COMSW 1003-1

Introduction to Computer Programming in C

Lecture 19

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html
Basic Data Structures
Basic Data Structures

• So far, the only data structures we have seen to store data have been arrays (and structs)

• There are other (and potentially more useful) data structures that can be used
  – Lists
  – Trees

• Benefits:
  – Dynamically grow and shrink is easy
  – Search is faster
Linked Lists

• A chain of elements
• First element is called HEAD
• Each element (called NODE) points to the next
• The last node does not point to anything
• Like a treasure hunt with clues leading one to another
Pointers to structs

• Pointers can point to any type, including structs
• There is a particular way of accessing fields in a struct through a pointer: the → operator

```c
struct person {  
    int age;  
    char *name;  
};

struct person p1 = {15, "Luke"};  
struct person *ptr = &p1;  
ptr->age = 20;  // (*ptr).age = 20;  
printf("%s\n", ptr->name);  
```
Linked Lists

- Structure declaration for a node of a linked list

```c
struct ll_node {
    int value;
    struct ll_node *next;
};

typedef struct ll_node node;
```

![Diagram of linked list nodes with arrows indicating next pointers and NULL at the end]
Linked Lists
Initialization

```
struct ll_node {
    int value;
    struct ll_node *next;
};
```

```
node *head = (node *) malloc(sizeof(node));
head->value = 0;
head->next = NULL;
```

• First node (HEAD) of the list is just a pointer to the list, it not counted as an actual node in the list

• Value set to 0 (could be any number, maybe a counter)

• The list is still empty, there is only HEAD, so next is NULL (end of the list)
Linked Lists Initialization

node *head = (node *) malloc(sizeof(node));
head->value = 0;
head->next = NULL;

• First node (HEAD) of the list is just a pointer to the list, it not counted as an actual node in the list

• Value set to 0 (could be any number, maybe a counter)

• The list is still empty, there is only HEAD, so next is NULL (end of the list)
Linked Lists
Insert node in front

```c
int addNodeFront( int val, node *head ){
    node *newNode = (node *) malloc(sizeof(node));
    newNode->value = val;
    newNode->next = head->next;
    head->next = newNode;
    return 0;
}
```

```c
struct ll_node {
    int value;
    struct ll_node *next;
};
```
Linked Lists - Insert node in front

```c
int addNodeFront( int val, node *head ){
    1) node *newNode = (node *) malloc(sizeof(node));
    2) newNode->value = val;
    3) newNode->next = head->next;
    4) head->next = newNode;
    return 0;
}
```

```
addNodeFront( 7, head );
```

![Diagram showing the process of adding a node to the front of a linked list]
**Linked Lists - Insert node in front**

```c
int addNodeFront( int val, node *head ){
    1) node *newNode = (node *) malloc(sizeof(node));
    2) newNode->value = val;
    3) newNode->next = head->next;
    4) head->next = newNode;    return 0;
}
```

1) \$newNode\$ = \$malloc\$(\$sizeof\$(\$node\$));
2) \$newNode\$->\$value\$ = \$val\$;
3) \$newNode\$->\$next\$ = \$head\$->\$next\$;
4) \$head\$->\$next\$ = \$newNode\$;    \$return\$ \$0\$;

1-2
\$newNode\$

```
addNodeFront( 7, head );
addNodeFront( 5, head );
```
Linked Lists

Insert node at position N

```c
struct ll_node {
    int value;
    struct ll_node *next;
};

int addNode( int val, node *head, int pos ){
    node *newNode = (node*) malloc( sizeof(node) );
    newNode->value = val;
    int i;
    node *tmp = head;
    for(i=0 ; i<pos; i++)
        tmp = tmp->next;
    newNode->next = tmp->next;
    tmp->next = newNode;
    return 0;
}
```
Linked Lists - Insert node at position N

```c
int addNode( int val, node *head, int pos ){
    node *newNode = (node*) malloc( sizeof(node) );
    newNode->value = val;
    node *tmp = head;
    for(i=0 ; i<pos; i++)
        tmp = tmp->next;
    newNode->next = tmp->next;
    tmp->next = newNode;
    return 0;
}
```

```c
addNode( 4, head, 2 );
```
Linked Lists - Insert node at position N

```c
int addNode( int val, node *head, int pos ){
    2) node *tmp = head;

        for(i=0 ; i<pos; i++)
            tmp = tmp->next;

    3) newNode->next = tmp->next;

    4) tmp->next = newNode;

    return 0;
}
```

`addNode( 4, head, 2 );`
Linked Lists - Insert node at position N

```c
int addNode( int val, node *head, int pos ){
    node *tmp = head;
    2) for(i=0 ; i<pos; i++)
        tmp = tmp->next;
    3) newNode->next = tmp->next;
    4) tmp->next = newNode;
    return 0;
}
```

`addNode( 4, head, 2 );`
Linked Lists
Delete Node

```c
struct ll_node {
    int value;
    struct ll_node *next;
};

int removeNodePosition( node *head, int pos ){
    int i;
    node *tmp = head;
    for(i=0 ; i<pos; i++)
        tmp = tmp->next;
    node* tmp2 = tmp->next;
    tmp->next = tmp->next->next;
    free(tmp2);
    return 0;
}
```
Linked Lists - Delete Node

```c
int removeNodePosition( node *head, int pos ){
    int i;
    1) node *tmp = head;
       for(i=0 ; i<pos; i++)
           tmp = tmp->next;
    2) node* tmp2 = tmp->next;
       tmp->next = tmp->next->next;
    3) free(tmp2);
    return 0;
}
```

```
removeNode( head, 1 );
```
Linked Lists - Delete Node

```c
int removeNode( node *head, int pos ){
    int i;
    1) node *tmp = head;
    for(i=0 ; i<pos; i++)
        tmp = tmp->next;
    2) node* tmp2 = tmp->next;
        tmp->next = tmp->next->next;
    3) free(tmp2);
    return 0;
}
```

```
removeNode( head, 1 );
```
Linked Lists - Delete Node

```c
int removeNode( node *head, int pos ){
    int i;
    node *tmp = head;
    for(i=0 ; i<pos; i++)
        tmp = tmp->next;
    node* tmp2 = tmp->next;
    tmp->next = tmp->next->next;
    free(tmp2);
    return 0;
}
```

```
removeNode( head, 1 );
```
Linked Lists
Delete Whole List

```c
int destroyList( node **head ){
    node *tmp;
    while( (*head)->next != NULL ){
        tmp = (*head);
        (*head) = (*head)->next;
        free(tmp);
    }
    return 0;
}

destroyList( &head );
```

```c
struct ll_node {
    int value;
    struct ll_node *next;
};
```
Linked Lists
Delete Whole List

```c
int destroyList( node **head ){
    node *tmp;
    while( (*head)->next != NULL ){
        tmp = (*head);
        (*head) = (*head)->next;
        free(tmp);
    }
    return 0;
}
```

I need to pass head by reference, because I am changing it within the function.
Doubly linked lists

- Pointer to next AND previous node
- Faster backtracking

```c
struct dll_node {
    int value;
    struct dll_node *prev;
    struct dll_node *next;
};
```
Binary Trees

• Like lists, but each node has a pointer to two elements:
  – Left has a value < current node
  – Right has a value > current node

• First node is called ROOT

```c
struct t_node {
    int value;
    struct t_node *left;
    struct t_node *right;
};
```
Binary Trees

- Left has a value < current node
- Right has a value > current node

```c
struct t_node {
    int value;
    struct t_node *left;
    struct t_node *right;
};
```
Binary Trees

- Left has a value < current node
- Right has a value > current node

```c
struct t_node {
    int value;
    struct t_node *left;
    struct t_node *right;
};
```
Binary Trees

- Left has a value < current node
- Right has a value > current node

Nodes at the bottom level or without children are called LEAVES

```c
struct t_node {
    int value;
    struct t_node *left;
    struct t_node *right;
};
```
Binary Trees

Inserting number $x$ into a Binary Tree:

1. Start at root

2. if (current node is NULL)
   create new node and set node’s value to $x$

3. else

   if ($x \geq$ current node’s value )
   follow right pointer

   else
   follow left pointer

Go to 1
Binary Trees

Example: [1 12 6 23 17 90 8]
Binary Trees

Example: [ 1 12 6 23 17 90 8 ]

Find all elements < 10
Binary Trees

Example: [ 1 12 6 23 17 90 8 ]

Find all elements < 10

Binary tree requires 4 checks
Binary Trees

Example: [ 1 12 6 23 17 90 8 ]

Find all elements < 10

Binary tree requires 4 checks

Standard array or linked list require 7 checks
Trees Definitions

- **Root**: node with no parents. **Leaf**: node with no children.
- **Depth (of a node)**: path from root to node.
- **Level**: set of nodes with same depth.
- **Height** or depth (of a tree): maximum depth.
- **Size** (of a tree): total number of nodes.
- **Balanced binary tree**: depth of all the leaves differs by at most 1.

Height of tree = 3
Size = $n = 15$
Read PCP Chapter 17