CS 126 Lecture P7:
Trees

First Midterm

- When: 7pm, 10/20 (Wednesday)
- Where: MC46 (here)
- What: lectures up to (and including) today's
- Format: close book, minimum coding
- Preparation: do the readings and exercises

Why Learn Trees?

Culmination of the programming portion of this class!

- Comparison against **arrays** and **linked lists**
- Trees -- a versatile and useful **data structure**
- A naturally **recursive** data structure
- Applications of **stacks** and **queues**
- Reinforce our **pointer manipulation** knowledge

Outline

- **Searching and insertion *without* trees**
- Searching and insertion *with* trees
- Traversing trees
- Conclusion

- **Class list**

192-034-2006	Alam
201-212-1991	Baer
202-123-0087	Bagyenda
177-999-9898	Balestri
232-876-1212	Benjamin
122-999-3434	Berube
...	

- Desired operations
  - add student
  - return name, given ID number

SEARCH KEY

- Similar applications
  - online phone book
  - airline reservations
  - "symbol table"
  - ...

GOAL: fast search *and* insert even for huge databases

Encapsulating the Item Type Stored

- Define "Item.h" file to encapsulate item type

```
typedef int Key;
typedef struct{ Key key; char name[30]; } Item;
Item NULLitem = { -1, ""}
```

- A single item itself is an ADT
- So we don't see the internals of the item type when we implement searching and insertion
- So our code will work for <u>any</u> item type

Array Representation: Binary Search

```
Item items[13];
```

Ex: search for 25

	0	1	2	3	4	5	6	7	8	9	10	11	12	
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	
Keys	06	13	14	25	33	43	51	53	64	72	84	97	99	← 1st step
.	06	13	14	25	33	43								← 2nd step
.				25	33	43								← 3rd step
.				25										← 4th step

- Keep array of Items, in sorted order
- Use bisection method to find Item sought

[See also Lecture P6; Programs 2.2 and 12.6]

Array Representation: Binary Search

```
Item search(int l, int r, Key v)
  { int m = (l+r)/2;
    if (l > r) return NULLitem;
    if (v == st[m].key) return st[m];
    if (l == r) return NULLitem;
    if (v < st[m].key)
      return search(l, m-1, v);
    else return search(m+1, r, v);
  }
```

Cost of Binary Search

Q: How many "comparisons" to find a name?

A: lg N

divide list in half each time

Ex: 5000 -> 2500 -> 1250 -> 625 -> 312 -> 156 -> 78 -> 39 -> 18 -> 9 -> 4 -> 2 -> 1

log N = number of digits in decimal rep. of N

lg N = number of digits in binary rep. of N

lg(thousand) = 10

lg(million) = 20

lg(billion) = 30

$\log_2 N \equiv \lg N$

$N=2^x, \quad x=\log_2 N$

Without binary search, might have to look at everything, so savings is substantial for very large files.

Insertion into Sorted Array

Problem: insert operation is usually slow

Ex: to insert 49

.	0	1	2	3	4	5	6	7	8	9	10	11	12
.	06	13	14	25	33	43	51	53	64	72	84	97	99

have to move larger keys over one position

.	0	1	2	3	4	5	6	7	8	9	10	11	12	13
.	06	13	14	25	33	43	49	51	53	64	72	84	97	99

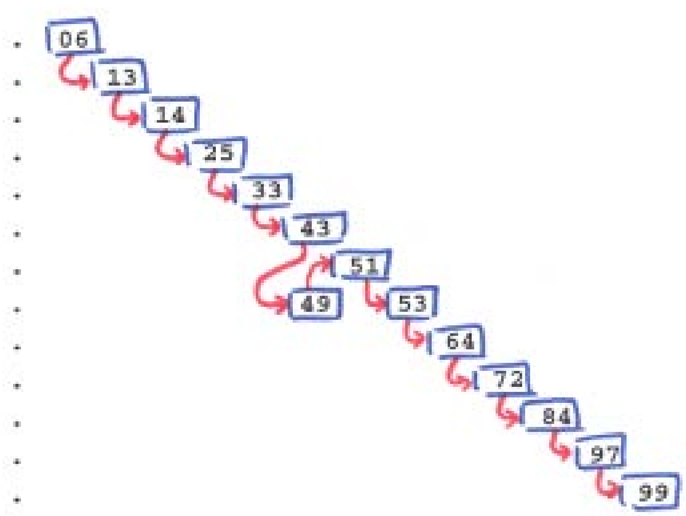
Linked List Representation

Keep items in a linked list

```
typedef struct STnode* link;
struct STnode { Item item; link next; };
```


Inserting into Linked List

- Advantage of linked representation
  can insert just by changing links
  (no need to "move" anything)



Exercises and Summary

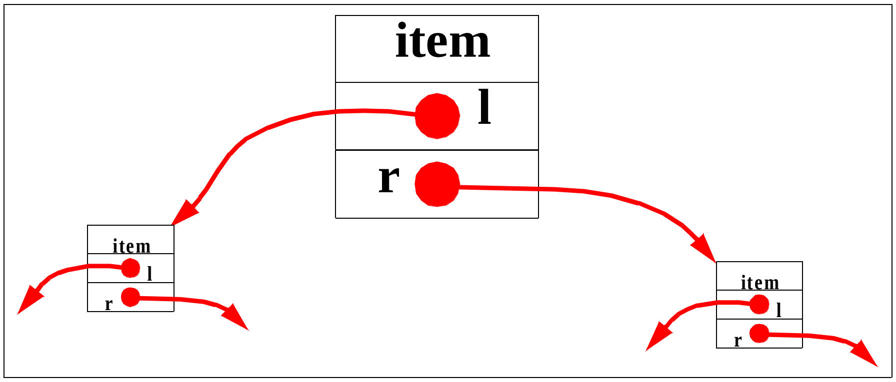
- Assuming a sorted linked list, try writing code for
  - both searching and insertion
  - using both loop and recursion
- Summary so far:

ARRAY: fast search, slow insert
LINKED LIST: slow search, fast insert

Outline

- ~~Searching and insertion *without* trees~~
- **Searching and insertion *with* trees**
- Traversing trees
- Conclusion

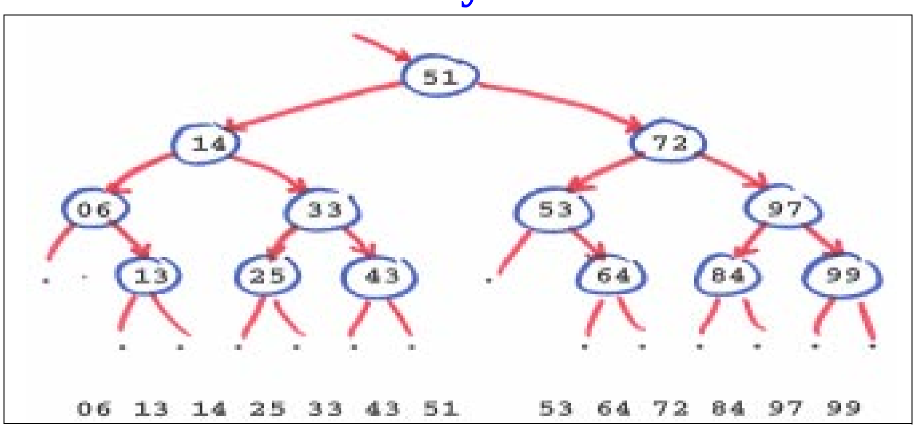
Declaring a Tree Type



- Use *two* links per node

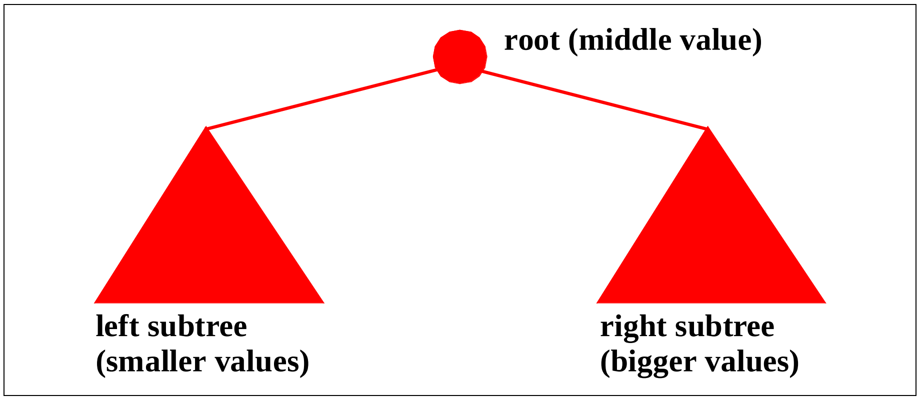
```
typedef struct STnode* link;
struct STnode { Item item; link l, r; };
```


Binary Tree



- Think of keys printed in order, left to right
  - take middle name for top, or "root" node
  - build tree recursively
    - "l" points to tree for left half
    - "r" points to tree for right half
- NULL links at bottom: "no information here"

Binary Search Tree Property



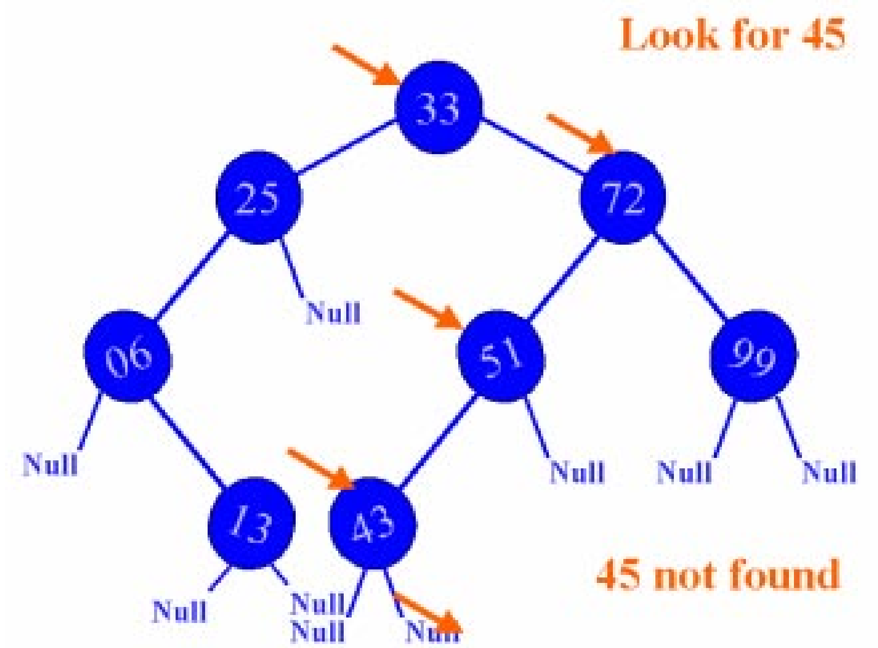
- Maintain ordering property for **all** subtrees
- Must maintain ordering property at all times (just like we keep an array or linked list sorted at all times)

Searching in Binary Search Tree

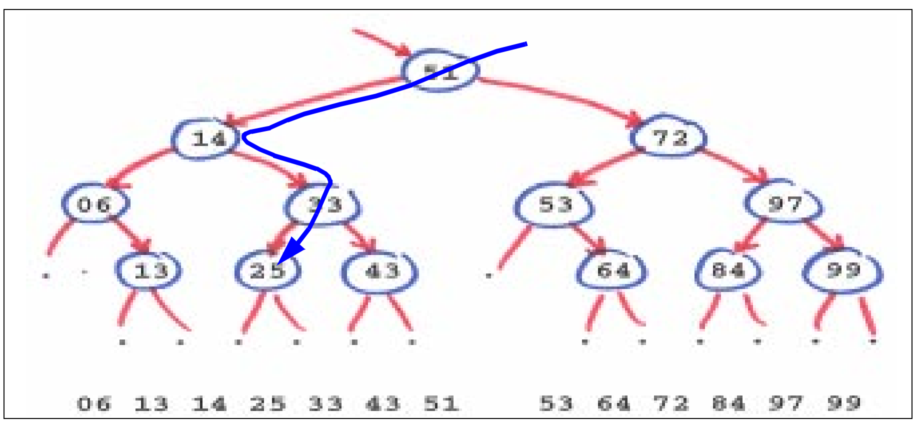
```
Item searchR(link h, Key v)
  {
    if (h == NULL) return NULLitem;
    if (v == h->item.key) return h->item;
    if (v < h->item.key)
      return searchR(h->l, v);
    else return searchR(h->r, v);
  }
Item STsearch(Key v)
  { return searchR(head, v); }
```

- Start at "head", link to the root
  - if current node has key sought, return
  - go left if key < key in current node
  - go right if key > key in current node

Search Demo



Search Cost



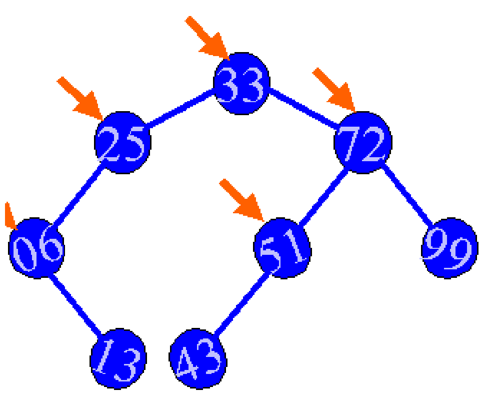
- Nodes examined on the search path roughly correspond to nodes examined during binary searching an array
- So the cost is same as binary searching an array (lg N)
- That is **if** the tree is balanced

Insertion into Binary Search Trees

```
link NEW(int item, link l, link r)
  { link x = malloc(sizeof *x);
    x->item = item; x->l = l; x->r = r;
    return x;
  }
link insertR(link h, Item item)
  { Key v = key(item);
    if (h == NULL)
      return NEW(item, NULL, NULL);
    if less(v, key(h->item))
      h->l = insertR(h->l, item);
    else h->r = insertR(h->r, item);
    return h;
  }
void STinsert(Item item)
  { head = insertR(head, item); }
```

- Search for key not in tree
  - ends on a NULL pointer
  - node ``belongs`` there
  - make a node, link it into the tree

Insertion Demo




```
Link
insert(Link h, Item it) {
  if (h == NULL)
    return newLeaf(it);
  if (less(key(it),
             key(h->item)))
    h->l = insert(h->l, it)
  else
    h->r = insert(h->r, it)
  return h;
}
```

More Notes on Binary Search Tree Insertion

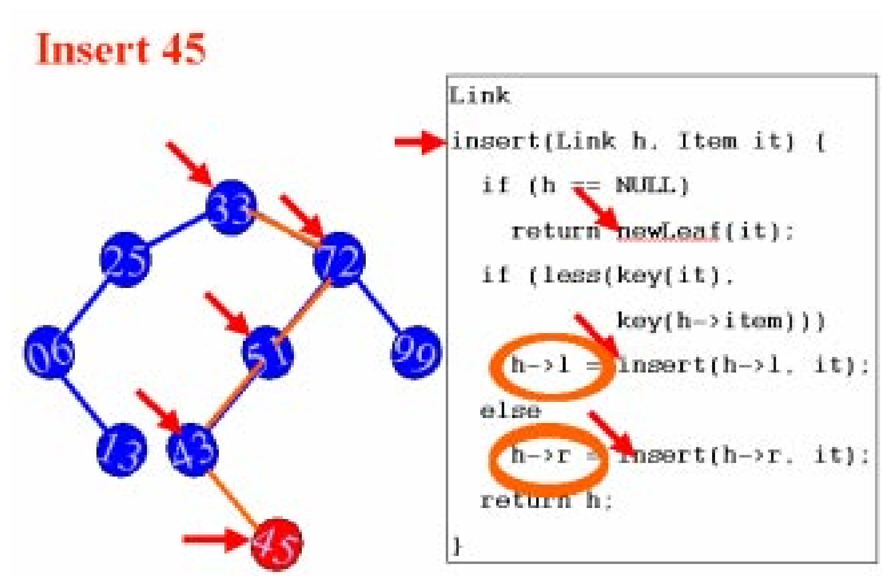
```
link insertR(link h, Item item)
  { Key v = key(item);
    if (h == NULL)
      return NEW(item, NULL, NULL);
    if less(v, key(h->item))
         h->l = insertR(h->l, item);
    else h->r = insertR(h->r, item);
    return h;
  }
```

attach the "new" subtree to the current root

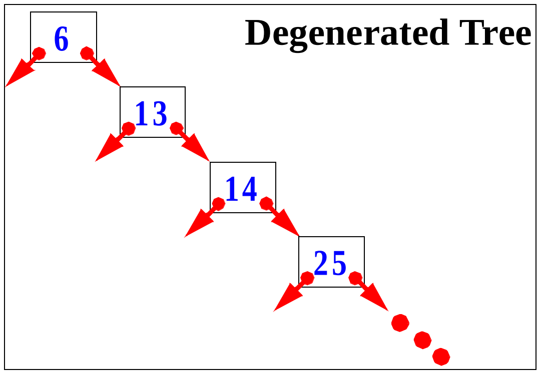
subtree containing the new value

- Each recursive call returns the root pointing to the subtree with the new value already inserted
- Do this for base case and inductive case

Another Insertion Demo



Insertion Cost



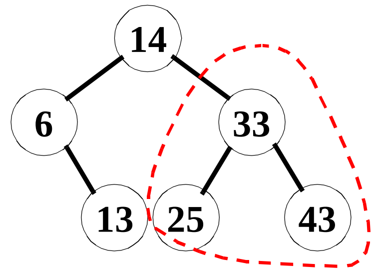
- "Normally", insertion is like search, so similar cost. But...
- Tree shape depends on key insertion order
  - sorted, reverse: degenerates to linked list
  - "random": avg. dist. to root is about 1.44 lg N

Outline

- ~~Searching and insertion *without* trees~~
- ~~Searching and insertion *with* trees~~
- **Traversing trees**
  - Goal: "visit" (process) each node in the tree
- Conclusion

Preorder Traversal

```
visit(link h) {
  printf(“%d %s “,
         h->item.ID,
          h->item.name);
}
traverse(link h) {
  if (h != NULL) {
    visit(h);
    traverse(h->l);
    traverse(h->r);
  }
}
```



14, 6, 13, 33, 25, 43

- Visit before recursive calls
- Generalizes to any tree: depth-first-traversal

Traversing Binary Trees

Goal: "visit" (process) each node in the tree

```
visit(link h)
  { printf("%d %s ", h->item.ID, h->item.name);
traverse(link h)
  {
    if (h != NULL)              <- preorder
      {
        traverse(h->l);
        visit(h);               <- inorder
        traverse(h->r);
      }                         <- postorder
  }
```

- Goal realized no matter what order the statements in the "if" are executed

Preorder: visit before recursive calls
Inorder: visit between recursive calls
Postorder: visit after recursive calls

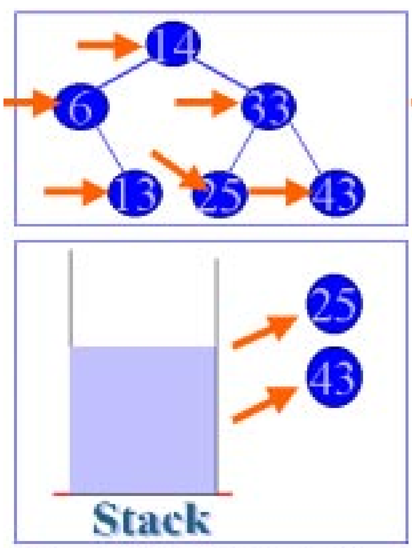
IMPORTANT NOTE:
inorder search provides "free" SORT in binary search trees!

Preorder Traversal with a Stack

- Visit the top node on the stack
  - push its children

```
traverse(link h)
  {
    STACKpush(h);
    while (!STACKempty())
      {
        h = STACKpop(); visit(h);
        if (h->r != NULL) STACKpush(h->r);
        if (h->l != NULL) STACKpush(h->l);
      }
  }
```


Preorder Traversal Demo



14 ,6,13,33,25,43
Output

```
Traverse(Link h) {
  stackPush(h);
  while (!stackEmpty()) {
    h = stackPop();
    printValue(h);
    if (h->r != NULL)
     stackPush(h->r);
    if (h->l != NULL)
     stackPush(h->l);
  }
}
```

Level Order Traversal

- Use a queue instead of a stack

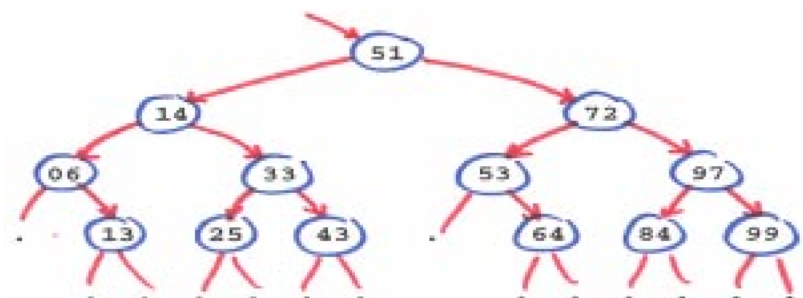
```
traverse(link h)
  {
    QUEUEput(h);
    while (!QUEUEempty())
      {
        h = QUEUEget(); visit(t);
        if (h->l != NULL) QUEUEput(h->l);
        if (h->r != NULL) QUEUEput(h->r);
      }
  }
```

Visits nodes in order of distance from root

- Works for general trees
- Generalizes to BREADTH-FIRST SEARCH in graphs

P7.12

Level Order Traversal Example



Level order traversal of tree on slide 5:

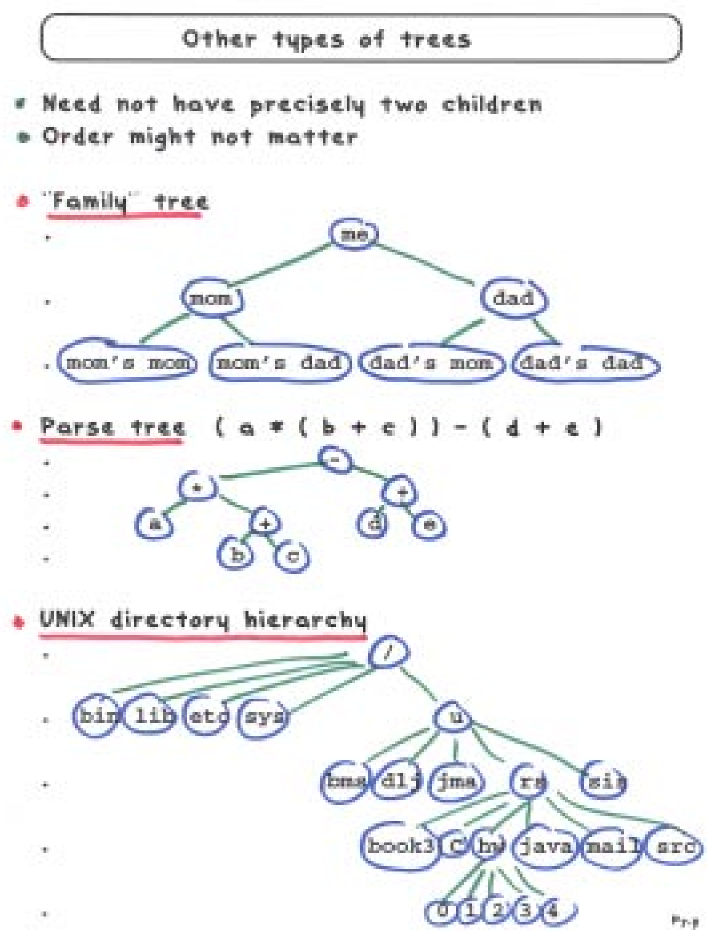
51 14 72 06 33 53 97 13 25 43 64 84 99

Queue contents:

14	72	06	33	53	97	13	25	43	64	84	99
72	06	33	53	97	13	25	43	64	84	99	
	33	53	97	13	25	43	64	84	99		
		97	13	25	43	64	84	99			
				43	64	84	99				
						99					

Outline

- ~~Searching and insertion *without* trees~~
- ~~Searching and insertion *with* trees~~
- ~~Traversing trees~~
- **<u>Conclusion</u>**



What We Have Learned

- How to search and insert into:
  - sorted arrays
  - linked lists
  - binary search trees
- How long these operations take for the different data structures
- The meaning of different traversal orders and how the code for them works