

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, rst 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



2

Cambridge University Press 978-0-521-88037-4 - Advanced Data Structures Peter Brass Excerpt More information

1 Elementary Structures

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[\infty];

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 )</pre>
    {
        stack[i++] = x ; return(0);
    }
    else
       return( -1);
}
item_t pop(void)
    return( stack[ --i] );
{
}
```



1.1 Stack 3

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:



4 1 Elementary Structures

```
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.



1.1 Stack 5

```
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

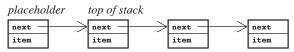


6

Cambridge University Press 978-0-521-88037-4 - Advanced Data Structures Peter Brass Excerpt More information

1 Elementary Structures

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:

© Cambridge University Press



1.1 Stack 7

```
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) );
```

© Cambridge University Press

www.cambridge.org



8

Cambridge University Press 978-0-521-88037-4 - Advanced Data Structures Peter Brass Excerpt More information

1 Elementary Structures

```
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



1.2 Queue

9

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:

```
\{ \mbox{ 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[\infty];

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;
```



10 1 Elementary Structures

```
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 \rightarrow rear = ((qu \rightarrow rear + 1) qu \rightarrow 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);
}
```