CS360 Lecture notes -- Doubly Linked Lists

- Jim Plank
- Directory: **~plank/plank/classes/cs360/360/www-home/notes/Dllists**
- Lecture notes: **http://www.cs.utk.edu/~plank/plank/classes/cs360/360/notes/Dllists**
- Original notes: *Wed Aug 25 10:14:37 EDT 1999*
- Last modified: *Wed Jan 25 10:11:15 EST 2012*

Doubly Linked Lists

Doubly linked lists are like singly linked lists, except each node has two pointers -- one to the next node, and one to the previous node. This makes life nice in many ways:

- You can traverse lists forward and backward.
- You can insert anywhere in a list easily. This includes inserting before a node, after a node, at the front of the list, and at the end of the list.
- You can delete nodes very easily.

The API for doubly linked lists is in **dllist.h**. It defines a doubly-linked list node:

```
typedef struct dllist {
  struct dllist *flink;
  struct dllist *blink;
  Jval val;
} *Dllist;
```

Here are the operations supported by **dllist.o**:

- **Dllist new_dllist()**: Allocates and returns a new doubly linked list.
- **free_dllist(Dllist l)**: Destroys the list, calling **free()** on all allocated memory in the list. The list does not have to be empty.
- **dll_prepend(Dllist l, Jval val)**: Adds a new node at the beginning of the list. This node's value is **val**. **Dll_prepend()** has no return value.
- **dll_append(Dllist l, Jval val)**: Adds a new node at the end of the list. This node's value is **val**. **Dll_append()** has no return value.
- **dll_insert_b(Dllist n, Jval val)**: Adds a new node to the list right before the specified node. This node's value is **val**.
- **dll_insert_a(Dllist n, Jval val)**: Adds a new node to the list right after the specified node. This node's value is **val**.
- **Dllist dll_nil(Dllist l)**: Returns a pointer to the **nil** (sentinel) node for the list. You can think of **nil** as a node with no value, that begins and ends the list. Since **l** points to the sentinel node, **dll_nil** returns**l**. You don't need to call **dll_nil()** to access the sentinel node. You can just use **l**, although it makes your code more readable if you use **dll_nil()**.

- **Dllist dll_first(Dllist l)**: Returns a pointer to the first node in the list. If the list is empty, this returns the sentinel. As with **dll_nil()**, you don't need to call **dll_first(l)** -- you can just use **l->flink**.
- **Dllist dll_last(Dllist l)**: Returns a pointer to the last node in the list. If the list is empty, this returns the sentinel. As with **dll_nil()**, you don't need to call **dll_last(l)** -- you can just use **l->blink**.
- **Dllist dll_next(Dllist n)**: Returns a pointer to the next node in the list after **n**. If **n** is the last node on the list, then **dll_next(n)** returns the sentinel. As with **dll_first()**, you don't need to call **dll_next(n)** -- you can just use **n->flink**.
- **Dllist dll_prev(Dllist n)**: Returns a pointer to the previous node in the list before **n**. If **n** is the first node on the list, then **dll_prev(n)** returns the sentinel. As with **dll_last()**, you don't need to call**dll_prev(n)** -- you can just use **n->blink**.
- **int dll_empty(Dllist l)**: Returns whether **l** is empty.
- **Jval dll_val(Dllist n)**: Returns node **n**'s **val** field. Again, you don't need to use this, but sometimes it comes in handy.
- **int dll_delete_node(Dllist n)**: Deletes and frees node **n**.

Finally, there are two macros for traversing **dllists** forwards and backwards. **ptr** should be a **Dllist** and **list** should be a **Dllist**:

```
#define dll_traverse(ptr, list) \
  for (ptr = (list)->flink; ptr != (list); ptr = ptr->flink)
#define dll_rtraverse(ptr, list) \
  for (ptr = (list)->blink; ptr != (list); ptr = ptr->blink)
```

Implementation

The implementation of each **dllists** is as a circular doubly-linked list with a sentinel node. The code is in **dllist.c**.

The **typedef** for a **dllist** node is:

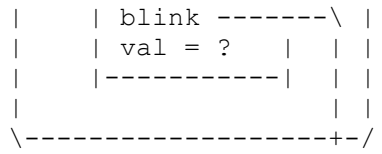
```
typedef struct dllist {
  struct dllist *flink;
  struct dllist *blink;
  Jval val;
} *Dllist;
```

Note that each node has two pointers -- a forward link (**flink**) to the next node on the list, and a backward link (**blink**) to the previous node on the list. A **Dllist** is a pointer to the sentinel node.

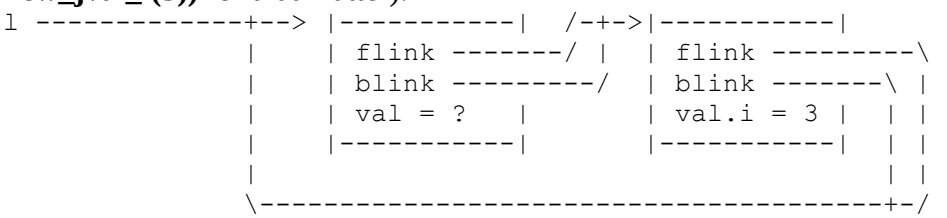
The list is circular in both directions -- the sentinel's **flink** points to the first node on the list, and its **blink** points to the last node on the list. The first node's **blink** points to the sentinel, as does the last node's**flink**.

Some ascii art: Here's an empty list **l**:

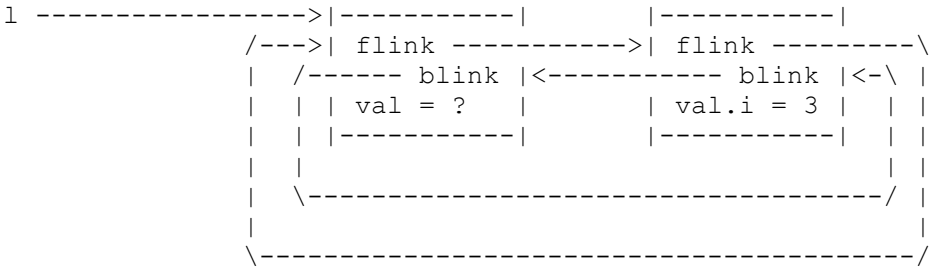
```
l -------------+--> |-----------|
               |    | flink ---------\
```



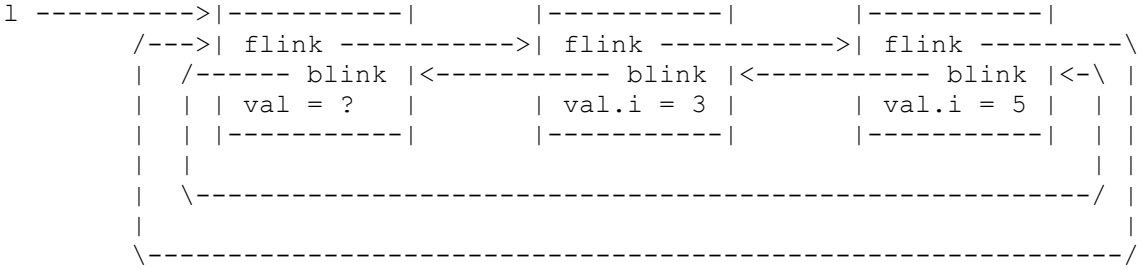
And here's that list after calling **dll_append(l, new_jval_i(3));**: (or **dll_prepend(l, new_jval_i(3))** for that matter).



Actually, it makes the drawing cleaner to have the back links go backwards:



Here's that list after calling **dll_append(l, new_jval_i(5));**:



I won't go over more examples with ascii art. You should be getting the hang of this by now.

Many of the procedure implementations are trivial procedures or macros:

```
Dllist new_dllist()
{
  Dllist d;

  d = (Dllist) malloc (sizeof(struct dllist));
  d->flink = d;
  d->blink = d;
  return d;
}
```

```
dll_empty(Dllist l)
{
  return (l->flink == l);
}
```

```
free_dllist(Dllist l)
{
  while (!dll_empty(l)) {
    dll_delete_node(dll_first(l));
  }
  free(l);
}
```

```
#define dll_first(d) ((d)->flink)
#define dll_next(d) ((d)->flink)
#define dll_last(d) ((d)->blink)
#define dll_prev(d) ((d)->blink)
#define dll_nil(d) (d)
```

The only subtle pieces of code are **dll_insert_b()** and **dll_delete_node**. With **dll_insert_b(n, v)** we **malloc()** a new node, set its value to **v**, and then link it into the list right before **n**. This means that we set the new node's **flink** field to **n**, and its **blink** field to **n->blink**. Then we set **n->blink** to the new node, and the old **n->blink**'s **flink** field to the new node. Here's the code:

```
dll_insert_b(Dllist node, Jval v)       /* Inserts before a given node */
{
  Dllist new;

  new = (Dllist) malloc (sizeof(struct dllist));
  new->val = v;

  new->flink = node;
  new->blink = node->blink;
  new->flink->blink = new;
  new->blink->flink = new;
}
```

Once we have **dll_insert_b()** the other three list insertion routines are simply calls to **dll_insert_b()**:

```
dll_insert_a(Dllist n, Jval val)        /* Inserts after a given node */
{
  dll_insert_b(n->flink, val);
}

dll_append(Dllist l, Jval val)     /* Inserts at the end of the list */
{
  dll_insert_b(l, val);
}

dll_prepend(Dllist l, Jval val)    /* Inserts at the beginning of the list */
{
  dll_insert_b(l->flink, val);
}
```

Deletion is pretty easy too. First you must remove the node **n**'s from the list by setting **n->flink->blink** to **n->blink** and by setting **n->blink->flink** to **n->flink**. Then you free **n**:

```
dll_delete_node(Dllist node)            /* Deletes an arbitrary iterm */
{
  node->flink->blink = node->blink;
  node->blink->flink = node->flink;
  free(node);
}
```

Usage examples

The first example is one of our standards: reversing standard input. This is simple enough to need no explanation. It's in **dllreverse.c**:

```
#include <stdio.h>
#include <string.h>
#include "fields.h"
#include "dllist.h"

main()
{
  IS is;
  Dllist l;
  Dllist tmp;

  is = new_inputstruct(NULL);
  l = new_dllist();

  while (get_line(is) >= 0) {
    dll_append(l, new_jval_s(strdup(is->text1)));
  }

  dll_rtraverse(tmp, l) printf("%s", jval_s(tmp->val));
}
```

The second example is another standard: printing the last **n** lines of standard input. We do this by reading standard input into a **Dllist**, and making sure that the **Dllist** always has at most **n** nodes. Then we print it out: The code is in **dlltail.c**:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "fields.h"
#include "dllist.h"

main(int argc, char **argv)
{
  IS is;
  int n;
  Dllist l;
  Dllist tmp;

  if (argc != 2) {
```

```
    fprintf(stderr, "usage: dlltail n\n");
    exit(1);
  }
  n = atoi(argv[1]);
  if (n < 0) {
    fprintf(stderr, "usage: dlltail n  -- n must be >= 0\n");
    exit(1);
  }

  is = new_inputstruct(NULL);
  l = new_dllist();

  while (get_line(is) >= 0) {
    dll_append(l, new_jval_s(strdup(is->text1)));
    if (is->line > n) {
      tmp = dll_first(l);
      free(jval_s(dll_val(tmp)));
      dll_delete_node(tmp);
    }
  }

  dll_traverse(tmp, l) printf("%s", jval_s(tmp->val));
}
```

A couple of notes about this program. First, you have to call **strdup()** to copy the string that you have read from standard input. Otherwise, **get_line()** will overwrite the string, and all of your lines will be the same (we've gone over this concept multiple times, but I like to keep hammering the point home).

Second, when you call **dll_delete_node()** it removes the node from the list and frees the node. However, if the **val** field points to data that has been allocated with **malloc()**, then that data is not freed automatically. That is because the **dllist** library has no idea what the data is. Perhaps you want it freed, or perhaps you don't, because it is on mutiple data structures.

In this program, you most definitely want to free the data -- suppose you read a file that has 10G worth of text and you simply want to read the last 10 lines -- if you don't free the data when you delete a node, you'll burn 10G of memory. If you do free the data, then you only store 10 lines in memory at a time.

A Last Example

The following assignment is from CS140, in C++:

COS: Columns of strings and doubles

You're going to hate this program. Sorry. You are going to write **cos.cpp**. This reads words from standard input and partitions them into doubles and non-doubles. On line 1, it will print the first

non-double and the first double. On line 2, it will print the second non-double and the second double. And so on. The format of each line should be:

- The non-double, padded to 30 characters and left justified.
- A space.
- The double, padded to 20 characters, right justified with four digits after the decimal point.

Thus, each line will be 51 characters. If there are the same number of doubles and non-doubles, then life is easy. However, if there are more doubles than non-doubles, you simply print 30 spaces when you get to the lines that have no non-doubles. If there are more non-doubles than doubles, then you simply print 20 spaces where the double would go.

You may assume that the words in standard input are 30 characters or less.

Examples:

```
UNIX> cat input1.txt
1 Fred 2 3 Binky Dontonio
UNIX> cos < input1.txt
Fred                                    1.0000
Binky                                   2.0000
Dontonio                                3.0000
UNIX> cat input2.txt
1 2 3 Fred
UNIX> cos < input2.txt
Fred                                    1.0000
                                        2.0000
                                        3.0000
UNIX> cat input3.txt
1 2 Fred Binky Dontonio
UNIX> cos < input3.txt
Fred                                    1.0000
Binky                                   2.0000
Dontonio
UNIX> cos < input3.txt | cat -e
Fred                                    1.0000$
Binky                                   2.0000$
Dontonio                                     $
UNIX>
```

In case you're wondering, "cat -e" prints standard input on standard output, and puts a '$' at the end of the line. It's nice to be able to see spaces at the end of a line.

The solution is to use two Dllists -- one for doubles and one for strings. When you read stdin, you use **sscanf()** to determine if a word is a double, and if it is, you put it on the double lists. Otherwise, you call**strdup()** and put the string on the string lists. Then you print out in three phases. In the first, you print all the lines which have both doubles and strings. When you're done with that loop, you may have doubles or strings leftover. The remaining two loops handle each case. The code is in **cos.c**:

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

```
#include <stdlib.h>
#include <string.h>
#include "fields.h"
#include "dllist.h"

main()
{
  Dllist dl, dtmp;  /* The list for doubles */
  Dllist sl, stmp;  /* The list for strings */
  double d;
  int i;
  IS is;

  dl = new_dllist();
  sl = new_dllist();
  is = new_inputstruct(NULL);

  while (get_line(is) >= 0) {
    for (i = 0; i < is->NF; i++) {
      if (sscanf(is->fields[i], "%lf", &d) == 1) {
        dll_append(dl, new_jval_d(d));
      } else {
        dll_append(sl, new_jval_s(strdup(is->fields[i])));
      }
    }
  }

  /* Print the lines that have both doubles and strings: */

  dtmp = dl->flink;
  stmp = sl->flink;

  while (dtmp != dl && stmp != sl) {
    printf("%-30s %20.4lf\n", stmp->val.s, dtmp->val.d);
    dtmp = dtmp->flink;
    stmp = stmp->flink;
  }

  /* Now print the lines that have strings only: */

  while (stmp != sl) {
    printf("%-30s %20s\n", stmp->val.s, "");
    stmp = stmp->flink;
  }

  /* And print the lines that have doubles only: */

  while (dtmp != dl) {
    printf("%-30s %20.4lf\n", "", dtmp->val.d);
    dtmp = dtmp->flink;
  }

  exit(0);
}
```