Lecture 14
Binary Search Trees

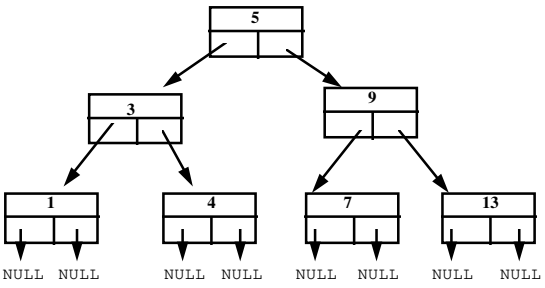
Motivating Observation:
If you start with a number n and halve it at each step, then you can go only for $log_2 n$ steps.

($log_2 n$ = k such that $2^k <= n < 2^{k+1}$)

For $n = 10^8$, $log_2 n$ = 25

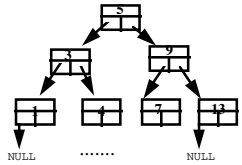
Trees

```
struct tree
  {
   int value;
   struct tree *left;
   struct tree *right;
  };
```

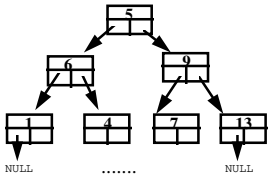


Binary Search Trees

Defining Property: "Every number in your left subtree is smaller than your number ; every number in your right subtree is larger than your number."



A search tree containing the numbers 1, 3, 4, 5, 7, 9, 13

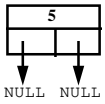


Not a valid search tree

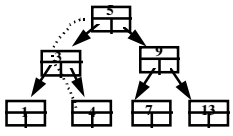
Operations on Search trees

(search() , insert())

A tree with one element.



Search(*val*):
- If value at root == *val,* we're done.
- If value at root > *val,* search the left subtree.
- If value at root < *val,* search the right subtree



Search(4)

Arguments: A pointer to a tree and an integer.
Returned Value: If the integer is present in the tree, a pointer to the node that contains it. If the integer is not present, a pointer to the node where the search ended.

```
/* assumes that *t is not NULL */
struct tree *searchtree(struct tree *t,
                            int val)
{
 if ((*t).value > val)&&((*t).left != NULL))
     return searchtree((*t).left, val);
 if ((*t).value < val)&&((*t).right != NULL))
     return searchtree((*t).right, val);
 return t;
}
```


Insert()

Search the tree for *val* . If *val* is not found, then create a new tree node and attach it where the search ended.

```
/* assumes t != NULL */
int insert(struct tree *t, int val)
{
 struct tree *x, *y;
 x = searchtree(t, val);
 if ((*x).value == val)
     return -1; /* val already present */
 y = malloc(sizeof(struct tree));
 (*y).value = val;
  (*y).left = (*y).right = NULL;
 if ((*x).value < val)
     (*x).right = y;
 else
     (*x).left = y;
 return 1; /*succeeded! */
}
```

What tree does this program produce?

```
void main()
{
 struct tree *t;

 t= malloc(sizeof(struct tree));
 (*t).value = 5;
 (*t).right = (*t).left =NULL;

 insert(t, 5);
 insert(t, 3);
 insert(t, 4)
 insert(t, 9);
 insert(t, 13);
 insert(t, 7);
 insert(t, 1);
}
```

Note: shape of the tree depends upon the order in which numbers are inserted

What does the following function do?

```
/*assumes t!= NULL */
void printtree(struct tree *t)
{
 if ((*t).left != NULL)
    printtree((*t).left);
 printf("%d\n", (*t).value);
 if ((*t).right != NULL)
    printtree((*t).right);
}
```

And this one?

```
/*assumes t!= NULL */
void printtree2(struct tree *t)
{
 if ((*t).right != NULL)
    printtree2((*t).right);
 printf("%d\n", (*t).value);
 if ((*t).left != NULL)
    printtree2((*t).left);
}
```

Efficiency of search() on a tree

To search for a value in a tree with n numbers, we look at the root, and if the value is not there, then we search in one of the subtrees

If tree is "balanced" , then both left and right subtrees have size approximately $n/2$

We have halved a range of size n to a range of size $n/2$ in one step!

(Fact: there exists a way to ensure that trees are always balanced (done in COS226))

(Versions using the -> notation)

```
/* assumes that *t is not NULL */
struct tree *searchtree(struct tree *t,
                            int val)
{
 if (t->value > val)&&(t->left != NULL))
     return searchtree(t->left, val);
 if ((t-> value < val)&&(t->right != NULL))
     return searchtree(t->right, val);
 return t;
}
```

```
/* assumes t != NULL */
int insert(struct tree *t, int val)
{
 struct tree *x, *y;
 x = searchtree(t, val);
 if (x->value == val)
     return -1; /* val already present */
 y = malloc(sizeof(struct tree));
 y->value = val;
 y ->left = y->right = NULL;
 if (x->value < val)
     x->right = y;
 else
     x->left = y;
 return 1; /*succeeded! */
}
```

(Versions using the -> notation)

```
/*assumes t!= NULL */
void printtree(struct tree *t)
{
 if (t->left != NULL)
    printtree(t->left);
 printf("%d\n", t->value);
 if (t->right != NULL)
    printtree(t->right);
}
```

```
/*assumes t!= NULL */
void printtree2(struct tree *t)
{
 if (t->right != NULL)
    printtree2(t->right);
 printf("%d\n", t->value);
 if (t->left != NULL)
    printtree2(t->left);
}
```


Corrected version of bubblesort

Bubble Sort:

do the following for i=0 to n-1:

"bubble" the smallest value among A[i], A[i+1], ... A[n-1] into A[i].

```
void swap(int *x, int *y)
{int t;
 t = *x; *x =*y; *y=t;
}
void bubblesort(int A[], int n)
{ int i, j;
  for(i=0; i<n; i++)
   for(j=n-1;j >i; j--)
     if (A[j-1] > A[j])
        swap(&A[j], &A[j-1]);
}
```

Running Time = $O(n^2)$