

An observation about lists

- Many other operations on list have a similar structure
 - Traverse through the list...
 - ... and while doing so, compute some value / perform some comparison / etc.
 - After traversing the list, return some value
- One approach is to write many such functions with this structure.
- Another approach is to write a more general-purpose function...
 - which traverses through the list...
 - ... and applies some function to each element in the list.
 - Let's call this function **apply**
 - It will take three arguments (the list; a function to be applied to each element on the list; and an argument for that function)

apply

```
/* apply: execute fn for each element of listp */  
  
void apply(Nameval *listp, void (*fn)(Nameval*, void*), void *arg)  
{  
    for ( ; listp != NULL; listp = listp->next) {  
        (*fn)(listp, arg); /* call the function */  
    }  
}
```

```
void (*fn)(Nameval*, void*),
```

Declare `fn` to be a pointer to a void-valued function (i.e., it is a variable that holds the address of a function that returns void).

Such a function takes two arguments: an address to a `Nameval` (list element) and a `void *` (a generic point to an argument for the function being passed in).

example: printing out all elements

```
/* apply: execute fn for each element of listp */  
  
void apply(Nameval *listp, void (*fn)(Nameval*, void*), void *arg)  
{  
    for ( ; listp != NULL; listp = listp->next) {  
        (*fn)(listp, arg); /* call the function */  
    }  
}  
  
void printnv(Nameval *p, void *arg)  
{  
    char *fmt;  
  
    fmt = (char *) arg;  
    printf(fmt, p->name, p->value);  
}
```

```
apply(nvlist, printnv, "%s: %x\n");
```

example: count of all elements

```
void inccounter(Nameval *p, void *arg)
{
    int *ip;

    /* p is not used -- all we care about is that this function
     * is called once per node.
     */
    ip = (int *)arg;
    (*ip)++;    /* Note the parentheses!!! */
}
```

```
int n;

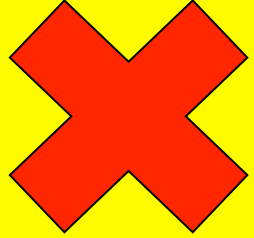
n = 0;
apply(nvlist, inccounter, &n);
printf("%d elements in nvlist\n", n);
```

Deleting elements from the list

- We have yet to see the use of `free` in the management of our lists
- Let's take the simplest case first: deleting the whole list
 - Here we must be rather careful
 - We cannot free an element if we need to dereference that same element later.
 - Also: `free` may itself modify the newly deallocated memory
- Must make good use of temporary variables

freeing the list

```
void bad_freeall(Nameval *listp)
{
    for ( ; listp != NULL; listp = listp->next ) {
        /* What is the value of listp->next after the next
         * operation?
         */
        free(listp);
    }
}
```



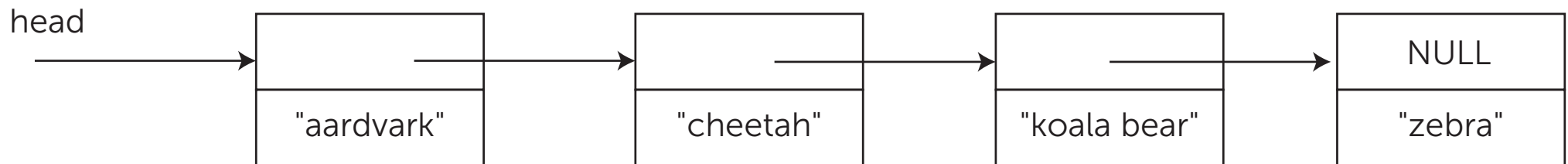
```
void freeall(Nameval *listp)
{
    Nameval *next;

    for ( ; listp != NULL; listp = next ) {
        next = listp->next;
        /* assume here the listp->name is freed someplace else */
        free(listp);
    }
}
```

Deleting elements from the list

- Deleting a single element requires more work than adding an element
 - Part of this is due to the consequences of using a singly-linked list.
 - It would be much easier with a doubly-linked list -- but then again, such a list does require twice as many pointers to be maintained.
- This is the place where bugs are often introduced
 - Yet if we are careful -- and correctly diagram what we intend to do -- then we can get it right the first time.
 - Recall the two main cases: are we deleting the first element? or one past the first

Singly-linked list



Deleting a single element

```
Nameval *delitem (Nameval *listp, char *name)
{
    Nameval *p, *prev;

    prev = NULL;
    for (p = listp; p != NULL; p = p->next) {
        if (strcmp(name, p->name) == 0) {
            if (prev == NULL) {
                listp = p->next;
            } else {
                prev->next = p->next;
            }
            free(p);
            return listp;
        }
        prev = p;
    }
    /* Ungraceful error handling, but gets the point across. */
    fprintf(stderr, "delitem: %s not in list", name);
    exit(1);
}
```

Trees

- Hierarchical data structure
 - We use them implicitly when navigating through the Unix file system
 - Also used in compilers (e.g., parse trees)
 - We also construct trees programmatically to get $O(\lg n)$ behavior rather than $O(n)$ for our algorithms (after some assumptions)
- Binary search tree
 - Simplest tree flavour, easiest to implement
 - Node in a binary search tree has a **value** and two pointers, **left** and **right**
 - The pointers lead to the node's children
 - All children to the left of a particular node have lower values than the node.
 - All children to the right of a particular node have greater values than the node.

Binary search tree example

