2510 : Lecture 29

Albert's office hours : Mon 11:30-12:45 (SW2 319)
3:30-5:20

```
int remove (node **phead, int data){
    node **tracer;
    for (tracer = phead; *tracer != 0; tracer = &(*tracer)
                                                  ->next){
      if ((*tracer)->data == data)
        break;
    }
    assert (*tracer == 0 || *(tracer)-> == data);
```

Note: There are two ways we can get out of the above for loop

```
    if (*tracer != 0) {
        node *tmp = *tracer;
        *tracer = temp->next;
        free (tmp);
        return 1;   /* if node is deleted */
    }
    return 0;   /* if data is not found */
}
```

Question: Can we print a singly-linked list backwards?

Yes, using recursion.
↳ base case + a recursive case

code below ↘

Recursion continued...

```
void reverse_print (node *head){
    if (head == 0) /* base case */
        return;
    reverse_print (head -> next); /* recursive case */
    printf("%d \n", head -> data);
}
```

list 3 2 7 6 8 → 2 7 6 8 Pt 3
7 6 8 Pt 2

simpler version ↓

```
void revSimple (node *head){
    if (head != 0){
        revSimple (head -> next);
        printf("%d \n", head -> data);
}
```

- Another example of recursion.

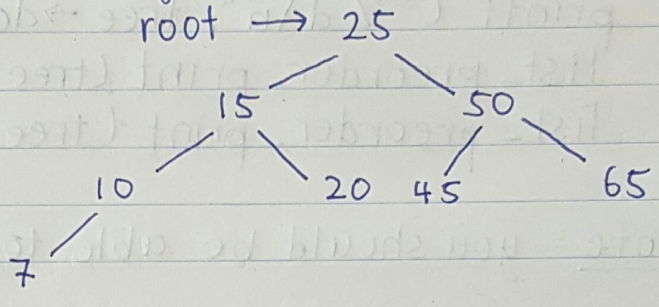
```
char * str_find (const char *s, int x){
    if (*s == '\0')
        return 0;
    return *s == x ? (char *) s : str_find (s+1, x);
```

▷ Binary Search Trees

- recursive data structure
- each node has a left subtree of right subtree
- for each node, data in left subtree is smaller
  data in right subtree is bigger

```
typedef struct node node;
struct node {     /* trees of integer */
    int data;
    node *left;
    node *right;
} node * root = 0;
```

ex. insert these integers into an empty binary search tree




```
int tree_insert (node **s ptree, int data) {
    if (*ptree == 0) {
        node * newNode = malloc (sizeof (node));
        if (newNode == 0)
            return 0;
        newNode → data = data;
        newNode → left = newNode → right = 0;
        *ptree = newNode;
        return 1;
    }
    if (data < (*ptree) → data)
        return tree_insert (&(*ptree) → left, data);
    if (data > (*ptree) → data)
        return tree_insert (&(*ptree) → right, data);
    return 0;
}
```

3 ways to traverse a binary tree:

Pre-order: current, left sub, right sub
In-order: left sub, current, right sub
Post-order: left sub, right sub, current

```
void list_preorder_print (node * tree) {
    if (tree != 0) {
        printf ("%d\n", tree->data);
        list_preorder_print (tree -> left);
        list_preorder_print (tree -> right);
    }
}
```

Note: you should be able to write post & in order

```
void list_destroy (node * tree) {
    if (tree != 0) {
        tree_destroy (tree -> left);
        tree_destroy (tree -> right);
        free (tree);
    }
}
```

- need to traverse tree and call free on each node
- need post-order traversal