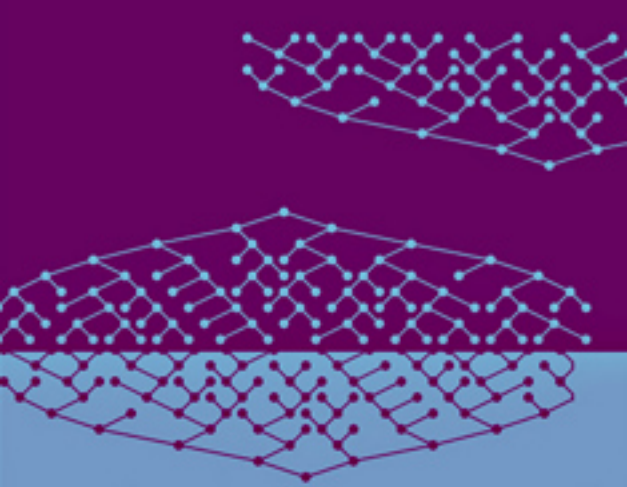


ADVANCED DATA STRUCTURES

PETER BRASS

CAMBRIDGE

CAMBRIDGE www.cambridge.org/9780521880374

This page intentionally left blank

Advanced Data Structures

Advanced Data Structures presents a comprehensive look at the ideas, analysis, and implementation details of data structures as a specialized topic in applied algorithms. This book examines efficient ways to realize query and update operations on sets of numbers, intervals, or strings by various data structures, including search trees, structures for sets of intervals or piecewise constant functions, orthogonal range search structures, heaps, union-find structures, dynamization and persistence of structures, structures for strings, and hash tables. Instead of relegating data structures to trivial material used to illustrate object-oriented programming methodology, this is the first volume to show data structures as a crucial algorithmic topic. Numerous code examples in C and more than 500 references make *Advanced Data Structures* an indispensable text.

PETER BRASS received a Ph.D. in mathematics at the Technical University of Braunschweig, Germany. He is an associate professor at the City College of New York in the Department of Computer Science and a former Heisenberg Research Fellow of the Free University of Berlin.

Advanced Data Structures

PETER BRASS
City College of New York

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK
Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521880374

© Peter Brass 2008

This publication is in copyright. Subject to statutory exception and to the provision of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published in print format 2008

ISBN-13 978-0-511-43388-7 eBook (Adobe Reader)

ISBN-13 978-0-521-88037-4 hardback

Cambridge University Press has no responsibility for the persistence or accuracy of urls for external or third-party internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

Dedicated to my parents,
Gisela and Helmut Brass

Contents

Preface *page* xi

1 Elementary Structures 1
1.1 Stack 1
1.2 Queue 8
1.3 Double-Ended Queue 16
1.4 Dynamical Allocation of Nodes 16
1.5 Shadow Copies of Array-Based Structures 18

2 Search Trees 23
2.1 Two Models of Search Trees 23
2.2 General Properties and Transformations 26
2.3 Height of a Search Tree 29
2.4 Basic Find, Insert, and Delete 31
2.5 Returning from Leaf to Root 35
2.6 Dealing with Nonunique Keys 37
2.7 Queries for the Keys in an Interval 38
2.8 Building Optimal Search Trees 40
2.9 Converting Trees into Lists 47
2.10 Removing a Tree 48

3 Balanced Search Trees 50
3.1 Height-Balanced Trees 50
3.2 Weight-Balanced Trees 61
3.3 (a, b)- and B-Trees 72
3.4 Red-Black Trees and Trees of Almost Optimal Height 89
3.5 Top-Down Rebalancing for Red-Black Trees 101
3.6 Trees with Constant Update Time at a Known Location 111
3.7 Finger Trees and Level Linking 114

3.8 Trees with Partial Rebuilding: Amortized Analysis 119
3.9 Splay Trees: Adaptive Data Structures 122
3.10 Skip Lists: Randomized Data Structures 135
3.11 Joining and Splitting Balanced Search Trees 143

4 Tree Structures for Sets of Intervals 148
4.1 Interval Trees 148
4.2 Segment Trees 154
4.3 Trees for the Union of Intervals 162
4.4 Trees for Sums of Weighted Intervals 169
4.5 Trees for Interval-Restricted Maximum Sum Queries 174
4.6 Orthogonal Range Trees 182
4.7 Higher-Dimensional Segment Trees 196
4.8 Other Systems of Building Blocks 199
4.9 Range-Counting and the Semigroup Model 202
4.10 kd-Trees and Related Structures 204

5 Heaps 209
5.1 Balanced Search Trees as Heaps 210
5.2 Array-Based Heaps 214
5.3 Heap-Ordered Trees and Half-Ordered Trees 221
5.4 Leftist Heaps 227
5.5 Skew Heaps 235
5.6 Binomial Heaps 239
5.7 Changing Keys in Heaps 248
5.8 Fibonacci Heaps 250
5.9 Heaps of Optimal Complexity 262
5.10 Double-Ended Heap Structures and Multidimensional Heaps 267
5.11 Heap-Related Structures with Constant-Time Updates 271

6 Union-Find and Related Structures 278
6.1 Union-Find: Merging Classes of a Partition 279
6.2 Union-Find with Copies and Dynamic Segment Trees 293
6.3 List Splitting 303
6.4 Problems on Root-Directed Trees 306
6.5 Maintaining a Linear Order 317

7 Data Structure Transformations 321
7.1 Making Structures Dynamic 321
7.2 Making Structures Persistent 330

8 Data Structures for Strings 335
8.1 Tries and Compressed Tries 336
8.2 Dictionaries Allowing Errors in Queries 356
8.3 Suffix Trees 360
8.4 Suffix Arrays 367

9 Hash Tables 374
9.1 Basic Hash Tables and Collision Resolution 374
9.2 Universal Families of Hash Functions 380
9.3 Perfect Hash Functions 391
9.4 Hash Trees 397
9.5 Extendible Hashing 398
9.6 Membership Testers and Bloom Filters 402

10 Appendix 406
10.1 The Pointer Machine and Alternative Computation Models 406
10.2 External Memory Models and Cache-Oblivious Algorithms 408
10.3 Naming of Data Structures 409
10.4 Solving Linear Recurrences 410
10.5 Very Slowly Growing Functions 412

11 References 415

Author Index 441

Subject Index 455

Preface

This book is a graduate-level textbook on data structures. A data structure is a method[1] to realize a set of operations on some data. The classical example is to keep track of a set of items, the items identified by key values, so that we can insert and delete (key, item) pairs into the set and find the item with a given key value. A structure supporting these operations is called a dictionary. Dictionaries can be realized in many different ways, with different complexity bounds and various additional operations supported, and indeed many kinds of dictionaries have been proposed and analyzed in literature, and some will be studied in this book.

In general, a data structure is a kind of higher-level instruction in a virtual machine: when an algorithm needs to execute some operations many times, it is reasonable to identify what exactly the needed operations are and how they can be realized in the most efficient way. This is the basic question of data structures: given a set of operations whose intended behavior is known, how should we realize that behavior?

There is no lack of books carrying the words "data structures" in the title, but they merely scratch the surface of the topic, providing only the trivial structures stack and queue, and then some balanced search tree with a large amount of handwaving. Data structures started receiving serious interest in the 1970s, and, in the first half of the 1980s, almost every issue of the *Communications of the ACM* contained a data structure paper. They were considered a central topic, received their own classification in the *Computing Subject Classification*,[2]

[1] This is not a book on object-oriented programming. I use the words "method" and "object" in their normal sense.

[2] Classification code: E.1 data structures. Unfortunately, the *Computing Subject Classification* is too rough to be useful.

and became a standard part of computer science curricula.[3] Wirth titled a book *Data Structures + Algorithms = Programs*, and *Algorithms and Data Structures* became a generic textbook title. But the only monograph on an algorithmic aspect of data structures is the book by Overmars (1983) (which is still in print, a kind of record for an LNCS series book). Data structures received attention in a number of application areas, foremost as index structures in databases. In this context, structures for geometric data have been studied in the monographs of Samet (1990, 2006); the same structures were studied in the computer graphics context in Langetepe and Zachmann (2006). Recently, motivated by bioinformatics applications, string data structures have been much studied. There is a steady stream of publications on data structure theory as part of computational geometry or combinatorial optimization. But in the numerous textbooks, data structures are only viewed as an example application of object-oriented programming, excluding the algorithmic questions of how to really do something nontrivial, with bounds on the worst-case complexity. It is the aim of this book to bring the focus back to data structures as a fundamental subtopic of algorithms. The recently published *Handbook of Data Structures* (Mehta and Sahni 2005) is a step in the same direction.

This book contains real code for many of the data structures we discuss and enough information to implement most of the data structures where we do not provide an implementation. Many textbooks avoid the details, which is one reason that the structures are not used in the places where they should be used. The selection of structures treated in this book is therefore restricted almost everywhere to such structures that work in the pointer-machine model, with the exception of hash tables, which are included for their practical importance. The code is intended as illustration, not as ready-to-use plug-in code; there is certainly no guarantee of correctness. Most of it is available with a minimal testing environment on my homepage.

This book started out as notes for a course I gave in the 2000 winter semester at the Free University Berlin; I thank Christian Knauer, who was my assistant for that course: we both learned a lot. I offered this course again in the fall semesters of 2004–7 as a graduate course at the City College of New York and used it as a base for a summer school on data structures at the Korean Advanced Institute of Science and Technology in July 2006. I finished this book in November 2007.

[3] ABET still lists them as one of five core topics: algorithms, data structures, software design, programming languages, and computer architecture.

I thank Emily Voytek and Günter Rote for finding errors in my code examples, Otfried Cheong for organizing the summer school at KAIST, and the summer school's participants for finding further errors. I thank Christian Knauer and Helmut Brass for literature from excellent mathematical libraries at the Free University Berlin and Technical University Braunschweig, and János Pach for access to the online journals subscribed by the Courant Institute. A project like this book would not have been possible without access to good libraries, and I tried to cite only those papers that I have seen.

This book project has not been supported by any grant-giving agency.

Basic Concepts

A data structure models some abstract object. It implements a number of operations on this object, which usually can be classified into

– creation and deletion operations,
– update operations, and
– query operations.

In the case of the dictionary, we want to create or delete the set itself, update the set by inserting or deleting elements, and query for the existence of an element in the set.

Once it has been created, the object is changed by the update operations. The query operations do not change the abstract object, although they might change the representation of the object in the data structure: this is called an adaptive data structure – it adapts to the query to answer future similar queries faster.

Data structures that allow updates and queries are called dynamic data structures. There are also simpler structures that are created just once for some given object and allow queries but no updates; these are called static data structures. Dynamic data structures are preferable because they are more general, but we also need to discuss static structures because they are useful as building blocks for dynamic structures, and, for some of the more complex objects we encounter, no dynamic structure is known.

We want to find data structures that realize a given abstract object and are fast. The size of structures is another quality measure, but it is usually of less importance. To express speed, we need a measure of comparison; this is the size of the underlying object, not our representation of that object. Notice that a long sequence of update operations can still result in a small object. Our

usual complexity measure is the worst-case complexity; so an operation in a specific data structure has a complexity $O(f(n))$ if, for any state of the data structure reached by a sequence of update operations that produced an object of size n, this operation takes at most time $Cf(n)$ for some C. An alternative but weaker measure is the amortized complexity; an update operation has amortized complexity $O(f(n))$ if there is some function $g(n)$ such that any sequence of m of these operations, during which the size of the underlying object is never larger than n, takes at most time $g(n) + mCf(n)$, so in the average over a long sequence of operations the complexity is bounded by $Cf(n)$.

Some structures are randomized, so the data structure makes some random choices, and the same object and sequence of operations do not always lead to the same steps of the data structure. In that case we analyze the expected complexity of an operation. This expectation is over the random choices of the data structure; the complexity is still the worst case of that expectation over all objects of that size and possible operations.

In some situations, we cannot expect a nontrivial complexity bound of type $O(f(n))$ because the operation might give a large answer. The size of the answer is the output complexity of the operation, and, for operations that sometimes have a large output complexity, we are interested in output-sensitive methods, which are fast when the output is small. An operation has output-sensitive complexity $O(f(n) + k)$ if, on an object of size n that requires an output of size k, the operation takes at most time $C(f(n) + k)$.

For dynamic data structures, the time to create the structure for an empty object is usually constant, so we are mainly interested in the update and query times. The time to delete a structure of size n is almost always $O(n)$. For static data structures we already create an object of size n, so there we are interested in the creation time, known as preprocessing time, and the query time.

In this book, $\log_a n$ denotes the logarithm to base a; if no base is specified, we use base 2.

We use the Bourbaki-style notation for closed, half-open, and open intervals, where $[a, b]$ is the closed interval from a to b, $]a, b[$ is the open interval, and the half-open intervals are $]a, b]$, missing the first point, and $[a, b[$, missing the last point.

Similar to the $O(\cdot)$-notation for upper bounds mentioned earlier, we also use the $\Omega(\cdot)$ for lower bounds and $\Theta(\cdot)$ for simultaneous upper and lower bounds. A nonnegative function f is $O(g(n))$, or $\Omega(g(n))$, if for some positive C and all sufficiently large n holds $f(n) \leq Cg(n)$, or $f(n) \geq Cg(n)$, respectively. And f is $\Theta(g(n))$ if it is simultaneously $O(g(n))$ and $\Omega(g(n))$. Here "sufficiently large" means that $g(n)$ needs to be defined and positive.

Code Examples

The code examples in this book are given in standard C. For the readers used to some other imperative programming language, most constructs are self-explanatory.

In the code examples, `=` denotes the assignment and `==` the equality test. Outside the code examples, we will continue to use = in the normal way.

The Boolean operators for "not," "and," "or" are `!`, `&&`, `||`, respectively, and `%` denotes the modulo operator.

Pointers are dereferenced with `*`, so if `pt` is a pointer to a memory location (usually a variable), then `*pt` is that memory location. Pointers have a type to determine how the content of that memory location should be interpreted. To declare a pointer, one declares the type of the memory location it points to, so "`int *pt;`" declares `pt` to be a pointer to an `int`. Pointers are themselves variables; they can be assigned, and it is also possible to add integers to a pointer (pointer arithmetic). If `pt` points to a memory object of a certain type, then `pt+1` points to the next memory location for an object of that type; this is equivalent to treating the memory as a big array of objects of that type. `NULL` is a pointer that does not point to any valid memory object, so it can be used as a special mark in comparisons.

Structures are user-defined data types that have several components. The components themselves have a type and a name, and they can be of any type, including other structures. The structure cannot have itself as a type of a component, because that would generate an unbounded recursion. But it can have a pointer to an object of its own type as component; indeed, such structures are the main tool of data structure theory. A variable whose type is a structure can be assigned and used like any other variable. If `z` is a variable of type `C`, and we define this type by

```
typedef struct { float x; float y; } C,
```

then the components of `z` are `z.x` and `z.y`, which are two variables of type `float`. If `zpt` is declared as pointer to an object of type `C` (by `C *zpt;`), then the components of the object that `zpt` points to are `(*zpt).x` and `(*zpt).y`. Because this is a frequently used combination, dereferencing a pointer and selecting a component, there is an alternative notation `zpt->x` and `zpt->y`. This is equivalent, but preferable, because it avoids the operator priority problem: dereferencing has lower priority than component selection, so `(*zpt).x` is not the same as `*zpt.x`.

We avoid writing the functions recursively, although in some cases this might simplify the code. But the overhead of a recursive function call is significant

and thus conflicts with the general aim of highest efficiency in data structures. We do not practice any similar restrictions for nonrecursive functions; a good compiler will expand them as inline functions, avoiding the function call, or they could be written as macro functions.

In the text we will also frequently use the name of a pointer for the object to which it points.

1
Elementary Structures

Elementary data structures usually treated in the "Programming 2" class are the *stack* and the *queue*. They have a common generalization, the *double-ended queue*, which is also occasionally mentioned, although it has far fewer applications. Stack and queue are very fundamental structures, so they will be discussed in detail and used to illustrate several points in data structure implementation.

1.1 Stack

The stack is the simplest of all structures, with an obvious interpretation: putting objects on the stack and taking them off again, with access possible only to the top item. For this reason they are sometimes also described as LIFO storage: last in, first out. Stacks occur in programming wherever we have nested blocks, local variables, recursive definitions, or backtracking. Typical programming exercises that involve a stack are the evaluation of arithmetic expressions with parentheses and operator priorities, or search in a labyrinth with backtracking.

The stack should support at least the following operations:

{ `push( obj )`: Put `obj` on the stack, making it the top item.
{ `pop()`: Return the top object from the stack and remove it from the stack.
{ `stack_empty()`: Test whether the stack is empty.

Also, the realization of the stack has, of course, to give the right values, so we need to specify the correct behavior of a stack. One method would be an algebraic specification of what correct sequences of operations and return values are. This has been done for simple structures like the stack, but even then the specification is not very helpful in understanding the structure. Instead, we can describe a canonical implementation on an idealized machine, which gives the correct answer for all correct sequences of operations (no pop on an

empty stack, no memory problems caused by bounded arrays). Assuming that the elements we want to store on the stack are of type `item_t`, this could look as follows:

```
int i=0;
item_t stack[∞];

int stack_empty(void)
{   return( i == 0 );
}

void push( item_t x)
{   stack[i++] = x ;
}

item_t pop(void)
{   return( stack[ --i] );
}
```

This describes the correct working of the stack, but we have the problem of assuming both an infinite array and that any sequence of operations will be correct. A more realistic version might be the following:

```
int i=0;
item_t stack[MAXSIZE];

int stack_empty(void)
{   return( i == 0 );
}

int push( item_t x)
{   if ( i < MAXSIZE )
    {   stack[i++] = x ;  return( 0 );
    }
    else
       return( -1 );
}

item_t pop(void)
{   return( stack[ --i] );
}
```

This now limits the correct behavior of the stack by limiting the maximum number of items on the stack at one time, so it is not really the correct stack we want, but at least it does specify an error message in the return value if the stack overflow is reached by one `push` too many. This is a fundamental defect of array-based realizations of data structures: they are of fixed size, the size needs to be decided in advance, and the structure needs the full size no matter how many items are really in the structure. There is a systematic way to overcome these problems for array-based structures, which we will see in Section 1.5, but usually a solution with dynamically allocated memory is preferable.

We specified an error value only for the stack overflow condition, but not for the stack underflow, because the stack overflow is an error generated by the structure, which would not be present in an ideal implementation, whereas a stack underflow is an error in the use of the structure and so a result in the program that uses the stack as a black box. Also, this allows us to keep the return value of `pop` as the top object from the stack; if we wanted to catch stack underflow errors in the stack implementation, we would need to return the object and the error status. A final consideration in our first stack version is that we might need multiple stacks in the same program, so we want to create the stacks dynamically. For this we need additional operations to create and remove a stack, and each stack operation needs to specify which stack it operates on. One possible implementation could be the following:

```
typedef struct {item_t *base; item_t *top;
                int size;} stack_t;

stack_t *create_stack(int size)
{   stack_t *st;
    st = (stack_t *) malloc( sizeof(stack_t) );
    st->base = (item_t *) malloc( size *
                sizeof(item_t) );
    st->size = size;
    st->top = st->base;
    return( st );
}

int stack_empty(stack_t *st)
{   return( st->base == st->top );
}
```

```
int push( item_t x, stack_t *st)
{   if ( st->top < st->base + st->size )
    {  *(st->top) = x; st->top += 1; return( 0 );
    }
    else
       return( -1 );
}

item_t pop(stack_t *st)
{   st->top -= 1;
    return( *(st->top) );
}

item_t top_element(stack_t *st)
{   return( *(st->top -1) );
}

void remove_stack(stack_t *st)
{   free( st->base );
    free( st );
}
```

Again, we include some security checks and leave out others. Our policy in general is to include those security checks that test for errors introduced by the limitations of this implementation as opposed to an ideal stack, but to assume both that the use of the stack is correct and that the underlying operating system never runs out of memory. We included another operation that is frequently useful, which just returns the value of the top element without taking it from the stack.

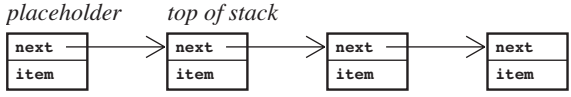
Frequently, the preferable implementation of the stack is a dynamically allocated structure using a linked list, where we insert and delete in front of the list. This has the advantage that the structure is not of fixed size; therefore, we need not be prepared for stack overflow errors if we can assume that the memory of the computer is unbounded, and so we can always get a new node. It is as simple as the array-based structure if we already have the `get_node` and `return_node` functions, whose correct implementation we discuss in Section 1.4.

```
typedef struct st_t { item_t       item;
                      struct st_t *next; } stack_t;
```

```
stack_t *create_stack(void)
{   stack_t *st;
    st = get_node();
    st->next = NULL;
    return( st );
}

int stack_empty(stack_t *st)
{   return( st->next == NULL );
}

void push( item_t x, stack_t *st)
{   stack_t *tmp;
    tmp = get_node();
    tmp->item = x;
    tmp->next = st->next;
    st->next = tmp;
}

item_t pop(stack_t *st)
{   stack_t *tmp; item_t tmp_item;
    tmp = st->next;
    st->next = tmp->next;
    tmp_item = tmp->item;
    return_node( tmp );
    return( tmp_item );
}

item_t top_element(stack_t *st)
{   return( st->next->item );
}

void remove_stack(stack_t *st)
{   stack_t *tmp;
    do
    {  tmp = st->next;
       return_node(st);
       st = tmp;
    }
    while ( tmp != NULL );
}
```

Notice that we have a placeholder node in front of the linked list; even an empty stack is represented by a list with one node, and the top of the stack is

only the second node of the list. This is necessary as the stack identifier returned by `create_stack` and used in all stack operations should not be changed by the stack operations. So we cannot just use a pointer to the start of the linked list as a stack identifier. Because the components of a node will be invalid after it is returned, we need temporary copies of the necessary values in `pop` and `remove_stack`. The operation `remove_stack` should return all the remaining nodes; there is no reason to assume that only empty stacks will be removed, and we will suffer a memory leak if we fail to return the remaining nodes.



STACK REALIZED AS LIST, WITH THREE ITEMS

The implementation as a dynamically allocated structure always has the advantage of greater elegance; it avoids stack overflow conditions and needs just the memory proportional to the actually used items, not a big array of a size estimated by the programmer as upper bound to the maximum use expected to occur. One disadvantage is a possible decrease in speed: dereferencing a pointer does not take longer than incrementing an index, but the memory location accessed by the pointer might be anywhere in memory, whereas the next component of the array will be near the previous component. Thus, array-based structures usually work very well with the cache, whereas dynamically allocated structures might generate many cache misses. So if we are quite certain about the maximum possible size of the stack, for example, because its size is only logarithmic in the size of the input, we will prefer an array-based version.

If one wants to combine these advantages, one could use a linked list of blocks, each block containing an array, but when the array becomes full, we just link it to a new node with a new array. Such an implementation could look as follows:

```
typedef struct st_t { item_t *base;
                      item_t  *top;
                      int     size;
                  struct st_t *previous;} stack_t;

stack_t *create_stack(int size)
{   stack_t *st;
    st = (stack_t *) malloc( sizeof(stack_t) );
    st->base = (item_t *) malloc( size *
               sizeof(item_t) );
    st->size = size;
    st->top = st->base;
```

```
    st->previous = NULL;
    return( st );
}

int stack_empty(stack_t *st)
{   return( st->base == st->top &&
            st->previous == NULL);
}

void push( item_t x, stack_t *st)
{   if ( st->top < st->base + st->size )
    {   *(st->top) = x; st->top += 1;
    }
    else
    {   stack_t *new;
        new = (stack_t *) malloc( sizeof(stack_t) );
        new->base = st->base;
        new->top = st->top;
        new->size = st->size;
        new->previous = st->previous;
        st->previous = new;
        st->base = (item_t *) malloc( st->size *
                   sizeof(item_t) );
        st->top = st->base+1;
        *(st->base) = x;
    }
}

item_t pop(stack_t *st)
{   if( st->top == st->base )
    {  stack_t *old;
       old = st->previous;
       st->previous = old->previous;
       free( st->base );
       st->base = old->base;
       st->top = old->top;
       st->size = old->size;
       free( old );
    }
    st->top -= 1;
    return( *(st->top) );
}

item_t top_element(stack_t *st)
{  if( st->top == st->base )
      return( *(st->previous->top -1) );
```

```
    else
       return( *(st->top -1) );
}

void remove_stack(stack_t *st)
{   stack_t *tmp;
    do
    {  tmp = st->previous;
       free( st->base );
       free( st );
       st = tmp;
    }
    while( st != NULL );
}
```

In our classification, push and pop are update operations and stack_empty and top_element are query operations. In the array-based implementation, it is obvious that we can do all the operations in constant time as they involve only a constant number of elementary operations. For the linked-list implementation, the operations involve the external get_node and return_node functions, which occur in both push and pop once, so the implementation works only in constant time if we can assume these functions to be constant-time operations. We will discuss the implementation of this dynamic node allocation in Section 1.4, but we can assume here (and in all later structures) that this works in constant time. For the block list we allocate large parts of memory for which we used here the standard memory management operations malloc and free instead of building an intermediate layer, as described in Section 1.4. It is traditional to assume that memory allocation and deallocation are constant-time operations, but especially with the free there are nontrivial problems with a constant-time implementation, so one should avoid using it frequently. This could happen in the block list variant if there are many push/pop pairs that just go over a block boundary. So the small advantage of the block list is probably not worth the additional problems.

The create_stack operation involves only one such memory allocation, and so that should be constant time in each implementation; but the remove_stack operation is clearly not constant time, because it has to destroy a potentially large structure. If the stack still contains n elements, the remove_stack operation will take time $O(n)$.

1.2 Queue

The queue is a structure almost as simple as the stack; it also stores items, but it differs from the stack in that it returns those items first that have been

entered first, so it is FIFO storage (first in, first out). Queues are useful if there are tasks that have to be processed cyclically. Also, they are a central structure in breadth-first search; breadth-first search (BFS) and depth-first search (DFS) really differ only in that BFS uses a queue and DFS uses a stack to store the node that will be explored next.

The queue should support at least the following operations:

- `enqueue( obj )`: Insert `obj` at the end of the queue, making it the last item.
- `dequeue()`: Return the first object from the queue and remove it from the queue.
- `queue_empty()`: Test whether the queue is empty.

The difference between queue and stack that makes the queue slightly more difficult is that the changes occur at both ends: at one end, there are inserts; at the other, deletes. If we choose an array-based implementation for the queue, then the part of the array that is in use moves through the array. If we had an infinite array, this would present no problem. We could write it as follows:

```
int lower=0; int upper=0;
item_t queue[∞];

int queue_empty(void)
{   return( lower == upper );
}

void enqueue( item_t x)
{   queue[upper++] = x ;
}

item_t dequeue(void)
{   return( queue[ lower++] );
}
```

A real implementation with a finite array has to wrap this around, using index calculation modulo the length of the array. It could look as follows:

```
typedef struct {item_t *base;
                int    front;
                int     rear;
                int     size;} queue_t;
```

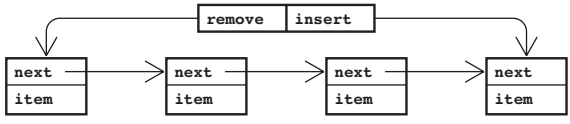
```
queue_t *create_queue(int size)
{   queue_t *qu;
    qu = (queue_t *) malloc( sizeof(queue_t) );
    qu->base = (item_t *) malloc( size *
               sizeof(item_t) );
    qu->size = size;
    qu->front = qu->rear = 0;
    return( qu );
}

int queue_empty(queue_t *qu)
{   return( qu->front == qu->rear );
}

int enqueue( item_t x, queue_t *qu)
{   if ( qu->front != ((qu->rear +2)% qu->size) )
    {   qu->base[qu->rear] = x;
        qu->rear = ((qu->rear+1)%qu->size);
        return( 0 );
    }
    else
       return( -1 );
}

item_t dequeue(queue_t *qu)
{   int tmp;
    tmp = qu->front;
    qu->front = ((qu->front +1)%qu->size);
    return( qu->base[tmp] );
}

item_t front_element(queue_t *qu)
{   return( qu->base[qu->front] );
}

void remove_queue(queue_t *qu)
{   free( qu->base );
    free( qu );
}
```

Again this has the fundamental disadvantage of any array-based structure – that it is of fixed size. So it possibly generates overflow errors and does not implement the structure correctly as it limits it this way. In addition, it always reserves this expected maximum size for the array, even if it never needs it. The preferred alternative is a dynamically allocated structure, with a linked list. The obvious solution is the following:

```
typedef struct qu_n_t {item_t    item;
                  struct qu_n_t *next; } qu_node_t;
typedef struct {qu_node_t *remove;
                qu_node_t *insert; } queue_t;

queue_t *create_queue()
{   queue_t *qu;
    qu = (queue_t *) malloc( sizeof(queue_t) );
    qu->remove = qu->insert = NULL;
    return( qu );
}

int queue_empty(queue_t *qu)
{   return( qu->insert ==NULL );
}

void enqueue( item_t x, queue_t *qu)
{   qu_node_t *tmp;
    tmp = get_node();
    tmp->item = x;
    tmp->next = NULL; /* end marker */
    if ( qu->insert != NULL ) /* queue nonempty */
    {   qu->insert->next = tmp;
        qu->insert = tmp;
    }
    else /* insert in empty queue */
    {   qu->remove = qu->insert = tmp;
    }
}

item_t dequeue(queue_t *qu)
{   qu_node_t *tmp; item_t tmp_item;
    tmp = qu->remove; tmp_item = tmp->item;
    qu->remove = tmp->next;
    if( qu->remove == NULL ) /* reached end */
        qu->insert = NULL; /* make queue empty */
    return_node(tmp);
```

```
    return( tmp_item );
}

item_t front_element(queue_t *qu)
{   return( qu->remove->item );
}

void remove_queue(queue_t *qu)
{   qu_node_t *tmp;
    while( qu->remove != NULL)
    { tmp = qu->remove;
      qu->remove = tmp->next;
      return_node(tmp);
    }
    free( qu );
}
```

Again we assume, as in all dynamically allocated structures, that the operations `get_node` and `return_node` are available, which always work correctly and in constant time. Because we want to remove items from the front of the queue, the pointers in the linked list are oriented from the front to the end, where we insert items. There are two aesthetical disadvantages of this obvious implementation: we need a special entry point structure, which is different from the list nodes, and we always need to treat the operations involving an empty queue differently. For insertions into an empty queue and removal of the last element of the queue, we need to change both insertion and removal pointers; for all other operations we change only one of them.



QUEUE REALIZED AS LIST, WITH FOUR ITEMS

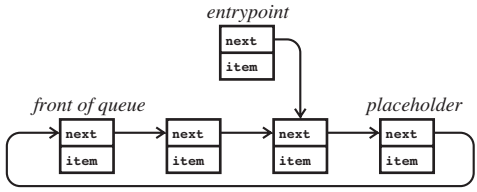
The first disadvantage can be avoided by joining the list together to make it a cyclic list, with the last pointer from the end of the queue pointing again to the beginning. We can then do without a removal pointer, because the insertion point's next component points to the removal point. By this, the entry point to the queue needs only one pointer, so it is of the same type as the queue nodes.

The second disadvantage can be overcome by inserting a placeholder node in that cyclic list, between the insertion end and the removal end of the cyclic list. The entry point still points to the insertion end or, in the case of an empty

list, to that placeholder node. Then, at least for the insert, the empty list is no longer a special case. So a cyclic list version is the following:

```
typedef struct qu_t { item_t  item;
                 struct qu_t *next; } queue_t;

queue_t *create_queue()
{   queue_t *entrypoint, *placeholder;
    entrypoint = (queue_t *) malloc( sizeof(queue_t) );
    placeholder = (queue_t *) malloc( sizeof(queue_t) );
    entrypoint->next = placeholder;
    placeholder->next = placeholder;
    return( entrypoint );
}

int queue_empty(queue_t *qu)
{   return( qu->next == qu->next->next );
}

void enqueue( item_t x, queue_t *qu)
{   queue_t *tmp, *new;
    new = get_node(); new->item = x;
    tmp = qu->next; qu->next = new;
    new->next = tmp->next; tmp->next = new;
}

item_t dequeue(queue_t *qu)
{   queue_t     *tmp;
    item_t  tmp_item;
    tmp = qu->next->next->next;
    qu->next->next->next = tmp->next;
    if( tmp == qu->next )
       qu->next = tmp->next;
    tmp_item = tmp->item;
    return_node( tmp );
    return( tmp_item );
}

item_t front_element(queue_t *qu)
{   return( qu->next->next->next->item );
}

void remove_queue(queue_t *qu)
{   queue_t *tmp;
    tmp = qu->next->next;
    while( tmp != qu->next )
    { qu->next->next = tmp->next;
      return_node( tmp );
```

```
        tmp = qu->next->next;
    }
    return_node( qu->next );
    return_node( qu );
}
```

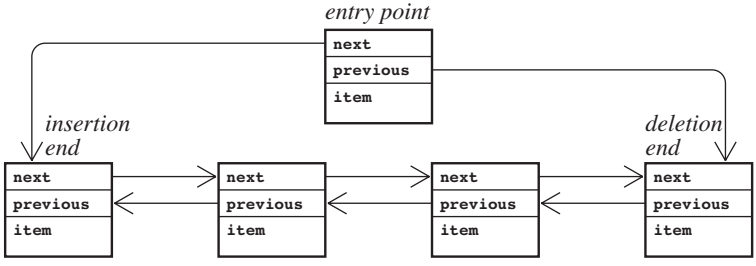


QUEUE REALIZED AS CYCLIC LIST, WITH THREE ITEMS

Or one could implement the queue as a doubly linked list, which requires no case distinctions at all but needs two pointers per node. Minimizing the number of pointers is an aesthetic criterion more justified by the amount of work that has to be done in each step to keep the structure consistent than by the amount of memory necessary for the structure. Here is a doubly linked list implementation:

```
typedef struct qu_t { item_t       item;
                      struct qu_t     *next;
                      struct qu_t *previous; } queue_t;

queue_t *create_queue()
{   queue_t *entrypoint;
    entrypoint = (queue_t *) malloc( sizeof(queue_t) );
    entrypoint->next = entrypoint;
    entrypoint->previous = entrypoint;
    return( entrypoint );
}

int queue_empty(queue_t *qu)
{   return( qu->next == qu );
}

void enqueue( item_t x, queue_t *qu)
{   queue_t *new;
    new = get_node(); new->item = x;
    new->next = qu->next; qu->next = new;
    new->next->previous = new; new->previous = qu;
}

item_t dequeue(queue_t *qu)
{   queue_t *tmp; item_t tmp_item;
    tmp = qu->previous; tmp_item = tmp->item;
```

```
    tmp->previous->next = qu;
    qu->previous = tmp->previous;
    return_node( tmp );
    return( tmp_item );
}

item_t front_element(queue_t *qu)
{   return( qu->previous->item );
}

void remove_queue(queue_t *qu)
{   queue_t *tmp;
    qu->previous->next = NULL;
    do
    { tmp = qu->next;
      return_node( qu );
      qu = tmp;
    }
    while ( qu != NULL );

}
```



QUEUE REALIZED AS DOUBLY LINKED LIST, WITH FOUR ITEMS

Which of the list-based implementations one prefers is really a matter of taste; they are all slightly more complicated than the stack, although the two structures look similar.

Like the stack, the queue is a dynamic data structure that has the update operations `enqueue` and `dequeue` and the query operations `queue_empty` and `front_element`, all of which are constant-time operations, and the operations `create_queue` and `delete_queue`, which are subject to the same restrictions as the similar operations for the stack: creating an array-based queue requires getting a big block of memory from the underlying system memory management, whereas creating a list-based queue should require only some `get_node` operations; and deleting an array-based queue just involves

returning that memory block to the system, whereas deleting a list-based queue requires returning every individual node still contained in it, so it will take $O(n)$ time to delete a list-based queue that still contains n items.

1.3 Double-Ended Queue

The double-ended queue is the obvious common generalization of stack and queue: a queue in which one can insert and delete at either end. Its implementation can be done as an array, or as a doubly linked list, just like a queue; because it does not present any new problems, no code will be given here. The double-ended queue does not have many applications, but at least a "one-and-a-half ended queue" sometimes is useful, as in the minqueue discussed in Section 5.11.

1.4 Dynamical Allocation of Nodes

In the previous sections we used the operations `get_node` and `return_node` to dynamically create and delete nodes, that is, constant-sized memory objects, as opposed to the generic operations `malloc` and `free` provided by the standard operating-system interface, which we used only for memory objects of arbitrary, usually large, size. The reason for this distinction is that although the operating-system memory allocation is ultimately the only way to get memory, it is a complicated process, and it is not even immediately obvious that it is a constant-time operation. In any efficient implementation of a dynamically allocated structure, where we permanently get and return nodes, we cannot afford to access this operating-system-level memory management in each operation. Instead, we introduce an intermediate layer, which only occasionally has to access the operating-system memory management to get a large memory block, which it then gives out and receives back in small, constant-sized pieces, the nodes.

The efficiency of these `get_node` and `return_node` operations is really crucial for any dynamically allocated structure, but luckily we do not have to create a full memory management system; there are two essential simplifications. We deal only with objects of one size, as opposed to the `malloc` interface, which should provide memory blocks of any size, and we do not return any memory from the intermediate level to the system before the program ends. This is reasonable: the amount of memory taken by the intermediate layer from the system is the maximum amount taken by the data structure up to that

moment, so we do not overestimate the total memory requirement; we only fail to free it earlier for other coexisting programs or structures.

This allows us to use the *free list* as a structure for our dynamical allocation of nodes. The free list contains all the nodes not currently in use; whenever a `return_node` is executed, the node is just added to the free list. For the `get_node`, the situation is slightly more complicated; if the free list is not empty, we may just take a node from there. If it is empty and the current memory block is not used up, we take a new node from that memory block. Otherwise, we have to get a new memory block with `malloc` and create the node from there.

An implementation could look as follows:

```
typedef struct nd_t { struct nd_t *next;
              /*and other components*/  } node_t;
#define BLOCKSIZE 256
node_t *currentblock = NULL;
int    size_left;
node_t *free_list = NULL;

node_t *get_node()
{ node_t *tmp;
  if( free_list != NULL )
  {  tmp = free_list;
     free_list = free_list -> next;
  }
  else
  {  if( currentblock == NULL || size_left == 0)
     {  currentblock =
                (node_t *) malloc( BLOCKSIZE *
                           sizeof(node_t) );
        size_left = BLOCKSIZE;
     }
     tmp = currentblock++;
     size_left -= 1;
  }
  return( tmp );
}

void return_node(node_t *node)
{  node->next = free_list;
   free_list = node;
}
```

Dynamical memory allocation is traditionally a source of many programming errors and is hard to debug. A simple additional precaution to avoid some common errors is to add to the node another component, `int valid`, and fill it with different values, depending on whether it has just been received back by `return_node` or is given out by `get_node`. Then we can check that a pointer does indeed point to a valid node and that anything received by `return_node` has indeed been a valid node up to that moment.

1.5 Shadow Copies of Array-Based Structures

There is a systematic way to avoid the maximum-size problem of array-based structures at the expense of the simplicity of these structures. We simultaneously maintain two copies of the structure, the currently active copy and a larger-sized structure which is under construction. We have to schedule the construction of the larger structure in such a way that it is finished and ready for use before the active copy reaches its maximum size. For this, we copy in each operation on the old structure a fixed number of items from the old to the new structure. When the content of the old structure is completely copied into the new, larger structure, the old structure is removed and the new structure taken as the active structure and, when necessary, construction of an even larger copy is begun. This sounds very simple and introduces only a constant overhead to convert a fixed-size structure into an unlimited structure. There are, however, some problems in the details: the structure that is being copied changes while the copying is in progress, and these changes must be correctly done in the still incomplete larger copy. To demonstrate the principle, here is the code for the array-based stack:

```
typedef struct { item_t  *base;
                 int      size;
                 int  max_size;
                 item_t  *copy;
                 int copy_size; }    stack_t;

stack_t *create_stack(int size)
{   stack_t *st;
    st = (stack_t *) malloc( sizeof(stack_t) );
    st->base = (item_t *) malloc( size *
               sizeof(item_t) );
    st->max_size = size;
    st->size = 0; st->copy = NULL; st->copy_size = 0;
    return( st );
}
```

```
int stack_empty(stack_t *st)
{   return( st->size == 0);
}

void push( item_t x, stack_t *st)
{   *(st->base + st->size) = x;
    st->size += 1;
    if ( st->copy != NULL ||
    st->size >= 0.75*st->max_size )
    {  /* have to continue or start copying */
       int additional_copies = 4;
       if( st->copy == NULL )
       /* start copying: allocate space */
       {  st->copy =
          (item_t *) malloc( 2 * st->max_size *
           sizeof(item_t) );
       }
       /* continue copying: at most 4 items
          per push operation */
       while( additional_copies > 0 &&
              st->copy_size < st->size )
       {  *(st->copy + st->copy_size) =
                         *(st->base + st->copy_size);
          st->copy_size += 1; additional_copies -= 1;
       }
       if( st->copy_size == st->size)
       /* copy complete */
       {  free( st->base );
          st->base = st-> copy;
          st->max_size *= 2;
          st->copy = NULL;
          st->copy_size = 0;
       }
    }
}

item_t pop(stack_t *st)
{   item_t tmp_item;
    st->size -= 1;
    tmp_item = *(st->base + st->size);
    if( st->copy_size == st->size) /* copy complete */
    {  free( st->base );
       st->base = st-> copy;
       st->max_size *= 2;
       st->copy = NULL;
       st->copy_size = 0;
    }
```

```
    return( tmp_item );
}

item_t top_element(stack_t *st)
{   return( *(st->base + st->size - 1) );
}

void remove_stack(stack_t *st)
{   free( st->base );
    if( st->copy != NULL )
       free( st->copy );
    free( st );
}
```

For the stack, the situation is especially easy because we can just copy from the base until we reach the current top; in between, nothing changes. The threshold when to start copying (here, at 0.75*`size`), the size of the new structure (here, twice the previous size), and the number of items copied in each step (here, four items) must, of course, be chosen in such a way that copying is complete before the old structure overflows. Note that we can reach the situation when the copying is finished in two ways: by actual copying in the `push` and by deleting uncopied items in the `pop`.

In general, the connection between copying threshold size, new maximum size, and number of items copied is as follows:

{ if the current structure has maximum size $s_{\max}$,
{ and we begin copying as soon as its actual size has reached $\alpha s_{\max}$ (with $\alpha \geq \frac{1}{2}$),
{ the new structure has maximum size $2s_{\max}$, and
{ each operation increases the actual size by at most 1,

then there are at least $(1-\alpha)s_{\max}$ steps left to complete the copying of at most $s_{\max}$ elements from the smaller structure to the new structure. So we need to copy $\lceil \frac{1}{1-\alpha} \rceil$ elements in each operation to finish the copying before the smaller structure overflows. We doubled the maximum size when creating the new structure, but we could have chosen any size $\beta s_{\max}$, $\beta > 1$, as long as $\alpha\beta > 1$. Otherwise, we would have to start copying again before the previous copying process was finished.

In principle, this technique is quite general and not restricted to array-based structures. We will use it again in Sections 3.6 and 7.1. We can always try to overcome the size limitation of a fixed-size structure by copying its content to a larger structure. But it is not always clear how to break this copying into many

small steps that can be executed simultaneously with the normal operations on the structure, as in our example. Instead, we have to copy the entire structure in one step, so we cannot get a worst-case time bound, but only an amortized bound.

A final example of this technique and its difficulties is the realization of an extendible array. Normal arrays need to be declared of a fixed size, they are allocated somewhere in memory, and the space that is reserved there cannot be increased as it might conflict with space allocated for other variables. Access to an array element is very fast; it is just one address computation. But some systems also support a different type of array, which can be made larger; for these, accessing an element is more complicated and it is really an operation of a nontrivial data structure. This structure needs to support the following operations:

{ `create_array` creates an array of a given size,
{ `set_value` assigns the array element at a given index a value,
{ `get_value` returns the value of the array element at a given index,
{ `extend_array` increases the length of the array.

To implement that structure, we use the same technique of building shadow copies. There is, however, an additional problem here, because the structure we want to model does not just grow by a single item in each operation; the `extend_array` operation can make it much larger a single operation. Still, we can easily achieve an amortized constant time per operation.

When an array of size s is created, we allocate space for it, but more than requested. We maintain that the size of the arrays we actually allocate is always a power of 2, so we initially allocate an array of size $2^{\lceil \log s \rceil}$ and store the start position of that array, as well as the current and the maximum size, in a structure that identifies the array. Any access to an array element first has to look up that start position of the current array. Each time an `extend_array` operation is performed, we first check whether the current maximum size is larger than the requested size; in that case we can just increase the current size. Else, we have to allocate a new array whose size is the next number 2^k larger than the requested size, and copy every item from the old array to the new array. Thus, accessing an array element is always done in $O(1)$ time; it is just one in the direction of the pointer; but extending the array can take linear time in the size of the array. But the amortized complexity is not that bad; if the ultimate size of the array is $2^{\lceil \log k \rceil}$, then we have at worst copied arrays of size $1, 2, 4, \ldots, 2^{\lceil \log k \rceil - 1}$, so we spent in total time $O(1 + 2 + \cdots + 2^{\lceil \log k \rceil - 1}) = O(k)$ with those `extend_array` operations that did copy the array, and $O(1)$

with each `extend_array` operation that did not copy the array. Thus, we have the following complexity:

Theorem. An extendible array structure with shadow copies performs any sequence of n `set_value`, `get_value`, and `extend_array` operations on an array whose final size is k in time $O(n + k)$.

If we assume that each element of the array we request is also accessed at least once, so that the final size is at most the number of element access operations, this gives an amortized $O(1)$ complexity per operation.

It would be natural to distribute the copying of the elements again over the later access operations, but we have no control over the `extend_array` operations. It is possible that the next extension is requested before the copying of the current array is complete, so our previous method does not work for this structure. Another conceptual problem with extendible arrays is that pointers to array elements are different from normal pointers because the position of the array can change. Thus, in general, extendible arrays should be avoided even if the language supports them. A different way to implement extendible arrays was discussed in Challab (1991).

2
Search Trees

A search tree is a structure that stores objects, each object identified by a key value, in a tree structure. The key values of the objects are from a linearly ordered set (typically integers); two keys can be compared in constant time and these comparisons are used to guide the access to a specific object by its key. The tree has a root, where any search starts, and then contains in each node some key value for comparison with the query key, so one can go to different next nodes depending on whether the query key is smaller or larger than the key in the node until one finds a node that contains the right key.

This type of tree structure is fundamental to most data structures; it allows many variations and is also a building block for most more complex data structures. For this reason we will discuss it in great detail.

Search trees are one method to implement the abstract structure called dictionary. A dictionary is a structure that stores objects, identified by keys, and supports the operations find, insert, and delete. A search tree usually supports at least these operations of a dictionary, but there are also other ways to implement a dictionary, and there are applications of search trees that are not primarily dictionaries.

2.1 Two Models of Search Trees

In the outline just given, we supressed an important point that at first seems trivial, but indeed it leads to two different models of search trees, either of which can be combined with much of the following material, but one of which is strongly preferable.

If we compare in each node the query key with the key contained in the node and follow the left branch if the query key is smaller and the right branch

if the query key is larger, then what happens if they are equal? The two models of search trees are as follows:

1. Take left branch if query key is smaller than node key; otherwise take the right branch, until you reach a leaf of the tree. The keys in the interior node of the tree are only for comparison; all the objects are in the leaves.
2. Take left branch if query key is smaller than node key; take the right branch if the query key is larger than the node key; and take the object contained in the node if they are equal.

This minor point has a number of consequences:

{ In model 1, the underlying tree is a binary tree, whereas in model 2, each tree node is really a ternary node with a special middle neighbor.
{ In model 1, each interior node has a left and a right subtree (each possibly a leaf node of the tree), whereas in model 2, we have to allow incomplete nodes, where left or right subtree might be missing, and only the comparison object and key are guaranteed to exist.

So the structure of a search tree of model 1 is more regular than that of a tree of model 2; this is, at least for the implementation, a clear advantage.

{ In model 1, traversing an interior node requires only one comparison, whereas in model 2, we need two comparisons to check the three possibilities.

Indeed, trees of the same height in models 1 and 2 contain at most approximately the same number of objects, but one needs twice as many comparisons in model 2 to reach the deepest objects of the tree. Of course, in model 2, there are also some objects that are reached much earlier; the object in the root is found with only two comparisons, but almost all objects are on or near the deepest level.

Theorem. A tree of height h and model 1 contains at most 2^h objects.
A tree of height h and model 2 contains at most $2^{h+1} - 1$ objects.

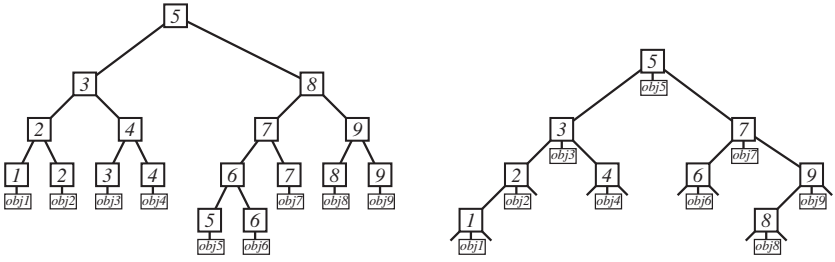
This is easily seen because the tree of height h has as left and right subtrees a tree of height at most $h - 1$ each, and in model 2 one additional object between them.

{ In model 1, keys in interior nodes serve only for comparisons and may reappear in the leaves for the identification of the objects. In model 2, each key appears only once, together with its object.

It is even possible in model 1 that there are keys used for comparison that do not belong to any object, for example, if the object has been deleted. By conceptually separating these functions of comparison and identification, this is not surprising, and in later structures we might even need to define artificial tests not corresponding to any object, just to get a good division of the search space. All keys used for comparison are necessarily distinct because in a model 1 tree, each interior node has nonempty left and right subtrees. So each key occurs at most twice, once as comparison key and once as identification key in the leaf.

Model 2 became the preferred textbook version because in most textbooks the distinction between object and its key is not made: the key is the object. Then it becomes unnatural to duplicate the key in the tree structure. But in all real applications, the distinction between key and object is quite important. One almost never wishes to keep track of just a set of numbers; the numbers are normally associated with some further information, which is often much larger than the key itself.

In some literature, where this distinction is made, trees of model 1 are called leaf trees and trees of model 2 are called node trees (Nievergelt and Wong 1973). Our preferred model of search tree is model 1, and we will use it for all structures but the splay tree (which necessarily follows model 2).



Search Trees of Model 1 and Model 2

A tree of model 1 consists of nodes of the following structure:

```
typedef struct tr_n_t {key_t      key;
                 struct tr_n_t   *left;
                 struct tr_n_t  *right;
            /* possibly additional information */
                        } tree_node_t;
```

We will usually need some additional balancing information, which will be discussed in Chapter 3. So this is just an outline.

From nodes of this type, we will construct a tree essentially by the following recursive definition: each tree is either empty, or a leaf, or it contains a special root node that points to two nonempty trees, with all keys in the left subtree being smaller than the key in the root and all keys in the right subtree being larger than or equal to the key in the root. This still needs some details; especially we have to specify how to recognize leaves. We will follow here the following convention:

- A node `*n` is a leaf if `n->right` = `NULL`. Then `n->left` points to the object stored in that leaf and `n->key` contains the object's key.

We also need some conventions for the root, especially to deal with empty trees. Each tree has a special node `*root`.

- If `root->left` = `NULL`, then the tree is empty.
- If `root->left` $\neq$ `NULL` and `root->right` = `NULL`, then `root` is a leaf and the tree contains only one object.
- If `root->left` $\neq$ `NULL` and `root->right` $\neq$ `NULL`, then `root->right` and `root->left` point to the roots of the right and left subtrees. For each node `*left_node` in the left subtree, we have `left_node->key` $<$ `root->key`, and for each node `*right_node` in the right subtree, we have `right_node->key` $\geq$ `root->key`.

Any structure with these properties is a correct search tree for the objects and key values in the leaves.

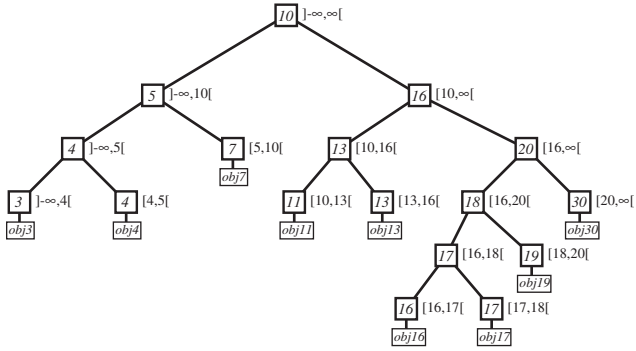
With these conventions we can now create an empty tree.

```
tree_node_t *create_tree(void)
{   tree_node_t *tmp_node;
    tmp_node = get_node();
    tmp_node->left = NULL;
    return( tmp_node );
}
```

2.2 General Properties and Transformations

In a correct search tree, we can associate each tree node with an interval, the interval of possible key values that can be reached through this node. The interval of `root` is $]-\infty, \infty[$, and if `*n` is an interior node associated with interval $[a, b[$, then `n->key` $\in [a, b[$, and `n->left` and `n->right` have as associated intervals $[a,$ `n->key`$[$ and $[$`n->key`$, b[$. With the exception of the intervals starting in $-\infty$, all these intervals are half-open, containing the left

endpoint but not the right endpoint. This implicit structure on the tree nodes is very helpful in understanding the operations on the trees.



INTERVALS ASSOCIATED WITH NODES IN A SEARCH TREE

The same set of (key, object) pairs can be organized in many distinct correct search trees: the leaves are always the same, containing the (key, object) pairs in increasing order of the keys, but the tree connecting the leaves can be very different, and we will see that some trees are better than others. There are two operations – the left and right rotations – that transform a correct search tree in a different correct search tree for the same set. They are used as building blocks of more complex tree transformations because they are easy to implement and universal.

Suppose `*n` is an interior node of the tree and `n->right` is also an interior node. Then the three nodes `n->left`, `n->right->left`, and `n->right->right` have consecutive associated intervals whose union is the associated interval of `*n`. Now instead of grouping the second and third intervals (of `n->right->left` and `n->right->right`) together in node `n->right`, and then this union together with the interval of `n->left` in `*n`, we could group the first two intervals together in a new node, and that then together with the last interval in `*n`. This is what the left rotation does: it rearranges three nodes below a given node `*n`, the rotation center. This is a local change done in constant time; it does not affect either the content of those three nodes or anything below them or above the rotation center `*n`. The following code does a left rotation around `*n`:

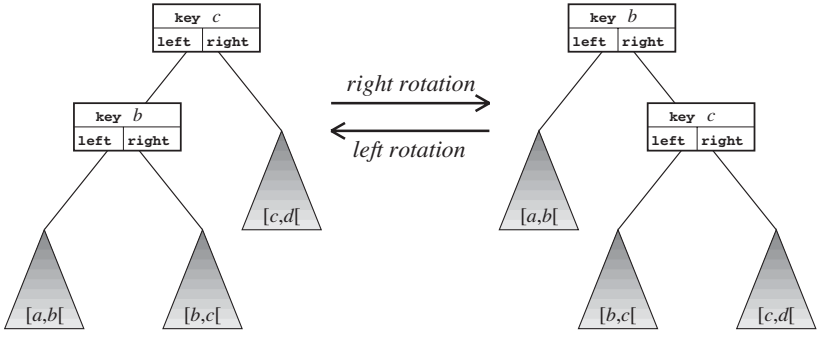
```
void left_rotation(tree_node_t *n)
{   tree_node_t *tmp_node;
    key_t         tmp_key;
    tmp_node = n->left;
    tmp_key  = n->key;
    n->left  = n->right;
```

```
    n->key   = n->right->key;
    n->right = n->left->right;
    n->left->right = n->left->left;
    n->left->left  = tmp_node;
    n->left->key   = tmp_key;
}
```

Note that we move the content of the nodes around, but the node `*n` still needs to be the root of the subtree because there are pointers from higher levels in the tree that point to `*n`. If the nodes contain additional information, then this must, of course, also be updated or copied.

The right rotation is exactly the inverse operation of the left rotation.

```
void right_rotation(tree_node_t *n)
{  tree_node_t *tmp_node;
   key_t         tmp_key;
   tmp_node = n->right;
   tmp_key  = n->key;
   n->right = n->left;
   n->key   = n->left->key;
   n->left  = n->right->left;
   n->right->left = n->right->right;
   n->right->right  = tmp_node;
   n->right->key   = tmp_key;
}
```



Left and Right Rotations

Theorem. The left and right rotations around the same node are inverse operations. Left and right rotations are operations that transform a correct search tree in a different correct search tree for the same set of (key, object) pairs.

The great usefulness of the rotations as building blocks for tree operations lies in the fact that they are universal: any correct search tree for some set of (key, object) pairs can be transformed into any other correct search tree by a sequence of rotations. But one needs to be careful with the exact statement of this property because it is obviously false: in our model of search trees, we can change the key values in the interior nodes without destroying the search tree property as long as the order relation of the comparison keys with the object keys stays the same. But the rotations, of course, do not change the key values. The important structure is the combinatorial type of the tree; any system of comparison keys is transformed correctly together with the tree.

Theorem. Any two combinatorial types of search trees on the same system of (key, object) pairs can be transformed into each other by a sequence of rotations.

But this is easy to see: if we apply right rotations to the search tree as long as any right rotation can be applied, we get a degenerate tree, a path going to the right, to which the leaves are attached in increasing order. So any search tree can be brought into this canonical shape using only right rotations. Because right and left rotations are inverse, this canonical shape can be transformed into any shape by a sequence of left rotations.

The space of combinatorial types of search trees, that is, of binary trees with n leaves, is isomorphic to a number of other structures (a Catalan family). The rotations define a distance on this structure, which has been studied in a number of papers (Culik and Wood 1982; Mäkinen 1988; Sleator, Tarjan, and Thurston 1988; Luccio and Pagli 1989); the diameter of this space is known to be $2n - 6$ for $n \geq 11$ (Sleator et al. 1988). The difficult part here is the exact value of the lower bound; it is simple to prove just $\Theta(n)$ bounds (see, e.g., Felsner 2004, Section 7.5).

2.3 Height of a Search Tree

The central property which distinguishes the different combinatorial types of search trees for the same underlying set and which makes some search trees good and others bad is the height. The height of a search tree is the maximum length of a path from the root to a leaf – the maximum taken over all leaves. Usually not all leaves are at the same distance from the root; the distance of a specific tree node from the root is called the depth of that node. As already observed in Section 2.1, the maximum number of leaves of a search tree of height h is 2^h. And at the other end, the minimum number of leaves is $h + 1$

because a tree of height h must have at least one interior node at each depth $0, \ldots, h-1$, and a tree with h interior nodes has $h+1$ leaves. Together, this gives the bounds.

Theorem. A search tree for n objects has height at least $\lceil \log n \rceil$ and at most $n-1$.

It is easy to see that both bounds can be reached.

The height is the worst-case distance we have to traverse to reach a specific object in the search tree. Another related measure of quality of a search tree is the average depth of the leaves, that is, the average over all objects of the distance we have to go to reach that object. Here the bounds are:

Theorem. A search tree for n objects has average depth at least $\log n$ and at most $\frac{(n-1)(n+2)}{2n} \approx \frac{1}{2}n$.

To prove these bounds, it is easier to take the sum of the depths instead of the average depth. Because the sum of depths can be divided in the depth of the a leaves to the left of the root and the depth of the b leaves to the right of the root, these sums satisfy the following recursions:

$$depthsum^{\min}(n) = n + \min_{\substack{a,b \geq 1 \\ a+b=n}} depthsum^{\min}(a) + depthsum^{\min}(b)$$

and

$$depthsum^{\max}(n) = n + \max_{\substack{a,b \geq 1 \\ a+b=n}} depthsum^{\max}(a) + depthsum^{\max}(b);$$

with these recursions, one obtains

$$depthsum^{\min}(n) \geq n \log n$$

and

$$depthsum^{\max}(n) = \frac{1}{2}(n-1)(n+2)$$

by induction. In the first case, one uses that the function $x \log x$ is convex, so $a \log a + b \log b \geq (a+b) \log (a+b)/2$.