Fall 2004 Handout 8

—Source code on the Web at **http://i5.nyu.edu/˜mm64/x52.9232/src/doubly2.c**

```
/*
Let the user type in positive numbers, not necessarily unique.  Store them in
ascending order in a dynamically allocated doubly linked list.  The smallest
number will be at the head of the list, and the biggest number at the tail.
Then print the list from head to tail and from tail to head.

Each number is stored in a structure called a node_t.  Each node is malloc’ed
separately.  The variable head holds the address of the first node, or is NULL
if there are no node’s yet.  The variable tail holds the address of the last
node, or is NULL if there are no nodes yet.

The next field of each node holds the address of the next node, or is NULL if
there is no next node.  The prev field of each node holds the address of the
previous node, or is NULL if there is no previous node.  The 0 that the user
types in to terminate the input is not stored in the list.
*/

#include <stdio.h>
#include <stdlib.h>

typedef struct node_t {
    int n;  /* the number */
    struct node_t *prev; /* the address of the previous node */
    struct node_t *next; /* the address of the next node */
} node_t;

int main()
{
    node_t *head = NULL;
    node_t *tail = NULL;
    node_t *new; /* the new node that the user typed in */
    node_t *p, *q;   /* for looping through the list */
    int n;  /* each number that the user types in */

    printf(
    "Please type positive numbers.\n"
    "Press RETURN after each number.\n"
    "Type a negative number to delete the corresponding positive number.\n"
    "Type 0 when done.\n"
    );

    for (;;) {
        scanf("%d", &n);
        if (n == 0) {
            break;
    }

    if (n > 0) { /* Insert a new node (four cases). */
        new = malloc(sizeof(node_t));
        if (new == NULL) {
            fprintf(stderr, "Can’t allocate %lu bytes.\n", sizeof(node_t));
            return EXIT_FAILURE;
        }
        new->n = n;
        if (head == NULL) {
            /* Case 1: the list was empty. */
            new->next = new->prev = NULL;
            head = tail = new;
        }

        else if (n < head->n) {
            /* Case 2: insert new at the head of the list. */
            new->next = head;
            new->prev = NULL;
            head->prev = new;
            head = new;
        }

        else if (new->n >= tail->n) {
            /* Case 3: insert new at the tail of the list. */
            new->prev = tail;
            new->next = NULL;
            tail->next = new;
            tail = new;
        }

        else {
            /* Case 4: insert new into the interior of the list.
            Search the list to find the insertion point. */
            for (p = head; (q = p->next) != NULL; p = p->next) {
                if (q->n >= n) {
                    break;
                }
            }

            /* Insert new between the nodes that p and q point to. */
            new->next = q;
            new->prev = p;
            p->next = new;
            q->prev = new;
        }
    }

    else {  /* Delete an existing node. */
        n = -n;

        /* Let p point to the node to be deleted. */
        for (p = head; p != NULL; p = p->next) {
            if (p->n == n) {
                    goto found;
                }
            }
            fprintf(stderr, "The number %d is not on the list.\n", n);
            continue;

            found:;
            if (p == head) { /* Delete the first node. */
                head = p->next;
            } else {
                p->prev->next = p->next;
            }

            if (p == tail) { /* Delete the last node. */
                tail = p->prev;
            } else {
                p->next->prev = p->prev;
            }

            free(p);
        }
    }

    printf("Print the list from head to tail:\n");
    for (p = head; p != NULL; p = p->next) {
        printf("%d\n", p->n);
    }

    printf("\nPrint the list from tail to head:\n");
    for (p = tail; p != NULL; p = p->prev) {
        printf("%d\n", p->n);
    }

    return EXIT_SUCCESS;
}
```

A binary tree: K&R pp. 139−143

—Source code on the Web at `http://i5.nyu.edu/~mm64/x52.9232/src/tree.c`

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

typedef struct node_t {
    char *string;
    struct node_t *left;
    struct node_t *right;
} node_t;

node_t *insert(node_t *root, node_t *new);
void print(node_t *root);
void dismantle(node_t *root);

int main()
```

```
{
    char line[256];
    node_t *root = NULL; /* The tree is initially empty. */

    printf(
        "Press RETURN after each line,\n"
        "control-d after the RETURN after the last line.\n"
        "The lines will be output in alpahabetical order.\n"
    );

    while (gets(line) != NULL) {
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
            fprintf(stderr, "Can’t malloc memory for node to hold \"%s\".", line);
            return EXIT_FAILURE;
        }

        new->string = malloc(strlen(line) + 1);
        if (new->string == NULL) {
            fprintf(stderr, "Can’t malloc memory for \"%s\".\n", line);
            return EXIT_FAILURE;
        }
        strcpy(new->string, line);

        root = insert(root, new);
    }

    print(root);
    dismantle(root);
    return EXIT_SUCCESS;
}

node_t *insert(node_t *root, node_t *new)
{
    if (root == NULL) {
        /* Insert the new node into an empty tree. */
        root = new;
    }

    else if (strcmp(new->string, root->string) <= 0) {
        /* Insert the new node to the lower left of the root. */
        root->left = insert(root->left, new);
    }

    else {
        /* Insert the new node to the lower right of the root. */
        root->right = insert(root->right, new);
    }

    return root;
}

void print(node_t *root)     /* in order */
{
    if (root != NULL) {
        print(root->left);
        printf("%s\n", root->string);
        print(root->right);
    }
}

void dismantle(node_t *root)     /* post order */
{
    if (root != NULL) {
        dismantle(root->left);
        dismantle(root->right);

        free(root->string);
        free(root);
    }
}
```

Suppose that the above linked list and the above binary tree each contain n items. On the average, you would therefore have to make $\frac{n}{2}$ comparisons to find the correct insertion point in the list, but only $\log_2 n$ comparisons to find the correct insertion point in the binary tree.

A queue

—Source code on the Web at **http://i5.nyu.edu/˜mm64/x52.9232/src/queue.c**

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

typedef struct node_t {
    char *name;
    struct node_t *next;
} node_t;

/* The queue is initially empty. */
node_t *head = NULL;
node_t *tail = NULL;

void insert(const char *name);

int main()
{
    char name[256];
    node_t *new;
    node_t *doomed;
    node_t *p;

    for (;;) {
        printf("Type a name to insert, delete to delete, quit to quit: ");
        scanf("%s", name);
        if (strcmp(name, "quit") == 0) {
            break;
        }

        if (strcmp(name, "delete") != 0) {
            /* Insert a new node at the tail of the queue. */
            new = malloc(sizeof(node_t));
            new->name = malloc(strlen(name) + 1);
            strcpy(new->name, name);

            if (tail == NULL) {
                /* Insert first node into empty queue. */
                head = new;
            } else {
                tail->next = new;
            }
            tail = new;
        }

        else {  /* Remove a node from the head of the queue. */
            if (head == NULL) {
                fprintf(stderr, "The queue is already empty.\n");
                continue;
            }
            doomed = head;
            printf("%s\n", doomed->name);
            if (head == tail) {
                /* The queue is now empty. */
                head = tail = NULL;
            } else {
                head = doomed->next;
            }

            free(doomed->name);
            free(doomed);
        }
    }

    if (head == NULL) {
        printf("The queue is empty.\n");
    } else {
        printf("Here are the people who are still on the queue:\n");
        for (p = head;; p = p->next) {
            printf("%s\n", p->name);
            if (p == tail) {
                break;
            }
        }
    }

    return EXIT_SUCCESS;
}
```

A hash table: K&R pp. 143−145

Imagine an array whose subscripts were last names and whose values were firstnames:

```
    firstname["Clinton"] = "Bill";
    firstname["Lincoln"] = "Abe";
```

—Source code on the Web at **http://i5.nyu.edu/˜mm64/x52.9232/src/hash.c**

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

typedef struct node_t {
    char *lastname;
    char *firstname;
    struct node_t *next;
} node_t;

#define MULTIPLIER 31    /* for hash function */
#define N 1000
node_t *hashtable[N];

unsigned hash(const char *p);
node_t *lookup(const char *lastname);
void insert(const char *firstname, const char *lastname);
void print(node_t *starting_point);

int main()
{
    char firstname[256];
    char lastname[256];
    node_t *person;

    printf(
        "Type firstname lastname on a line by themselves.\n"
        "Press RETURN after each line,\n"
        "control-d after the RETURN after the last line.\n"
    );

    while (scanf("%s%s", firstname, lastname) == 2) {
        if (lookup(lastname) != NULL) {
            fprintf(stderr, "Sorry, the lastname \"%s\" already exists.\n",
                lastname);
            continue;
        }

        insert(firstname, lastname);
    }

    for (;;) {
        printf("Type a last name and press RETURN (or control-d to quit):\n");
        if (scanf("%s", lastname) != 1) {
            break;
        }

        person = lookup(lastname);
        if (person == NULL) {
            fprintf(stderr, "Sorry, lastname \"%s\" doesn’t exist.\n", lastname);
        } else {
            printf("%s %s\n", person->firstname, person->lastname);
        }
```

```c
    }

    return EXIT_SUCCESS;
}

/* Return the subscript in the hashtable where a lastname may be found. */
unsigned hash(const char *p)
{
    unsigned h = 0;

    for (; *p != ’\0’; ++p) {
        h *= MULTIPLIER;
        h += (unsigned char)*p;
    }

    return h % N;
}

node_t *lookup(const char *lastname)
{
    node_t *p;

    for (p = hashtable[hash(lastname)]; p != NULL; p = p->next) {
        if (strcmp(lastname, p->lastname) == 0) {
            break;
        }
    }

    return p;
}

void insert(const char *firstname, const char *lastname)
{
    const unsigned h = hash(lastname);
    node_t *const new = malloc(sizeof(node_t));

    new->lastname = malloc(strlen(lastname) + 1);
    strcpy(new->lastname, lastname);

    new->firstname = malloc(strlen(firstname) + 1);
    strcpy(new->firstname, firstname);

    new->next = hashtable[h];
    hashtable[h] = new;
}
```

□