Linked List

Based on slides from K. N. King and Dianna Xu
Bryn Mawr College
CS246 Programming Paradigm

Self-referential Structures

• A basic data type (building block) for complex data structures such as trees and linked lists.
• Structure tags (i.e. tnode, node) are required for self-referential structure declarations.

```
typedef struct tnode
{
    int x;
    struct tnode *left;
    struct tnode *right;
} Treenode;
```

```
typedef struct node
{
    int x;
    struct node *next;
} Listnode;
```

linked lists

• A linked list stores a lists of items (structs).
• Linked lists are typically unbounded, that is, they can grow infinitely.
• An array is a single consecutive piece of memory, a linked list is made of many pieces.
• A linked list offers quick insertion, deletion and reordering of the items.
• The last node in the list contains a null pointer.
• No ”random access” capability of an array.

Singly and Doubly Linked Lists

• A singly linked list has each struct containing only one pointer to the next.
• A doubly linked list has each struct containing both a pointer to the previous as well as the next struct in the list.

```
struct node
{
    int num;
    struct node *next;
};
typedef struct node Node;
Node *head = NULL; //the list is initially empty
Node *tail = NULL;
```

```
makenode
Node *
makenode
    (int x) {
    Node *new;
    if ( (new = (Node *) malloc( sizeof(Node) ) ) != NULL) {
        new->num = x; //(*new).num = x;
        scanf("%d", &new->num); An scanf example
        new -> next = NULL;
    } else {
        printf("Out of memory\n");
        exit(0);
    }
    return new;
}
```
void append (Node *p) {
    if (head == NULL) {
        head = p;
        tail = p;
    } else {
        tail->next = p;
        tail = p;
    }
}

void delete (Node *p) {
    Node *tmp = NULL, *prev = NULL;
    if ((p == head) && (p == tail))
        head = tail = NULL;
    else if (p == head)
        head = p->next;
    else {
        for (tmp = head, prev = NULL; tmp != p; prev = tmp, tmp = tmp->next);
        if (p == tail)
            tail = prev;
        prev->next = p->next;
    }
    free(p);
}

void insert_after (Node *p, Node *p2) {
    p->next = p2->next;
    if (p2 == tail)
        tail = p;
    p2->next = p;
}

void print() {
    Node *tmp;
    for (tmp = head; tmp != NULL; tmp = tmp->next)
        printf("%d ", tmp->num);
    printf("\n");
}

Node *search(int x) {
    Node *tmp;
    for (tmp = head; tmp != NULL; tmp = tmp->next)
        if (tmp->num == x)
            return tmp;
    return NULL;
}

int main() {
    Node *tmp;
    int i;
    for (i = 0; i < 10; i++) {
        tmp = makenode(i);
        append(tmp);
    }
    print();
    tmp = makenode(9);
    insert_after(tmp, head->next->next);
    delete(head->next);
    print();
}

void clear() {
    Node *tmp, *tmp2;
    for (tmp = head; tmp != NULL; tmp = tmp2){
        tmp2 = tmp->next;
        free(tmp);
    }
    head = tail = NULL;
}

• Note that this only works if structure Node does not contain any other pointers to memory
General Purpose Linked Lists

- **void** *
  - Generic pointer – just a memory address
  - Can be casted to any type

```c
struct llist_node {
    void *data;
    struct llist_node *prev;
    struct llist_node *next;
};
```

```c
typedef struct llist_node Lnode;
```

Avoid Memory Leaks

- Whenever dynamically allocated storages are in use, memory leaks are plentiful
- The problem is more evident when complicated data structures are used
  - Mixing: list of trees, trees of lists, etc
  - Nesting: list of lists of lists
- When implementing complex data structures, plan your **clear**/**release** functions very carefully

Shallow and Deep Copy

- There are two ways to make a copy of a linked list
  - Shallow copy:
    - The new list consist of duplicated pointers only
  - Deep copy:
    - The new list consist of duplicated data as well as pointers

Pointers to Pointers

- A variable can be modified by a function if and only if it is passed by reference/pointer.
- If the variable to be modified is a pointer itself, one must pass a pointer to pointer, i.e. one must always add an extra level of referencing.

```c
int make_node(Node **new) {
    *new=(Node *)malloc(sizeof(Node));
    if (*new != NULL) {
        return 1;
    } else {
        return 0;
    }
}
```