# Lecture Notes on Data Structure Invariants

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 12
October 5, 2010

## 1 Introduction

In this lecture we will highlight *data structure invariants*, one of the important and recurring themes in algorithm design and implementation. We use a simple text buffer as an example and implement it using doubly-linked lists.

Before we launch into the particular data structures we need, let's review the role of contracts that we have seen so far. The *preconditions* for a function establish and document the correct way of calling it, while the *postconditions* promise what will hold when the function returns. Together, they allow us to abstract away from exactly *how* a function computes, saying only *what* it computes. This allows us, for example, to substitute a different implementation of a function with confidence: as long as we continue to satisfy our side of the contract, that is, the postconditions, the client code should continue to work correctly if it adhered to its obligation to satisfy the preconditions. So pre- and post-conditions are function interfaces that go beyond the simple type information that is always checked by a compiler. In order to verify the correctness of our implementation we use the *loop invariants* to localize reasoning within a function.

Data structures, such stacks or queues or associative arrays have a similar overall structure. The *interface* to the data structure declares some types and functions to create and manipulate elements of these types. Just as the internals of a function should be hidden from a client, this implementation of the data structure should not be available to the client so that we can improve or substitute the implementation without breaking clients of the data structure. The pre- and post-conditions of the functions in the interface play the role of the pre- and post-conditions of stand-alone functions that we explored at the beginning of the course. Reasoning about the internals of a data structure implementation requires an analogue of loop invariants. The qualitative difference is that the computation of interest does not take place inside a single loop, but is spread out across many calls to functions in the interface to the data structure. For example, we might create a queue and then have many calls inserting and deleting elements. The properties of the data structure that must hold across these calls at the interface are called *data structure invariants*. Usually, we assume that they hold when a function in the interface is called, and have to ensure that they still hold when the function returns.

The data structure invariants play multiple roles, each of which we discuss with our example in some more detail. The first is simply to ensure a minimal consistency so that the stored information is meaningful. For example, we specified that the `front` and `back` pointers of a queue are not `NULL`. The second is to ensure the correct operation of the algorithms on the data structure. For example, in a hash table, all data elements stored in the chain at index $i$ must have the property that the hash value of their key is $i$. Otherwise, we could not find the data element at all. The third role is to make sure that the operations on the data structure have the promised asymptotic complexity or efficiency. For example, in unbounded arrays we want to make sure that after resizing of the array only half of the array is full, so that our adding and deleting elements has constant amortized time.

At this point it should become clear how central data structure invariants are. They are critical in all phases: design, analysis, and implementation of data structures. In C0, there are no new primitives to express and enforce data structure invariants. Instead, we write functions to check them and then use these in the pre- and post-conditions of the functions at the interface to the data structure. Moreover, we will often need to use them in internal functions or loops in order to ascertain that they are preserved by the data structure implementation.

Now on to our example.

## 2 A Text Buffer

A *text buffer* is a simple abstract type that supports the kind of operations that have to be performed in a text editor. Conceptually, we have a sequence of characters and a *point*. We insert characters into the buffer, and delete characters from the buffer at the point.

For example, the state of the buffer might be:

$$\begin{array}{ccccccc} a & e & b & & x & j & w \\ & & & \uparrow & & & \\ & & & \text{point} & & & \end{array}$$

To insert a character $c$, we place it just before the point. In the example above, we would obtain:

$$\begin{array}{cccccccc} a & e & b & c & & x & j & w \\ & & & & \uparrow & & & \\ & & & & \text{point} & & & \end{array}$$

Deleting a character is just the reverse: it deletes the character just before the point. So if we apply `delete` in the situation after inserting $c$, we are back to the first diagram.

We can also move the point *forward* or *backward*. After a move forward by one character in the above picture we have

$$\begin{array}{cccccccc} a & e & b & c & x & & j & w \\ & & & & & \uparrow & & \\ & & & & & \text{point} & & \end{array}$$

If we now move backward we are at the same situation as before.

If the point is before the first character, as in the following diagram,

$$\begin{array}{cccccccc} & a & e & b & c & x & j & w \\ \uparrow & & & & & & & \\ \text{point} & & & & & & & \end{array}$$

moving backward will have no effect, and trying to delete a character will yield an error. Similarly, if the point is to the right of the last character,

$$\begin{array}{cccccccc} a & e & b & c & x & j & w & \\ & & & & & & & \uparrow \\ & & & & & & & \text{point} \end{array}$$

then moving forward will have no effect.

## 3 Text Buffer Interface

Here is the text buffer interface.

```
typedef struct tbuf* tbuf;
tbuf tbuf_new();                /* create new edit buffer */
bool tbuf_empty(tbuf B);        /* is it empty? */
void insert_char(tbuf B, char c);  /* insert character after point */
bool at_min(tbuf B);            /* is point at minimum buffer position */
void delete_char(tbuf B);       /* delete character before point */
void forward_chars(tbuf B, int n);  /* move forward n chars */
void backward_chars(tbuf B, int n);  /* move backward n chars */
string tbuf_to_string(tbuf B);  /* convert edit buffer to string */
```

We haven't discussed the last function which we added for testing purposes. It converts the text buffer to a string.

This interface defines the type `tbuf`, which is provided by the implementation and used, abstractly, by the client. Unfortunately, C and C0 do not provide a way to enforce this abstraction. Our approximation is to define `tbuf` as a pointer to a `struct tbuf`, where the struct is not yet defined. It will be provided by the implementation of the interface.

We now discuss a few implementation ideas for text buffers and analyze their complexity on a buffer of size $n$.

## 4 Text Buffers as Arrays

This first idea is to implement a buffer as an array, possibly an unbounded array so it can grow as characters are inserted. Moving the point forward or backward are constant-time operations ($O(1)$). However, inserting or deleting a character is an $O(n)$ operation since we have to move the characters after the point to avoid any gaps. On the other hand, the representation is quite compact. A buffer for editing a large file takes about almost exactly the amount of space as a file itself.

The space efficiency of the representation makes the idea interesting for real editor implementation such as Emacs or vim. In Emacs, a buffer is represented as an array with a *gap*.[1] Insertion is into the gap and deletion is from the gap, which are constant time operations. These constant time

[1]See, for example, http://www.gnu.org/s/emacs/manual/html_node/elisp/Buffer-Internals.html#Buffer-Internals

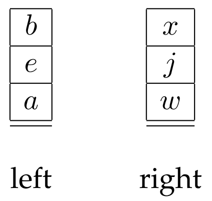
operations allow us to amortize the cost of moving large segments of the buffer when that gap has to be moved, or a new gap has to be created.

## 5 Text Buffers as Two Stacks

Another possibility is to use an idea you explored in the first midterm. We can represent the text buffer via two stacks, one containing the characters to the left of point and one containing the characters to the right of point. For example, the buffer

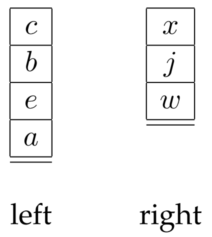
$$\begin{array}{cccccccc} a & e & b & & x & j & w \\ & & & \uparrow & & & \\ & & & \text{point} & & & \end{array}$$

would be



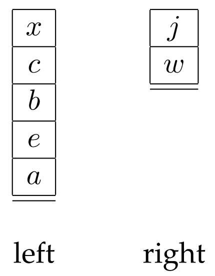
where the top of the stack is displayed at the top.

To insert a character we just push it onto the left stack. For example, inserting $c$ into the buffer displayed above yields



Deleting a character from the buffer just pops it from the left stack, leading us back to the first display.

Moving backward and forward is also straightforward. Moving forward one character just pops an element from the right stack and pushes into onto the left stack. In the buffer just above:



Moving backward one character just does the opposite, popping a character from the left stack to and pushing it onto the right stack. In the above situation we just return to the previously displayed buffer.

All of these operations are constant time, if pushing and popping from stacks are constant time. However, moving by $m$ characters forward or backward is not requires $O(m)$ operations. Moreover, if we implement stacks via linked lists, this representation is not particularly space efficient since we need a pointer for every character. In C, a character just takes 1 byte, while a pointer on modern systems is either 4 or 8 bytes. Thus, when editing a large file, the space requirement would be a 4 or 8 times as much as the size of the file, which is not acceptable. Therefore, the gap array representation seems preferable.

## 6 Doubly-Linked Lists

When files get very large, even the gap array representation may have unacceptable time overhead, needing to move large sections of the file in an array. A possible way out would be to represent the buffer as a linked list of arrays, each representing a *segment* of the file smaller than some maximum size. The character sequence making up the buffer is then the concatenation of the all the segments, an operation that hopefully would never have to be performed explicitly. However, we may move forward or backward across the segment boundaries, so an ordinary linked list representation as we have used for stacks and queues is awkward. It is more convenient to have a *doubly-linked list*, where each node points to its predecessor node as well as to its successor node.

For simplicity, we study doubly-linked lists not using buffer segments, but text buffers directly. So each node of a doubly-linked list contains a character as well as pointers to the previous and next nodes called `prev` and `next`, respectively.

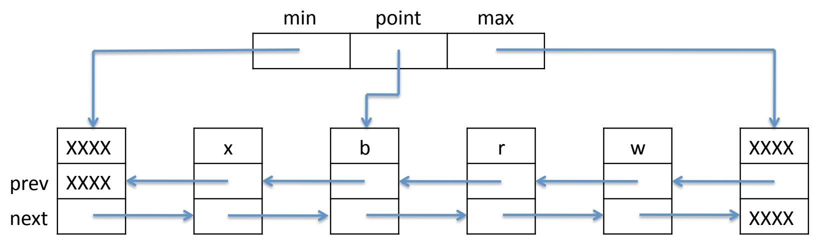
```
typedef struct dll* dll;
struct dll {
  char data;  /* data item */
  dll prev;   /* prev node, where p->prev->next == p, if valid */
  dll next;   /* next node, where p->next->prev == p, if valid */
};
```

It is important to remember that this declaration is internal to the implementation of text buffers and should not be referenced by client code.

A text buffer `struct tbuf` contains a pointer to the first node called `min`, a pointer to a node just beyond the end of the buffer called `max` and the `point` itself. Since there is no natural place *between* two nodes that `point` could refer to, we could either have a dummy node at `point` whose character datum we ignore, or we can just arbitrarily decide to call the node before or after `point`. We use this last idea and have `point` be the node just preceding the insertion point in the buffer.

```
struct tbuf {
  dll min;                         /* min buffer position, only min->next valid */
  dll point;                       /* min <= point < max */
  dll max;                         /* max buffer position, only max->prev valid */
  int size;                        /* number of characters */
};
```

This means that `point` could be `min` (if the point is before the first character in the buffer), or just below `max` (if the point is to the right of the last character in the buffer), but never equal to `max`. The following diagram illustrates a `tbuf`, ignoring the `size` field.



This buffer holds characters $x$, $b$, $r$, $w$, with the point between $b$ and $r$. The fields marked XXXX in the picture hold data that are irrelevant. They should never be used. The previous and next pointers are drawn as if they point to the middle of the struct before or after the current node, but the C0 language has no such interior pointers. In the implementation they point to the beginning of the struct, but this is awkward to represent faithfully with lines and arrows. So, for example, the prev pointer of the node labeled $r$ is equal to point which is equal to the next pointer of the node labeled $x$.

## 7 Text Buffer Invariants

For any data structure, before we launch into the implementation of the functions acting on it, we should make our invariants precise. As usual, we express them as a boolean function that tests, as far as possible, that the invariants are satisfied. Occasionally, some invariants are difficult or impossible to check, in which case we write the conditions in comments.

We develop the invariants incrementally, keeping the picture in mind. Recall that

```
typedef struct tbuf* tbuf;
```

which means that a value of type tbuf could be NULL. This would not be a valid text buffer, since even the empty buffer has min and max nodes and a point. We check these conditions at the beginning and return false if they are not satisfied. This allows us to assume that the are not NULL in the remainder of the function.

```
bool is_tbuf(tbuf B) {
  if (B == NULL) return false;        /* (1) */
  if (B->min == NULL) return false;   /* (2) */
  if (B->point == NULL) return false; /* (3) */
  if (B->max == NULL) return false;   /* (4) */
  ...
}
```

We number our invariants so we can refer to them when we reason about them. The next part of the checking function will traverse the doubly-linked list from left to right, always checking that the prev and next pointer line up correctly. We stop the iteration when the pointer $p$ has reached max.

```
  { dll p = B->min;
```

```
   while (p != B->max)
      //@loop_invariant p != NULL;           /* (6) */
      {
        if (p->next == NULL) return false;    /* (8) */
        if (p->next->prev != p) return false; /* (9) */
        p = p->next;
        if (p->prev->next != p) return false; /* (11), redundant w. (9) */
      }
    return true;
  }
```

Invariants (5), (7), and (10) are still missing from this fragment. Invariant (6) is an internal loop invariant; it is a precondition for calculating, for example, `p->next` without running the risk of $p$ being `NULL`. Invariant (8) ensures that the `next` pointer is never `NULL`, except possibly on `max`, which cannot be the case while inside the loop due to the exit test. Invariant (9) verifies that if we follow the `next` pointer and then return via the `prev` pointer, we end up again at $p$. We then advance $p$ by following the `next` pointer. We now check that following the `prev` pointer and then `next` we return back to the same place, although this is redundant since we already check the corresponding condition at (9). Some redundancies in these checks are acceptable, since they are not written for efficiency, but for correctness. They should not be called except in annotations, which are only used when their dynamic checking is explicitly enabled. In the current C0 compiler, this is accomplished with the `-d` flag.

The invariant function so far does not check that `point` is somewhere between `min` (inclusively) and `max` (exclusively). We can incorporate this by creating a flag `point_ok` which is initialized to `false`, and set to `true` if $p$ equals `point` during the traversal of the text buffer. We then return `true` only if `point_ok` is `true` after the iteration.

```
bool is_tbuf(tbuf B) {
  if (B == NULL) return false;        /* (1) */
  if (B->min == NULL) return false;   /* (2) */
  if (B->point == NULL) return false; /* (3), redundant with (7ab) */
  if (B->max == NULL) return false;   /* (4) */
  { dll p = B->min;
    bool point_ok = false;            // new
    while (p != B->max)
      //@loop_invariant p != NULL;           /* (6) */
      {
        if (p == B->point) point_ok = true;   /* (7a) */ // new
        if (p->next == NULL) return false;    /* (8) */
        if (p->next->prev != p) return false; /* (9) */
        p = p->next;
        if (p->prev->next != p) return false; /* (11), redundant w. (9) */
      }
    return point_ok;                          /* (7b) */ // modified
  }
}
```

We make one more addition: we track the size of the text buffer with a counter and compare it to the stored `size` field.

```
bool is_tbuf(tbuf B) {
  if (B == NULL) return false;        /* (1) */
  if (B->min == NULL) return false;   /* (2) */
  if (B->point == NULL) return false; /* (3), redundant w. (7ab) */
  if (B->max == NULL) return false;   /* (4) */
  { dll p = B->min;
    bool point_ok = false;
    int i = 0;                        // new
    while (p != B->max)
      //@loop_invariant p != NULL;            /* (6) */
      {
        if (p == B->point) point_ok = true;   /* (7a) */
        if (p->next == NULL) return false;    /* (8) */
        if (p->next->prev != p) return false; /* (9) */
        p = p->next;
        i++;                                  /* (10a) */ // new
        if (p->prev->next != p) return false; /* (11), redundant w. (9) */
      }
    return point_ok && B->size == i-1;        /* (7b) and (10b) */ // modified
  }
}
```

The astute reader will notice that invariant (5) is still missing. Whenever we write functions (including those testing data structure invariants), we need to check that the functions terminate. The function above could get into an infinite loop if the end of the doubly-linked list is hooked up with the beginning. A student suggested that we can avoid this situation by setting the `prev` pointer of `min` to `NULL`, because even if we had a cycle, condition (9) would then be violated at the node that pointed to `min`. If the cycle completed to a later node, again condition (9) would have be violated at some earlier point.

So here is our final function

```
bool is_tbuf(tbuf B) {
  if (B == NULL) return false;        /* (1) */
  if (B->min == NULL) return false;   /* (2) */
  if (B->point == NULL) return false; /* (3), redundant w. (7ab) */
  if (B->max == NULL) return false;   /* (4) */
  if (B->min->prev != NULL) return false; /* (5), to avoid possible cycles */
  { dll p = B->min;
    bool point_ok = false;
    int i = 0;
    while (p != B->max)
      //@loop_invariant p != NULL;            /* (6) */
      {
        if (p == B->point) point_ok = true;   /* (7a) */
        if (p->next == NULL) return false;    /* (8) */
        if (p->next->prev != p) return false; /* (9) */
        p = p->next;
        i++;                                  /* (10a) */
        if (p->prev->next != p) return false; /* (11), redundant w. (9) */
      }
    return point_ok && B->size == i-1;        /* (7b) and (10b) */
  }
}
```

## 8 Creating a Text Buffer

The code to create a text buffer is straightforward, but we note on each line the invariant that is being established.

```
tbuf tbuf_new()
//@ensures is_tbuf(\result);
//@ensures tbuf_empty(\result);
{
  tbuf B = alloc(struct tbuf);  /* establishes (1) */
  dll min = alloc(struct dll);
  dll max = alloc(struct dll);
  min->prev = NULL;             /* establishes (5) */
  min->next = max;              /* ests (8) */
  max->prev = min;              /* ests (9) on min */
  /* max->next is irrelevant */
  B->min = min;                 /* ests (2) */
  B->point = min;               /* ests (3) and (7a) */
  B->max = max;                 /* ests (4) */
  B->size = 0;                  /* ests (10b) */
  return B;
}
```

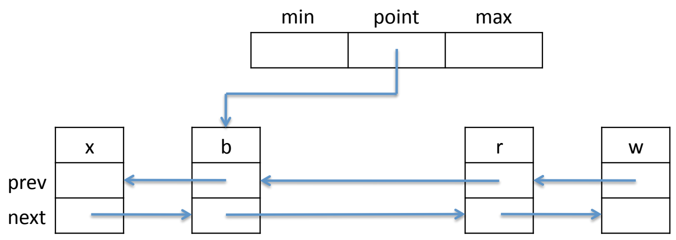
We can easily check that a buffer is empty in more than one way. For example:

```
bool tbuf_empty(tbuf B)
//@requires is_tbuf(B);
{
  return B->size == 0; /* or: B->min->next == B->max; */
}
```

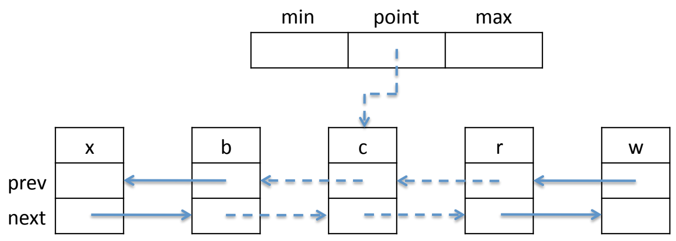
Note that calculating `B->size` cannot fail because $B$ is not `NULL` by invariant (1) and the precondition that $B$ is a valid text buffer.

## 9 Inserting a Character

In order to insert a new character into a text buffer, we have to allocate a new node reroute a number of pointers. Two strategies help us to obtain correct code. One is to draw a box-and-pointer diagram to make sure we understand how the nodes are connected. A second one is to explicitly refer to the invariants and (a) make sure that all pointer that are dereferenced are known to be not null, and (b) that the assignments reestablish the invariants. We show here diagrams before and after the insertion of the character $c$ after $b$. We ignore some of the parts of the earlier diagram that are not relevant to an insertion, simply leaving them blank.



After insertion, we have the following situation.



```
void insert_char(tbuf B, char c)
//@requires is_tbuf(B);
//@ensures is_tbuf(B);
{
  dll q = alloc(struct dll);
  q->data = c;
  q->prev = B->point;           /* ok by (1) */
  q->next = B->point->next;     /* ok by (3) */
  q->next->prev = q;            /* ok by (8) and (7); ests (9) */
  q->prev->next = q;            /* ok by (8); ests (11) */
  B->point = q;                 /* ests (3) and (7) */
  B->size++;                    /* ests (10) */
}
```

When checking this code, we have to make sure that all pointers we dereference are not null; the corresponding reason is given by reference to the data structure invariants in the `is_tbuf` function. We also want to make sure that the invariants are reestablished appropriately, so we annotate each line with the invariant that is explicitly established there. Please make sure to read the code above line by line and understand how it relates to the stated invariants.

It is also important to make sure that we update the pointers in the right order. Doing it in the wrong order might destroy a value we still need, or we may accidentally copy the wrong pointer value into a struct. Generally speaking, when we create a new node, it is safest to update its pointers first, because that does not overwrite any existing meaningful pointers. After that, we change the existing pointers in the auxiliary structure (here in the doubly-linked list), and finally we change the information in the main struct tracking the abstract type (here `struct tbuf`).

You are invited to read the complete implementation of the interface presented in this lecture in the file tbuf.c0 available on the course web pages.