Data Structures
Linked List & Binary Tree

Based on slides © McGraw-Hill
Additional material © 2004/2005 Lewis/Martin

Linked List Data Structure
• Sequence of nodes, each containing arbitrary data fields
• One or two references ("links") pointing to the next and/or previous nodes

Node 0 → Node 1 → Node 2

Single link linked list

Linked List vs. Array

Advantage of linked list
• Dynamic size (easy to add or remove nodes)

Advantage of array
• Easy/fast element access

Element access for a list
• Linked list elements can only be accessed sequentially e.g., to find the 5th element, you must start from head and follow the links through four other nodes

Link List Implementation in C

struct listNode {
  int val;
  struct listNode *next;
};
typedef struct listNode item;
Create a list of 10 elements

```c
item * curr, * head, *tail;
int i;
head = tail = NULL;
for(i=1;i<=10;i++) {
    curr = (item *)malloc(sizeof(item)); //allocate one item
    curr->val = i;
    curr->next = NULL;
    if(head == NULL){
        head = curr;
        tail = head;
    } else{
        tail->next = curr;
        tail = tail->next;
    }
}
```

Print the linked list element

```c
//reassign curr to head
curr = head;
while(curr != NULL) {
    printf("%d\n", curr->val);
    curr = curr->next ;
}
```

Deallocate all memory used by list

```c
//Deallocate
item * temp = head;
while(head != NULL){
    head= head->next;
    free(temp);
    temp = head;
}
```

Sorted List Example: Car Lot

Goal

- Create inventory database for used car lot
- Support the following operations
  - Search database for a particular vehicle
  - Add new vehicle to database
  - Delete vehicle from database

Implementation

- Since we don't know how many cars might be on lot at one time, we choose a linked list representation
- In order to have “faster” search, the database must remain sorted by vehicle ID
CarNode

Each vehicle has the following characteristics
• Vehicle ID, make, model, year, mileage, cost
• (And... pointer to next node)

typedef struct car_node CarNode;

struct car_node {
    int vehicleID;
    char make[20];
    char model[20];
    int year;
    int mileage;
    double cost;
    CarNode *next; /* ptr to next car in list */
};

Scanning the List

Scanning
• Searching, adding, and deleting require us to find a particular node in the list
• Scan list until we find node whose ID is >= one we’re looking for

CarNode *ScanList(CarNode *head, int searchID)
{
    CarNode *previous, *current;
    previous = head;
    current = head->next; /* Traverse until ID >= searchID */
    while (current!=NULL)
        if (current->vehicleID < searchID) {
            previous = current;
            current = current->next;
        } else {
            printf("Car already exists in database.");
            free(newNode);
            return previous;
        }
}

Adding a Node

void addCar(CarNode ** head, CarNode * newNode){..}

Steps
• Create new node with proper info (via malloc)
• Find node (if any) with a greater vehicleID (via ScanList)
• “Splice” the new node into the list (update next fields)

Excerpts from Code to Add a Node

newNode = (CarNode*) malloc(sizeof(CarNode));
/* initialize node with new car info */
...
prevNode = ScanList(*head, newNode->vehicleID);
nextNode = prevNode->next;
...
if ((nextNode == NULL)
|| (nextNode->vehicleID != newNode->vehicleID))
    prevNode->next = newNode;
    newNode->next = nextNode;
} else {
    printf("Car already exists in database.");
    free(newNode);
}
Deleting a Node

Steps
- Find the node that points to the desired node (via ScanList)
- Redirect that node’s pointer to the next node (or NULL)
- Free the deleted node’s memory

Trees

- Hierarchical structure with a set of linked nodes
- A node may contain a value or a condition or represent a separate data structure
- A node has zero or more child nodes
  - Most common type is Binary tree with each node having a left and right child

Binary Tree

```c
struct tnode{
    int val;
    struct tnode * left;
    struct tnode * right;
};
typedef struct tnode node;
```

E.g. Sorted Binary Tree

Structure
- At each node, value of the left child is less than value at node
- Also, value of the right child is greater than the value at node

Implications
- Faster lookup
- Slower insertion (can’t just prepend)

Optimizations
- Balancing
- Redistributing
Inserting an element to keep tree sorted

```c
node * root = NULL; //global ptr

void insert(node * item){
  if(root == NULL){
    root = item;
  } else{
    node * curr = root; //curr is your reference that keeps moving
    while(1){
      if(item->val < curr->val){
        if(curr->left == NULL){
          curr->left = item;
          break;
        } else{
          curr = curr->left;
        }
      } else if(item->val > curr->val){
        if(curr->right == NULL){
          curr->right = item;
          break;
        } else{
          curr = curr->right;
        }
      } else{ //duplicate
        break;
      }
    } //end while
  } //end else
} //end insert
```

Inserting an element to keep tree sorted (contd..)

```c
else if(item->val > curr->val){
  if(curr->right == NULL){
    curr->right = item;
    break;
  } else{
    curr = curr->right;
  }
} //end else if
else{
  //duplicate
  break;
}
} //end while
} //end else
} //end insert
```