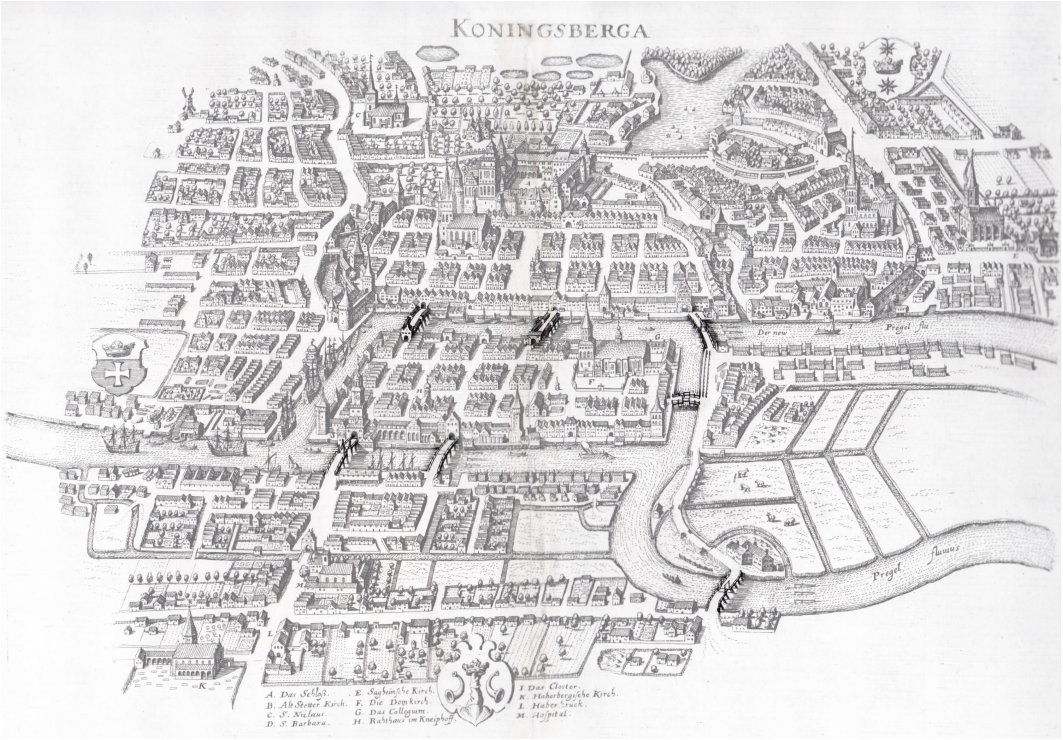

Figure 3.2: The city of Königsberg with seven bridges.

The river Pregel passes through the city of Königsberg. There are two islands in the river. Seven bridges connect them to each other and to the mainland.

The citizens of Königsberg liked to make a walk through the city and cross the seven bridges. They tried to make a walk *in which every bridge is crossed exactly once*. Finding such a walk is known as the Problem of the Seven Bridges of Königsberg. Nobody ever found such a walk, and until Euler nobody could explain why.

How did Euler solve this problem? His solution consists of two steps: first an abstract formulation of the problem, then a clever argument. Euler observed that the essence of the problem consists of pieces of land and bridges. They can be captured in Figure 3.3.

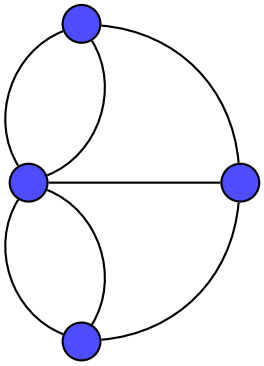

Figure 3.3: Euler's graph representation of Königsberg.

The two islands and the two pieces of mainland are reduced to *nodes*, and the bridges are

reduced to *edges*. Thus the city is transformed into a graph consisting of four nodes and seven edges. A walk through the city corresponds with a path in the graph. 'Passing each bridge once' corresponds to 'passing each edge once'.

With this reduction of the problem to a graph, Euler analysed the problem as follows. Suppose there is a path (walk) in which we cross each edge once. In any node we pass on our way, we arrive n times and we leave n times: in such a node an *even* number of edges comes together (viz. $2n$). When the start node and the end node are equal, we have an even number of edges there, too. When the start node and the end node are different, we have an *odd* number of edges in these nodes.

Now it is easy to solve the problem. If there is a path that crosses each edge exactly once, then at most two nodes have an odd degree. However, all four nodes have an odd degree! So there cannot be a path that crosses each edge exactly once. We conclude that the Problem of the Seven Bridges of Königsberg has no solution.

In honour of Euler, we call a path in which every edge occurs exactly once an *Euler path*. An *Euler cycle* is an Euler path with identical begin and end node. We have the following theorem, generalizing Euler's solution of the Seven Bridges of Königsberg.

> A connected graph has an Euler cycle if and only if all nodes have an even degree.
> A connected graph has an Euler path if and only if at most two nodes have an odd degree.

The 'only if' part is proved by the argument of Euler given above. The 'if' part is somewhat harder to prove.

Question: what can we say about Euler paths in a connected graph with exactly 1 node with odd degree?

3.2 More notions related to graphs

The graphs described in the previous section are *undirected* graphs: their edges have no direction. In a *directed* graph the edges do have a direction: one of the incident nodes is called the initial node, the other the terminal node. See Figure 3.4. In a directed graph, two edges are parallel when they have the same initial node and the same terminal node.

A (directed or undirected) graph is called *simple* when it has no loops and no parallel edges. In a simple graph, we can denote an edge by the two nodes incident with it. Observe that the Königsberg graph in Figure 3.3 is *not* simple. From now on, we shall encounter only simple graphs.

When a graph $G = (V,E)$ is simple, it has no parallel edges and we may assume that $E \subseteq V \times V$. In mathematical terms: E is a binary relation on V.

A connected graph without simple cycles is called a *tree*. A difference with the trees that we have seen before is that there is no root indicated, and that the edges do not have a direction. Trees with a root are called *rooted* trees. It is not hard to see that every node in a tree can act as its root.

A *subgraph* of a graph $G = (V,E)$ is a graph $G' = (V',E')$ with $V' \subseteq V$ and $E' \subseteq E$. When we have $V' = V$, we call G' a *spanning* subgraph. A special case is the *spanning tree* of a connected graph: it is a minimal spanning subgraph that is a tree.

A *weighted* graph is a graph where all edges contain a number: the *weight* of that edge. A weighted graph can be used to represent a network where the weights indicate e.g. the length of an edge, or its capacity. See Figure 3.4.

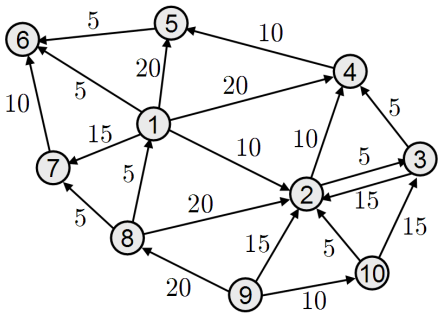

Figure 3.4: A directed and weighted graph.

3.3 Representation of graphs in C

Suppose we want to develop some C functions that deal with graphs, e.g. to traverse a weighted graph with data stored in the nodes. This can be done as follows. We use the numbers $0, \ldots, N-1$ for the nodes. Then an array `data` can be used for the data, and also an array `visited` to indicate whether a node has been visited. To represent the neighbours of the nodes, we can use an array `neighbourList` of linked lists. Every item in the linked list `neighbourList[n]` contains an integer `node`, an integer `weight` that indicates the weight of the edge between `n` and `node`, and a pointer `next` to the next item.

```
#define N 10 /* graph size */

int data[N];
int visited[N];

typedef struct ListNode *ListPointer;

typedef struct {
  int node;
  int weight;
  ListPointer next;
} ListNode;

ListPointer neighbourList[N];
```

3.4 Searching in a graph

Given a graph, it may be useful to have an algorithm to traverse it and to visit all its nodes and edges. In the case of trees, we have seen several ways to do this: preorder, inorder (only for binary trees) and postorder. A graph has in general less structure than a tree, and hence graph traversal is somewhat more complicated than tree traversal. We will consider two traversal strategies: DFS (*depth-first search*) and BFS (*breadth-first search*). Besides a visit to all nodes and edges, these algorithms also provide a spanning tree of the graph, provided the graph is connected. Is the graph not connected, then DFS and BFS are restricted to a *connected component* of the graph: all nodes and edges that are reachable from the starting point.

3.5 Depth-First Search

To explain the idea, we imagine graph traversal as the exploration of a maze. The edges are paths in the maze, the nodes are (cross)points where one or more paths meet. How are we to explore the maze without getting lost or running in circles? For that purpose, we will use a rope and a can of paint. We fix the rope at the starting point, so that we can always return. We use the paint to mark the paths and cross-points that we have visited. So we start walking with the loose end of the rope and the can of paint. We only take edges where we have not been before, i.e. that are unmarked. We mark every crosspoint that we pass. When we encounter a crosspoint where we have not been before, we mark the path that led us to it with NEW and we continue. However, when arriving on a crosspoint where we have been already, we mark the path to it with OLD and we walk back. When we are on a crosspoint where all paths are marked, we use the rope to return to the previous crosspoint. We continue until we are back at the starting point and all paths from it are marked.

Now we translate this maze exploration method into a graph algorithm. The role of the rope is played by the recursion stack, and instead of paint we use labels to mark the nodes and edges. This leads to the following algorithm DFS for depth-first search:

```
algorithm DFS(G,v)
    input connected graph G with node v
    result labelling of the edges of G with NEW and OLD;
        the edges with label NEW form a spanning tree of G,
        and all nodes have been visited (and labeled VISITED)
    give v the label VISITED
    forall e incident with v do
        if e has no label then
            w ← the other node incident with e
            if w has no label then /* new node discovered! */
                give e the label NEW
                DFS(G,w)
            else /* w has label VISITED */
                give e the label OLD
```

Now we can also explain the name "depth-first search": the algorithm traverses (i.e. goes down) edges as long as possible, until it meets a node it has visited before.

About the time complexity we can say the following. Every node corresponds with one recursive call of the algorithm. In a node, we inspect all incident edges one by one. When we use adjacency lists in the representation of the graph (as described above), we can find an uninspected edge in $\mathcal{O}(1)$ time: it is the edge represented by the next node in the adjacency list. Every edge is inspected twice, viz. once from every node incident with it. So every edge takes $\mathcal{O}(1)$ time. Altogether the time complexity is in $\mathcal{O}(n+m)$, where n is the number of nodes and m the number of edges. This is the best we can expect, since we have to visit every node and every edge.

Depth-first search is an efficient way for the systematic exploration of a graph. As such it has many applications. We mention a few:

- finding a path between two nodes;
- checking whether a graph is connected;
- checking whether a graph contains a cycle;
- finding a spanning tree for the graph.

3.6 Breadth-First Search

Breadth-first search is not as greedy as depth-first search: first it systematically treats all nodes that are one step away from the starting node, then all nodes that are two steps away, and so on. This is realized via a queue: newly discovered nodes are enqueued, and when a node u is dequeued all its incident and unexplored edges are explored. This queue is also called the *fringe*.

```
algorithm BFS(G,v)
    input connected graph G with node v
    result labelling of the edges of G with NEW and OLD;
        the edges with label NEW form a spanning tree of G,
        and all nodes have been visited (and labeled VISITED)
    Q ← empty queue of nodes
    give v the label VISITED
    enqueue(v)
    while Q not empty do
        u ← dequeue()
        forall e incident with u do
            if e has no label then
                w ← the other node incident with e
                if w has no label then /* new node discovered! */
                    give e the label NEW
                    give w the label VISITED
                    enqueue(w)
                else /* w has label VISITED */
                    give e the label OLD
```

The time complexity of breadth-first search is in $\mathcal{O}(n+m)$: every node and every edge takes only $\mathcal{O}(1)$ time.

Due to the systematic way in which breadth-first search begins with nearby nodes, we have the following property:

BFS finds minimal *paths from the starting node to the other nodes.*

In contrast, note that DFS does not always find a minimal path. On the other hand, BFS has the disadvantage that more memory is needed for the queue.

3.7 Dijkstra's shortest path algorithm

We now present a famous and important algorithm: Dijkstra's algorithm for finding the shortest path between two nodes in an undirected weighted graph. The weights are non-negative numbers that represent the length of the edges.

How are we to do this? First we reformulate the problem slightly. We have a graph G with some node v, and we want to know for every node w in G the length of a shortest path from v to w: we call this the *distance* between v and w. We shall develop an algorithm to solve this problem. In an exercise, you will be asked to adapt the algorithm, so that it does not only compute the distance between v and the other nodes w, but also indicates what the shortest paths are. And this solves our original problem.

So now we have the problem of finding the distances between v and the other nodes. We aim at a stepwise solution: first we determine the distance to nodes that are close to v, and we use these to determine distances to nodes that are farther away.

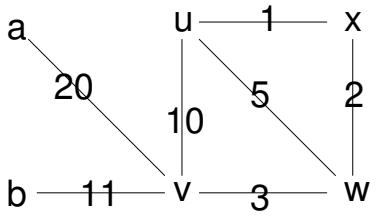

Figure 3.5: A weighted graph.

Let us perform an experiment, considering a part of a graph near the starting point v: see Figure 3.5. First we observe that the distance from v to v equals 0. Now we look at the nodes adjacent to v, and we take the node w for which the length of edge (v,w) is minimal, i.e. 3. Now we already know that 3 is the distance between v and w, for there cannot be a shorter path between v and w.

How can we determine the distance to a third node? We might look at the second shortest edge incident with v, that is edge (v,u) with length 10. Do we know that 10 is the distance between v and u? No, we do not: there is an edge (w,u) with length 5, hence a path (v,w,u) with length 8. Does this imply that 8 is the distance between v and u? Again no: there is the edge (w,x) with length 2 and an edge (x,v) with length 1, so (v,w,x,u) has length 6.

It is time for another approach. Let us see what we really know: the distance of v is 0, the distance of w is 3. Now we look at all nodes that can be reached in one step from the set {v,w}. For these nodes we compute the *pseudo-distance*, i.e. the shortest path length based on our present knowledge. Then we choose the node with the least pseudo-distance, for that pseudo-distance is the real distance: minimality excludes the existence of a shorter path.

Let us generalize this method for finding the distance of the third node. In general, we have a collection of nodes for which the true distance has been found. We call this collection the *cloud*. We start with a cloud containing only the start node v, with distance 0. We repeatedly add a node to the cloud when we have found its true distance. This continues until all nodes are in the cloud.

The node to be added to the cloud is found as follows. We consider all nodes outside the cloud that are reachable from the cloud in one step. For all these nodes, we compute the pseudo-distance to v. See Figure 3.6 which extends Figure 3.5. Then we choose the node with the smallest pseudo-distance, because that pseudo-distance is the real distance to v. In Figure 3.6, this is node n. We add n to the cloud, and we update the pseudo-distance of nodes connected to n. In Figure 3.6, this is node r.

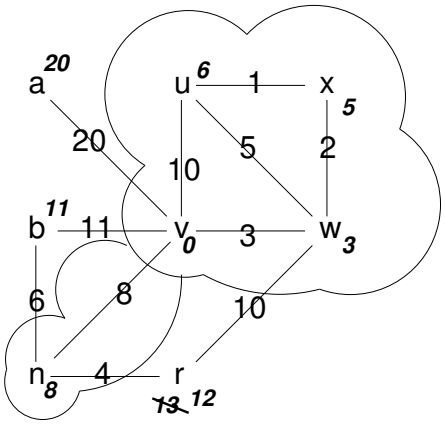

Figure 3.6: A weighted graph during the application of Dijkstra's shortest path algorithm. The nodes are labeled with their pseudo-distance from v. The cloud is being extended with node n, and the pseudo-distance of node r is being updated from 13 to 12.

The above is a verbal description of Dijkstra's algorithm. Its conversion to pseudocode is as follows. We do not work with the cloud as introduced above, but with its complement S ('sky' as opposed to cloud). One can also think of S as a ToDo list: these are the nodes we still have to deal with.

algorithm Dijkstra(G, v)
  input connected weighted graph G with node v
  output function d yielding for every node the length of a shortest path to v
  S ← nodes(G) /∗ initialise ToDo list S, the complement of the cloud ∗/
  forall u ∈ nodes(G) **do**
    d[u] ← **if** u=v **then** 0 **else** ∞ /∗ initialise d ∗/
  while S is not empty **do**
    u ← node in S such that d[u] is minimal
    remove u from S
    forall z ∈ S with (u,z) ∈ edges(G) **do** /∗ the relaxation step ∗/
      d[z] ← min(d[z], d[u] + weight[u][z])
  return d

The implementation of Dijkstra's algorithm is rather straightforward, except for the first statement in the **while** body: the selection of the node in S with a minimal value of d. The naive way to do this is: inspect all nodes in S to find out which nodes has the least d-value. But this takes in general $\mathcal{O}(n)$ steps, which is not very efficient, for we have to do this n times. A better approach is: use a heap to implement an inverted priority queue (with function RemoveMin instead of RemoveMax) that stores the nodes in S. Then the node with the least d-value can be obtained in $\mathcal{O}(\log(n))$ time. The initialization of the heap is straightforward, since we begin in the situation where one node has priority 0 and all others have priority ∞. Moreover, in the relaxation step we may have to reposition nodes in the heap because their d-value has changed.

Let us look at the time complexity of Dijkstra's algorithm. The body of the **forall** loop has complexity $\mathcal{O}(1)$ and is performed n times, so $\mathcal{O}(n)$ in total. The same holds for the initialization of the heap. In the **while** body, RemoveMin is performed n times, which leads to $\mathcal{O}(n \log(n))$. In the relaxation step, d is recomputed for every edge, so $\mathcal{O}(m)$ times. The computation takes $\mathcal{O}(1)$ time, but the repositioning in the heap will take $\mathcal{O}(\log(n))$ time. So the relaxation step will take $\mathcal{O}(m \log(n))$ steps. Adding all up, we end with $\mathcal{O}((n+m)\log(n))$.

We can sharpen this upper bound by using the fact that $m \geq n-1$ in a connected graph. As a consequence, n is in $\mathcal{O}(m)$. This leads to the time complexity of $\mathcal{O}(m \log(n))$ for Dijkstra's algorithm.

3.8 A variant: the A* algorithm

Suppose we want to find in a weighted graph the shortest path between v and w. We may use Dijkstra's algorithm, but it will determine the shortest path to v for *all* nodes, while we only want to go from v to w. Isn't there a faster way to do this?

Yes, there is, provided we have a hunch how far we are from w. More precisely: when we have a function $h : \mathit{nodes}(G) \to \mathbb{N}$ where $h[u]$ indicates a lower bound for the distance to w. We call this a *heuristic* function. An example of a heuristic function is the distance between u and w as the crow flies, i.e. along a straight line. In general, the heuristic function provides a kind of preference in the direction towards the goal node w.

We use h when choosing a new node u to add to the cloud. Instead of the smallest d-value (the length of a shortest path from v to u), we consider the smallest value of $d + h$: the estimated length of a path from v via u to w. Recall that $d[u]$ is the length of a shortest path from v to u, and $h[u]$ is the estimated length of a shortest path from u to w.

This algorithm is known under the name A* ('A star'). In pseudocode:

```
algorithm A*(G,h,v,w)
    input connected weighted graph G with nodes v,w
        and heuristic function h : nodes(G) → int
        with: h[u] ≤ (length shortest path from u to w)
    output length of a shortest path from v to w
    S ← nodes(G) /* S is initialized */
    forall u ∈ nodes(G) do
        d[u] ← if u=v then 0 else ∞ /* d is initialized */
    while S not empty do
        u ← node in S with minimal value of d + h
        if u = w then
            return d[u]
        remove u from S
        forall z ∈ S with (u,z) ∈ edges(G) do
            d[z] ← min(d[z], d[u] + weight[u][z])
```

3.9 Exercises

Exercise 3.1. **a.** In Figure 3.7, a house with five rooms and 16 doors is given. Is it possible to walk inside and outside the house in such a way that you pass exactly once through every door? Solve this problem by drawing and analysing the corresponding graph.

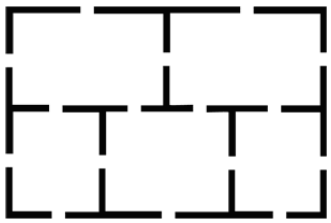

Figure 3.7: The five rooms puzzle.

b. Apply Depth-first Search to the graph in Figure 3.8. Start in the uppermost node. Number the nodes in the order that the algorithm visits them. Label the edges with N (for NEW) and O (for OLD). Question: Is the outcome unique? Why could your colleague find a different numbering?

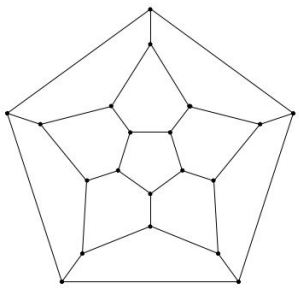

Figure 3.8: A graph.

c. Do the same with Breadth-first Search.

Exercise 3.2. **a.** Adapt Depth-First Search to obtain an algorithm FindPath(G,v,w) that, given a graph G and two nodes v and w, returns a path in G from v to w if it exists.
b. Adapt Breadth-First Search to obtain an algorithm FindShortestPath(G,v,w) that, given a graph G and two nodes v and w, returns a path in G from v to w with minimal length (number of edges) if it exists.

Exercise 3.3. **a.** Replace in the algorithm BFS the queue by a stack (and enqueue/dequeue by push/pop, of course). The resulting algorithm is almost DFS, but what is the difference?
b. Modify the algorithm of (a) so that it behaves like DFS.

Exercise 3.4. **a.** Give an algorithm that transforms a path in a graph into a *simple* path, with the same begin and end node as the given path. For example, consider Figure 3.9.

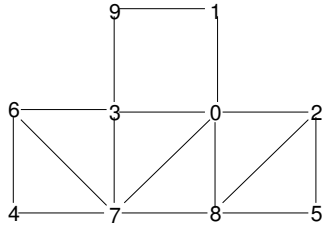

Figure 3.9: A graph.

An example of a non-simple path in this graph:

1 9 1 9 3 9 3 9 1 9 3 0 3 0 3 0 2 5 8 2 5 8 0 7 6 7 8 5 8 7 0 3 0 7 6 4

The corresponding simple path is 1 9 3 0 7 6 4.
What is the time complexity of your algorithm?
b. Is this also possible for cycles, i.e. can every cycle be transformed into a simple cycle?

Exercise 3.5. We consider simple unweighted graphs G with nodes v, w and edge e. The length of a path is the number of edges in it. Give algorithms in pseudocode.
a. Give an algorithm FindShortestPath that computes the length of a shortest path between v and w. What is the time complexity of your algorithm?
b. Use FindShortestPath in an algorithm FindShortestCycle that computes the length of a shortest cycle containing edge e. What is the time complexity of your algorithm?
c. Use FindShortestCycle in an algorithm FindShortestCycle2 that computes the length of a shortest cycle containing node v. What is the time complexity of your algorithm?
d. Use FindShortestCycle in an algorithm FindShortestCycle3 that computes the length of a shortest cycle in graph G. What is the time complexity of your algorithm?

Exercise 3.6. We consider a graph $G = (V,E)$. The collection V of nodes is a subset of the n-by-n-square $\{(i,j) \mid 0 \leq i,j < n\}$ of grid points. The collection E of edges consists of all pairs $((i,j),(i',j'))$ of nodes (i,j) and (i',j') in V with distance 1 (so $i = i'$ and $|j - j'| = 1$, or $j = j'$ and $|i - i'| = 1$). The length of a path is the number of edges in it. It is given that (0,0) and $(n-1,n-1)$ are in G.
a. Write a function in C that computes the length of a shortest path in G from (0,0) to $(n-1,n-1)$.
b. Extend the function so that it also provides a shortest path.

Exercise 3.7. Extend Dijkstra's algorithm so that it yields, given v and w, a shortest path from v to w (and not only its length).

Exercise 3.8. Define the *width* of a path as the minimal weight of its edges. Develop a variant of Dijkstra's algorithm that finds a path with maximal width between the nodes v and w.

Appendix A

More about C

A.1 Main

In the course Imperative Programming you have learned to define the function `main` by

```
int main(int argc, char *argv[]) {
  ...
}
```

You may have found out that this can be shortened to `main ()`. Two remarks about this

- When the return type is missing in a function definition, the default type `int` is taken.
- The parameters `argc` and `argv` can be used to process arguments from the command line. The value `argc` (*argument count*) indicates the number of arguments and `argv` (*argument vector*) is a pointer to an array that contains the arguments.

More about this in *Kernighan & Ritchie* 5.10, p. 114.

A.2 Sequential evaluation

At first sight, the operators `&&` and `||` correspond with the logical operators $\wedge$ and $\vee$. There is an important difference, however.

In C, an expression `(E1 && E2)` is evaluated from left to right, so it begins with the evaluation of `E1`. However, it might take a shortcut: `E2` is only evaluated when that is needed to determine the value of `(E1 && E2)`. As a consequence, when `E1` yields the value 0 (false), we know that `(E1 && E2)` will also yield the value 0 so we do not need to evaluate `E2`. The conjunct `E2` is only evaluated when the evaluation of `E1` yields 'true' (i.e. a non-zero value).

Similarly for `(E1 || E2)`: when `E1` yields 'true', evaluation of `E2` is unnecessary, it is only evaluated when `E1` yields 0.

We often use *sequential evaluation* (or: *short-circuit evaluation*), e.g. in the condition

```
lp != NULL  &&  lp->item == 3
```

Without sequential evaluation this would result in a `Segmentation fault` when `lp` is a NULL-pointer, because then `lp->item` would not exist. Recall that `lp->item` is the same as `(*lp).item`.

A.3 Value transfer

Programming in C mainly consists of defining functions, and during execution of a program most of the work is done by and in functions. We have a closer look at a number of ways in which a value, computed during the execution of a function, can be made available to other parts of the program. We distinguish three styles: functional, procedural and global.

Functional: via the result of a function. This is the most straightforward way: with the `return` statement, a value computed inside the function is returned. In general this looks as follows:

```
int f(int m) {
  return 2*m;
}
...
x = f(5) + 1;
/* now x has the value 11 */
```

Examples of the functional style can be found e.g. in Section 1.3. The functional style does not work, however, when more than one value from inside the function is to be used outside. We might combine two values in a `struct pair`, but we may also choose the next method.

Procedural: via a reference parameter. One of the arguments of the function is a pointer, and will point to the value computed inside the function.

```
int n;
void f(int m, int *ip) {
  *ip = 2*m;
}
...
f(5,&n);
x = n + 1;
/* now x has the value 11 */
```

This style is less common, but may be handy when more than one value has to be passed to outside the function.

The use of an array is closely related to this style:

```
int b[10];
void f(int m, int a[]) {
  a[7] = 2*m;
}
...
f(5,b);
x = b[7] + 1;
/* now x has the value 11 */
```

We used the procedural style in Sections 1.1 and 1.2, and also in Section 1.4.6.

Global: via a global variable. Here the value computed inside the function is assigned to a global variable. Contrary to the local variables, the global variables remain in existence when the execution of a function call ends.

```
int n;
void f(int m) {
  n = 2*m;
}
...
f(5);
x = n + 1; /* this will set x to 11 */
```

Alert!

The global variable style is *not recommended*. When global variables are used inside a function, it is no longer possible to understand the function solely from its definition: you also have to know how the global variables are used elsewhere in the program. Hence global variables can reduce readability and understandability of function definitions. Unnecessary use of global variables is considered *bad programming*.

Tip

Avoid global variables! If you only need to *read* a variable in many places, but never modify it while your program runs, then use a global *constant* with `const` or `#define`.

A.4 Segmentation faults

Segmentation faults are nasty when their cause is hard to find. The compiler only says `Segmentation fault`, without any information about what went wrong. Fortunately, there are programs that can help. Two well-known examples are `gdb` and `valgrind`.

Here we only elaborate on `valgrind`. Feel free to work out yourself how `gdb` works, you can find its manual at `https://sourceware.org/gdb/current/onlinedocs/gdb/`.

Consider the following program that is guaranteed to yield a segmentation fault, since it tries to write to *read-only* memory (see Section A.6 for more):

```
int main(int argc, char **argv) {
  char *s = "this string is stored in read-only memory";
  s[0] = 'T';
  return 0;
}
```

Compiling and executing this program yields `Segmentation fault` (as expected). Now we compile the program again with the additional option `-g`. This adds debugging information that `valgrind` uses. Then we start the program with `valgrind a.out`.

```
==20344== Process terminating with default action of signal 11 (SIGSEGV)
==20344==  Bad permissions for mapped region at address 0x4005CC
==20344==    at 0x4004D4: main (main.c:3)
==20344==
==20344== HEAP SUMMARY:
==20344==     in use at exit: 0 bytes in 0 blocks
==20344==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==20344==
==20344== All heap blocks were freed -- no leaks are possible
==20344==
==20344== For counts of detected and suppressed errors, rerun with: -v
==20344== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
Segmentation fault
```

What does this tell us? We now know exactly where the segmentation fault occurred! When the program tried to execute line 3 (`main.c:3`), this led to the signal `SIGSEGV`: segmentation fault caused by an attempt to write in read-only memory! With this information, it is not hard to find out how to adapt the program. Observe that `valgrind` also provides a `HEAP SUMMARY`[1] that can be used to detect memory leaks. See Appendix A.5 below.

Tip

Problem with `Segmentation fault`? Use `valgrind`!

For further information, see `http://www.valgrind.org/docs/manual/manual.html`.

A.5 Memory reservation and memory leaks

We recall from Section 5.1.7 about Dynamic memory allocation of the Lecture Notes *Imperative Programming* the following. The function `malloc` can be used to allocate memory, but it does not initialize the values to 0. When that is required, use `calloc`. The function `realloc` is used to resize the allocated memory, in combination with `memset` when initialization to 0 is required. The function `free` is a special instance of `realloc`, viz. with second parameter 0.

With `malloc` and `calloc`, the creation of a dynamic one-dimensional array is rather straightforward. But what to do when you need a two- or more-dimensional dynamic array? This is the subject of one of the exercises at the end of the Appendix.

The allocation of memory can lead to *memory leaks*. What is a memory leak? This term is used to describe a situation where a program keeps memory allocated without being able to reference it. This is a common error, because it is not easy to keep track of all memory usage in a program. Modern programming languages like e.g. Java have a *garbage collector*, a routine to clean up unused memory automatically. C, however, does not have this functionality. As a consequence, the programmer has to keep track of memory usage, so that memory is freed whenever it is no longer in use.

Here is an example of a program that leaks memory:

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

int main(int argc, char *argv[]){
  int i;
  double j = 1;
  char *text;
  for (i=0; i < 100; i++) {
    text = malloc(100 * sizeof(char));
    assert(text != NULL);
    sprintf(text, "2 to the power %d is %lf", i, j);
    /* print a string to a char array */
    printf("%s\n", text);
    /* print on the screen */
    j = j * 2;
  }
  return 0;
}
```

[1] The heap mentioned here has nothing to do with the heap data structure presented in Section 2.3.

Let us execute this program via `valgrind` (see Appendix A.4). So we add the option `-g` when compiling, and then we execute with `valgrind a.out`. Then we obtain the following information:

```
==6990== HEAP SUMMARY:
==6990==     in use at exit: 10,000 bytes in 100 blocks
==6990==   total heap usage: 100 allocs, 0 frees, 10,000 bytes allocated
==6990==
==6990== LEAK SUMMARY:
==6990==    definitely lost: 10,000 bytes in 100 blocks
==6990==    indirectly lost: 0 bytes in 0 blocks
==6990==      possibly lost: 0 bytes in 0 blocks
==6990==    still reachable: 0 bytes in 0 blocks
==6990==         suppressed: 0 bytes in 0 blocks
==6990== Rerun with --leak-check=full to see details of leaked memory
```

So we have a memory leak. This is not surprising when we inspect the code: in each iteration of the `for` loop, `malloc` allocates 100 bytes for `text`, but this memory is not freed. So after 100 iterations we have allocated 10 000 bytes. Only the 100 bytes that were allocated in the last iteration are accessible, viz. via the variable `text`. So we cannot free all memory at the end of the program. This has to be done earlier: at the end of the loop body (after the `printf` statement, before the closing bracket of the `for` loop) we should add `free(text)`. When we do this and check the modified program with `valgrind`, we get

```
==7066== HEAP SUMMARY:
==7066==     in use at exit: 0 bytes in 0 blocks
==7066==   total heap usage: 100 allocs, 100 frees, 10,000 bytes allocated
==7066==
==7066== All heap blocks were freed -- no leaks are possible
```

The general rule reads:

Tip

Every `malloc` and every `calloc` requires a corresponding `free`!

To check whether you managed to apply this important principle, we suggest:

Tip

Use `valgrind` to check for memory leaks.

Alert!

For some test cases the submission system Themis will also run your program in `valgrind`. You will receive the exit code 111 if there are any problems.

When a program ends, all memory it uses is freed by the operating system, including leaking memory. So why bother about freeing memory in a program that does not run for a long time? The answer is: mainly for learning purposes. A professional programmer should be resource-aware: allocate only what is needed for your program, as long as it is needed. For a single standalone program with restricted running time, a small memory leak will often do no harm. However, when many programs are to run concurrently for longer periods, even the smallest memory leak may lead to increasing loss of available memory and hence to performance degradation.

A.6 String constant table

Consider the following two program fragments:

```
char *string = "hello";
printf("%s\n", string);
free(string);
```

and

```
char *string = malloc(6*sizeof(char));
/* sixth position for the null character \0 */
strcpy(string, "hello");
printf("%s\n", string);
free(string);
```

At first sight, these fragments do the same. However, the first yields a `Segmentation fault`, while the second terminates successfully. Why? The reason for this requires some knowledge of the operating system.

During the compilation of a program, the C compiler generates machine code and a table with all strings in the program (in our example program only the string `"hello"`). All these strings have type `const char *` and cannot be modified. In the first program, `string` gets the location of the constant string `"hello"` in the string table. So `free(string)` tries to free part of the string table, which is not allowed and leads to a segmentation fault.

The second variant uses the function `strcpy` to make a copy of the string `"hello"` and to write this copy on the memory location string points to. Now `free(string)` is indeed able to free this memory location, and the program ends successfully.

A.7 Header files and conditional compilation

In the course Imperative Programming, the programs were short enough to fit in one file. However, with increasing program size it becomes useful to split the program in several files. This helps to structure the program, and it enables to work on a program with several programmers simultaneously. The term for this is *modular programming*: the program is split in several components that can be developed/tested/optimized/extended independently.

We apply modular programming in Section 1.4, when we presented the evaluator of arithmetical expressions. The program is split in the three files `scanner.c`, `recognizeExp.c` and `evaluateExp.c`, corresponding with the three main subtasks of the program: scanning, recognition and evaluation.

When a program is split in several files, header files `xxx.h` are used to define types and declare functions that are defined in `xxx.c` and that can be used in other program files. So, for the evaluator program mentioned above, we have the header files `scanner.h`, `recognizeExp.h` and `evaluateExp.h`. When we inspect `scanner.h`, we see

```
#ifndef SCANNER_H
#define SCANNER_H

#define MAXINPUT 100  /* maximum length input */
#define MAXIDENT 10   /* maximum length identifier */

/* Some type definitions:
 * tokenType with three values, to indicate the three types
```

```
 * of tokens: number, identifier and symbol;
 * the union type token in which these types are unified;
 * the type list for lists with notes that contain tokens.
 */

typedef enum TokenType {
  Number,
  Identifier,
  Symbol
} TokenType;

typedef union Token {
  int number;
  char *identifier;
  char symbol;
} Token;

typedef struct ListNode *List;

typedef struct ListNode {
  TokenType tt;
  Token t;
  List next;
} ListNode;

/* Now the declaration of the functions that are defined
 * in scanner.c and are to be used outside it, e.g. in
 * recognizeExp.c en in evalExp.c
 */

char *readInput();
List tokenList(char *array);
int valueNumber(List *lp, double *wp);
void printList(List l);
void freeList(List l);
void scanExpressions();

#endif
```

So `scanner.h` contains definitions of the types `TokenType`, `Token`, `List` and `ListNode`. Moreover, it contains declarations of the functions `readInput`, `tokenList`, `valueNumber`, `printList`, `freeList` and `scanExpressions`. These functions are defined in `scanner.c`, but they can be used in any program file that contains `#include "scanner.h"`. This applies to `recognizeExp.c` and `evaluateExp.c`.

But what is the purpose of the statements `#ifndef SCANNER_H`, `#define SCANNER_H` and `#endif`? They form an *include guard*, preventing the definitions on the header file to be included more than once when combining several program files that contain **#include** "scanner.h". The C compiler applies the *one-definition rule*: an entity (type, variable, function) can be defined only once. With the global variable `SCANNER_H` we can ensure that the compiler skips the contents of `#include "scanner.h"` whenever it is not read for the first time. As a consequence, it is safe to include the same header file multiple times.

A.8 Makefiles

In general, splitting a large program in several files is a good strategy. However, when you end up with many files this may lead to chaos. It also becomes tiresome to type long `gcc ...` commands by hand. Here the program `make` comes in handy. It can be used for compiling (parts of) the program, and it does so efficiently. Calling `make` executes the content of the `Makefile`. As an example, here is a `Makefile` for the programs from Section 1.4.

```
CC = gcc
CFLAGS = -O2 -std=c99 -pedantic -Wall -o -lm

scan: mainScan.c scanner.c
    $(CC) $(CFLAGS) $^ -o $@

recog: mainRecog.c scanner.c recognizeExp.c
    $(CC) $(CFLAGS) $^ -o $@

eval: scanner.c recognizeExp.c evalExp.c mainEvalExp.c
    $(CC) $(CFLAGS) $^ -o $@
```

What do we see here? The first two lines define two variables, `CC` and `CFLAGS`. They are given the value `gcc` and `-std=-O2 -std=c99 -pedantic -Wall -o -lm`, respectively. These variables are used later on in the makefile, where the labels `scan`, `recog` and `eval` are defined. Then we see three instances of the following rule structure:

```
<target>: <prerequisites>
    <recipe>
```

Here `<target>` is the name of the file to be created by the rule. The `<prerequisites>` are the files that are used to create the target file, and `<recipe>` is a command to create the target file. Note: The line with `<recipe>` *must be indented using a tab*, not spaces!

If we now enter `make recog` in the command line, then the following command is executed:

```
gcc -O2 -std=c99 -pedantic -Wall -o -lm mainRecog.c scanner.c recognizeExp.c -o recog
```

That is, `$(CC)` and `$(CFLAGS)` are replaced by their values, `$^` is replaced by the prerequisites and `$@` is replaced by the name of the target, `recog`. The flag `-o` tells `gcc` to write its result not to `a.out`, but to the file named here (in this case `recog`).

When the compilation ends successfully, we can run `./recog` to execute the compiled program. When `make recog` is entered again, nothing will happen when the files `mainRecog.c scanner.c recognizeExp.c` in the prerequisites list have not been modified. But when one of them has been modified, the recipe will be executed again.

Another useful feature of `make` are so-called *phony* targets. Our example `Makefile` for Section 1.4 continues with:

```
.PHONY: clean debug-scan

clean:
    rm -f eval recog scan

debug-scan: scan
    cat example_part1_input.txt | valgrind ./scan
```

These commands do not produce a file, but they are simply shortcuts to easily run the same command again and again — for example when debugging a nasty memory leak.

There is much more to say about `make`. The interested reader is referred to the manual available at `https://www.gnu.org/software/make/manual/make.html`.

A.9 Exercises

Exercise A.1. Define a function in C that provides, given two numbers m and n, a two-dimensional array with dimension $m \times n$, filled with zeros.

Exercise A.2. Given the following program:

```
int main(int argc, char *argv[]) {
  int i;
  int **data = malloc(64 * sizeof(??));
  for (i=0; i< 64; i++) {
    data[i] = malloc(32 * sizeof(??));
  }
  free(data);
  return 0;
}
```

What is the correct replacement for the question marks (??) ?
Does this program leak memory? If so, indicate where and write a variant that does not leak memory.

Exercise A.3. In the next program fragment memory is allocated but not freed. Replace `/* ??? */` by a statement so that memory is freed correctly.

```
int main(int argc, char *argv[]) {
  int *p, *q;
  p = malloc(512 * sizeof(int));
  assert(p != NULL);
  q = p + 256;
  p = NULL;
  /* ??? */
  return 0;
}
```

Exercise A.4. The next program fragment initializes an array with squares. After this, it appears that the second half of the array is not needed and has to be freed. Give a statement that does this correctly.

```
int *p = malloc(512 * sizeof(int));
assert(p != NULL);
for (i=0; i<512; i++) {
  p[i] = i*i;
}
```

Exercise A.5. The program below is written by a freshman Computing Science to help the teaching staff. Unfortunately, it contains some errors with respect to memory management. Analyse the code and explain what the code is supposed to do. Repair all errors related to memory management.

```
void printHistogram(int *list,int num) {
  int i, j;
  int max;
```

```
  int *hist = malloc(10);
  for (i=0; i<num; i++) {  /* Make from list a histogram */
    hist[list[i]]++;
  }
  for (i=0; i<10; i++) {  /* Find the max in the histogram */
    max = (hist[i] > max) ? hist[i] : max;
  }
  for (j=max; j>=0; j--) {  /* Print the histogram */
    for (i=0;i<10; i++) {
      if (hist[i] > j) {
        printf("* ");
      } else {
        printf(". ");
      }
    }
    printf("\n");
  }
  printf("1 2 3 4 5 6 7 8 9 10\n");
}

int main(int argc, char *argv[]) {
  int numstudents, numgrades, grade;
  int i, j;
  int *histogram;
  int *avg = malloc(sizeof(int)), *avglist;
  char *namestudent;

  printf("Input the number of students:\n");
  scanf("%d", &numstudents);
  printf("Input the number of grades:\n");
  scanf("%d", &numgrades);

  avglist = malloc(numstudents);

  for (i=0; i<numstudents; i++) {
    /* Assume that 512 characters suffice for a name */
    namestudent = malloc(512);
    histogram = calloc(10, sizeof(int));
    printf("Input student name: %d\n", i+1);
    scanf("%s", namestudent);
    printf("Input the grades of %s:\n", namestudent);

    /* Read all grades in */
    for (j=0; j<numgrades; j++) {
      scanf("%d", &grade);
      histogram[grade-1]++;
    }

    /* Compute average */
    *avg = 0;
    for (j=0;j<10; j++) {
      *avg += (j+1)*histogram[j];
    }
    *avg = *avg / numgrades;
```

```
    avglist[i] = *avg;

    printf("The student's average grade is: %d\n", *avg);
  }

  printHistogram(avglist, numstudents);
  return 0;
}
```

Appendix B

Pseudocode

The pseudocode used in these lecture notes is defined as follows.

Declaration:	**algorithm** AlgName(par$_1$,. . . ,par$_n$) **input** description of the input parameters **output** description of the output (if any) **result** description of the result (e.g. when there is no output)
Expressions:	built from variables, numbers, mathematical and logical operators
Assignments:	of the form $x \leftarrow$ expression
If/Then/Else:	**if** condition **then** body [**else** body]
While/Do:	**while** condition **do** body
For/Do:	**for** $x = 1$ **to** n **do** body
Forall:	**forall** $x \in V$ **do** body
Return value:	**return** expression
Comment:	/∗ comment ∗/

The block structure is indicated by indentation.

This definition of pseudocode is not unique: there are many 'dialects'. In the lecture notes on Program Correctness, another form of pseudocode is used. We list the differences with the definition given here.

- A&DinC pseudocode has declarations of the name of the algorithm and the input parameters, and descriptions of the input parameters, the output and the result; these are not in PC pseudocode.
- In A&DinC pseudocode we often omit type declarations.
- A&DinC pseudocode uses '$\leftarrow$ ' for assignment.
- In A&DinC pseudocode ';' and '**end** ' are not used: blocks are indicated by indentation.
- A&DinC pseudocode has the program constructs '**for** ... **to** ... **do** and '**forall** ... **do** '.
- In A&DinC pseudocode the output of an algorithm is indicated by **return** .
- A&DinC pseudocode admits /∗ ∗/ to incorporate comment.

Appendix C

Time complexity: the $\mathcal{O}$ notation

Here we repeat the definition of the $\mathcal{O}$ ('big Oh') notation as given in Section 7.1.1. of the lecture notes *Imperative Programming*.

$f(n)$ is in $\mathcal{O}(g(n))$ iff:
there exist constants c and N such that $\forall\ n > N\ f(n) \leq c \cdot g(n)$.
Loosely formulated in words:
in the long run, $f(n)$ is smaller than a constant multiple of $g(n)$.

When using this definition, $f(n)$ is a function that indicates the number of computation steps of an algorithm, and n is a relevant parameter, e.g. the number of items in the stack, the height of the binary tree, or the length of a pattern. Furthermore $g(n)$ is a simple function of n, e.g. the constant 1, or n, or $n \cdot \log(n)$, or n^2, or 2^n.

Example: let $f(n)$ be the number of computational steps of the algorithm mergesort for a list with length n. The following statements all have the same meaning (and are all true):

- $f(n)$ is in $\mathcal{O}(n \log(n))$;
- the number of computation steps of mergesort for a list with length n is of the order $n \log(n)$;
- the *time complexity* of mergesort is of the order $n \log(n)$;
- mergesort is in $\mathcal{O}(n \log(n))$.

Appendix D

Criteria for programs

In this appendix we list some important quality criteria that your programs are to meet.

correctness A program should be syntactically correct: it compiles without error messages, and the code executes without errors (e.g. segmentation faults). Moreover, a program should be semantically correct: its input-output behaviour satisfies the specification, and it terminates when executed on acceptable input. Ideally, correctness can be proved rigorously. In most cases, it can only be tested, by executing the program on test sets.

efficiency A program should make efficient use of its resources (time, memory, processing power). Its time complexity should be as low as possible, and it should have no memory leaks.

simplicity The program should satisfy Albert Einstein's adagium:

> *make everything as simple as possible, but not simpler.*

clarity The structure of the program should be clear and logical, so that it can be understood by a human reader. The functions introduced should have a clear meaning. Global variables should be used only when they really contribute to the understandability of the program. The names of constants, variables, types and functions should reflect their meaning and use in a systematic way. The layout of the program text should contribute to its readability and understandability. Appropriate comments should be added, wherever useful.

In the next sections, we present conventions for naming and layout.

D.1 Naming conventions

- Names should be easily distinguishable (so avoid using e.g. both `e1` and `el`).
- A short name is acceptable if it is used very locally. When there is considerable distance between the declaration and the use of a name, a longer and more informative name is recommended.
- A descriptive name should accurately describe the entity it refers to.
- Use camelCase for composite names: the name should begin with a lower-case letter, each new part of the name starts with a capital letter (e.g. `newStack`).
- When a function returns a result and does not change the program state (i.e. the memory content), its name is a description of the result (e.g. `int firstItem(List li)`).
- When a function changes the program state, its name is (or: starts with) a verb that describes what the function does (e.g. `void doubleStackSize()`).
  This also applies to functions that change the program state *and* return a value (e.g. `List addItem(int n, List li)`).

D.2 Layout conventions

Every elementary statement is placed on a different line and ends with a semicolon ';'. For composite statements we use block layout, with indentation of fixed size (we use 2 spaces, no tabs). We demonstrate this with several programming constructs.

1. `if`-statement (the `else` part is optional):

```
if (condition) {
  thenStatement(s);
} else {
  elseStatement(s);
}
```

   in case of nesting:

```
if (condition1) {
  thenStatement(s);
} else if (condition2) {
  otherThenStatement(s);
} else {
  elseStatement(s);
}
```

2. `for`-statement:

```
for (init; condition; update) {
  forStatement(s);
}
```

3. `while`-statement:

```
while (condition) {
  whileStatement(s);
}
```

4. function definition:

```
outputType functionName(parameters) {
  functionDefinitionStatement(s);
}
```

Especially when collaborating with others on larger software projects it is a good idea to enforce certain layout conventions automatically. Many tools for this exist. For example, you can use `clang-format -i hello.c` to automatically format your program.

Appendix E

Programming reports

Being a good programmer is more than mastering the skill of writing good programs. It also comprises the skill of *communicating* about your programs: documenting them, explaining how they work, arguing why they are correct and efficient, and indicating how they were designed and how they may be extended. Some of these communication requirements are met by appropriate comments in the program text. More generally, however, a *programming report* is the standard document for communication about programs.

In this appendix, a description is given of the type of programming reports that is requested for some of the practical assignments. The general structure of your programming report should be as follows.

Problem description
Problem analysis
Program design
Evaluation of the program
Extension of the program (optional)
Process description
Appendix: program text
Appendix: test sets
Appendix: extended program text and test sets (optional)

Below you find a detailed description of the contents of each section.

Problem description Here you explain what the problem is. Describe the task your program should solve in your own words. Do not just copy the assignment description here. Suggestion: first give a short description in general terms, then provide a precise specification. You can also describe the problem by giving your own example of input-output behaviour.

Problem analysis Here you analyse the problem and how it could be solved. Some guidelines:

- Which part of the problem is easy, what is difficult?
- Skip irrelevant details and focus on the essence of the problem.
- Use mathematical terms (sets, functions, trees, etc.), not concepts specific to `C`.
- If possible, divide the problem into subproblems, in such a way that solution of the subproblems leads to a solution of the original problem.
- Look for similar problems for which we already have a solution.
- When you see more than one way to solve the problem (e.g. using different data structures), you may indicate this, and explain why you choose one specific solution.
- You may use pseudocode (see Appendix C) to describe one or more algorithms that are part of the solution, but do not use `C` source code here.

Program design Here you explain how you translate your ideas for solving the problem into a `C` program. The general advice here is: keep it general! Do not explain every statement in the program, but describe instead the strategic choices you made. Examples of such choices are the implementation of data structures, the functions and the programming constructs that are used in the program.

Evaluation of the program Here you report about the testing and the performance of the program. Indicate (here or in the Appendix) the test sets that you have used, and the output they generated. Also indicate whether your program was accepted by Themis. Do not forget to check for memory leaks.

Extension of the program (optional) In this optional section you may describe any extensions you made to the program. Indicate whether you followed one of the suggested extras, or that you came up with an extension of your own. Also discuss the problem analysis, program design and evaluation of your extension.

Process description Here you describe shortly the process that led to the final code and the report. What was easy, what was difficult? Did you make interesting mistakes? What have you learned from this assignment? Also indicate who did what while working on the assignment.

Conclusions Does your program solve the problem? How efficient is your program? Is it optimal? Note that "We found no better solution" is not an argument for optimality.

Appendix: program text Here you should include the program text. Do NOT use screenshots or similar methods. Instead, you should include your program with `\lstinputlisting`.

The program text must be exactly the same as your final submission on Themis! The program should contain comments to improve readability. See also the criteria given in Appendix D of the lecture notes.

Appendix: test sets If you wrote your own test cases to evaluate your program, please provide them here.

Appendix: extended program text (optional) In this optional section you describe any extensions you made to the program. For example if you also implemented the extra part, or ideas you came up with on your own.

Last but not least, a general remark:

Keep your writing style professional: write correct sentences, check your spelling!