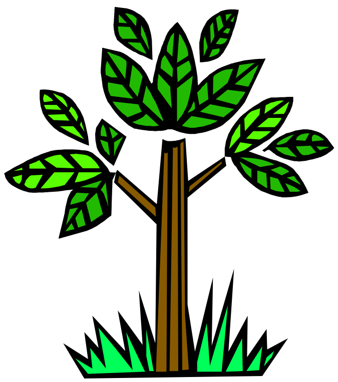
Lecture P10: Trees



Overview

Culmination of the programming portion of this class.
- Solve a database search problem.

Tree data structure.
- Useful.
- Versatile.
- Naturally recursive.

Searching a Database

Database entries.
- Names and social security numbers.

Desired operations.
- Insert student.
- Delete student.
- Search for name given ID number.

Goal.
- All operations fast, even for huge databases.

Data structure that supports these operations is called a SYMBOL TABLE.

SS #	Last
192042006	Arac
201211991	Baron
177999898	Bergbreiter
232871212	Buchen
122993434	Durrett
162882273	Gratzer

"search key"

Other Symbol Table Applications

Other applications.
- Online phone book looks up names and telephone numbers.
- Spell checker looks up words in dictionary.
- Internet domain server looks up IP addresses.
- Compiler looks up variable names to find type and memory address.

Web Site	IP Address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.amazon.com	208.216.181.15
www.pregnantchad.com	209.052.165.60

Representing the Database Entries

Define `Item.h` file to encapsulate generic database entry.

- Insert and search code should work for any item type.
  - ideally `Item` would be an ADT
- Key is field in search.

ITEM.h

```
typedef int Key;
typedef struct {
  Key ID;
  char name[30];
} Item;

Item NULLitem = {-1, ""};

int eq(Key, Key);
int less(Key, Key);
Key key(Item);
void show(Item);
```

item.c

```
#include "ITEM.h"

int eq(Key k1, Key k2) {
  return k1 == k2;
}
int less(Key k1, Key k2) {
  return k1 < k2;
}
Key key(Item x) {
  return x.ID;
}
void show(Item x) {
  printf("%d %s\n", x.ID, x.name);
}
```


Symbol Table ADT

Define `ST.h` file to specify database operations.

- Make it a true symbol table ADT.

ST.h (Sedgewick 12.1)

```
Item STsearch(Key);    /* search for Key in database    */
void STinsert(Item);   /* insert new Item into database */
void STshow(void);     /* print all Items in database   */
int  STcount(void);    /* number items in database      */
void STdelete(Item);   /* delete Item from database     */
```


Unsorted Array Representation of Database

Maintain array of Items.

- Use SEQUENTIAL SEARCH to find database `Item`.

STunsortedarray.c

```
#define MAXSIZE 10000
static Item st[MAXSIZE];
static int size = 0;

Item STinsert(Item item) {
   st[size] = item;
   size++;
}

Item STsearch(Key k) {
   int i;
   for (i = 0; i < size; i++)
      if eq(k, key(st[i]))
         return st[i];
   return NULLitem;
}
```

\# elements → `static int size = 0;`

Array of database Items. → `static Item st[MAXSIZE];`

Key k found. → `return st[i];`

Key k not found. → `return NULLitem;`

Unsorted Array Representation of Database

Maintain array of Items.

- Use SEQUENTIAL SEARCH to find database `Item`.

Advantage.

Key drawback.

Extra problem.

Sorted Array Representation of Database

Maintain array of Items.

- Store in sorted order (by Key).
- Use BINARY SEARCH to find database `Item`.

Array of database Items. →
Key k not found. →
Key k found. →
Divide-and-conquer. →

STsortedarray.c (Sedgewick 12.6)

```
#define MAXSIZE 10000
static Item st[MAXSIZE];
static int size = 0;

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


Sorted Array Representation of Database

Maintain array of Items.

- Store in sorted order (by Key).
- Use BINARY SEARCH to find database `Item`.

"Wrapper" for search function. →

STsortedarray.c (Sedgewick 12.6)

```
Item STsearch(Key k) {
   return search(0, size-1, k);
}
```


Cost of Binary Search

How many "comparisons" to find a name in database of size N?

- Divide list in half each time.
  - 5000 ⇒ 2500 ⇒ 1250 ⇒ 625 ⇒ 312 ⇒ 156 ⇒ 78 ⇒ 39 ⇒ 18 ⇒ 9 ⇒ 4 ⇒ 2 ⇒ 1
- $\lceil \log_2 (N+1) \rceil$ = number of digits in binary representation of N.
- $5000_{10} = 1001110001000_2$

The log functions grows very slowly.

- $\log_2$ (thousand) ≈ 10
- $\log_2$ (million) ≈ 20
- $\log_2$ (billion) ≈ 30

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

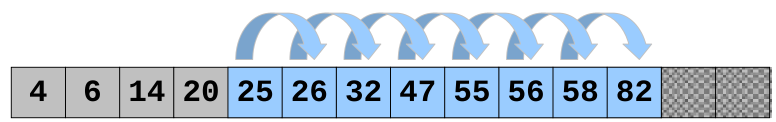
Without binary search (or if unsorted): may examine all N items.

- N vs. $\log_2$ N savings is staggering for large files.
- Milliseconds vs. hours (or more!).

Insert Using Sorted Array Representation

Key Problem: insertion is slow.

- Want to keep entries in sorted order.
- Have to move larger keys over one position to right.



Demo: inserting 25 into a sorted array.

Sorted Array Representation of Database

Maintain array of Items.

- Store in sorted order (by Key).
- Use BINARY SEARCH to find database `Item`.

Advantage.

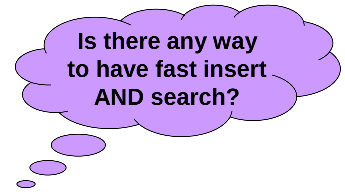
Key drawback.

Extra problem.

Summary

Database entries.

- Names and social security numbers.



Desired operations.

- Insert, delete, search.

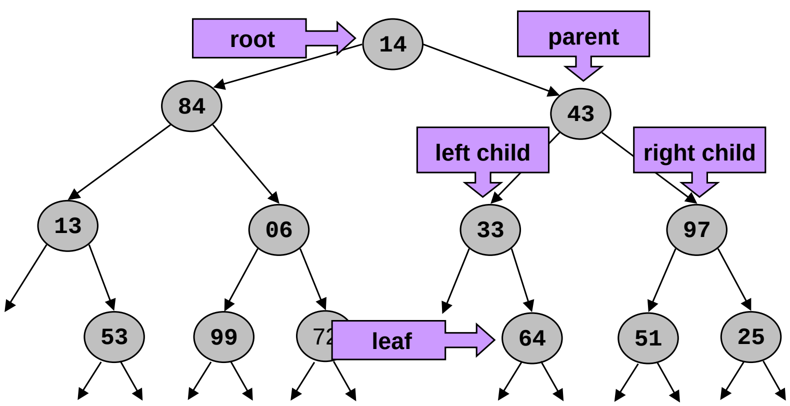
Goal.

- Make all of these operations FAST even for huge databases.

	asymptotic time			computer time		
	search	insert	delete	search	insert	delete
sorted array	log N	N	N	instant	2 hour	2 hour
unsorted array	N	1	1	2 hour	instant	instant
goal	log N	log N	log N	instant	instant	instant

Binary Tree

Yes. Use TWO links per node!



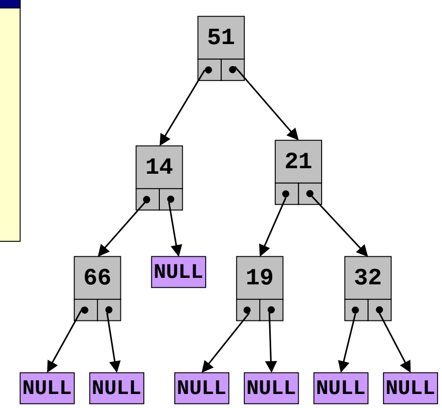
Binary Tree in C

STbst.h

```
typedef struct STnode* link;
struct STnode {
  Item item;
  link left;
  link right;
};

static link root;
```

item	
left	right



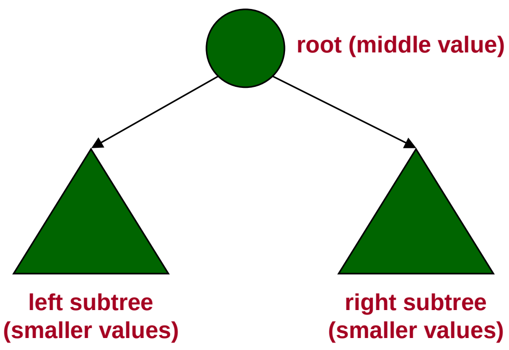
Represent in C with TWO links per node.

- Leftmost arrow corresponds to left link.
- Rightmost to right link.

Binary Search Tree

Binary tree in "sorted" order.

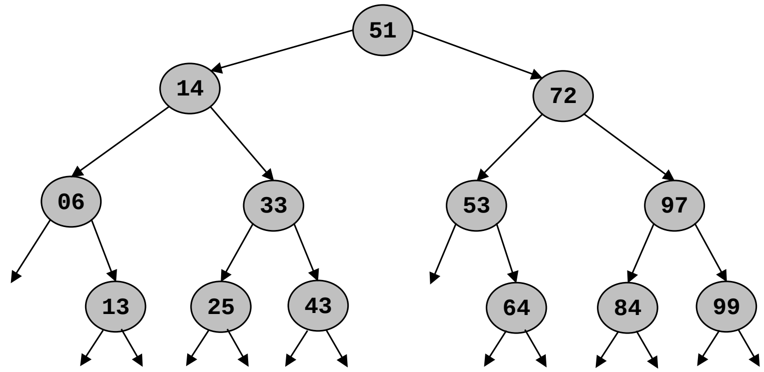
- **Maintain ordering property for ALL sub-trees.**



Binary Search Tree

Binary tree in "sorted" order.

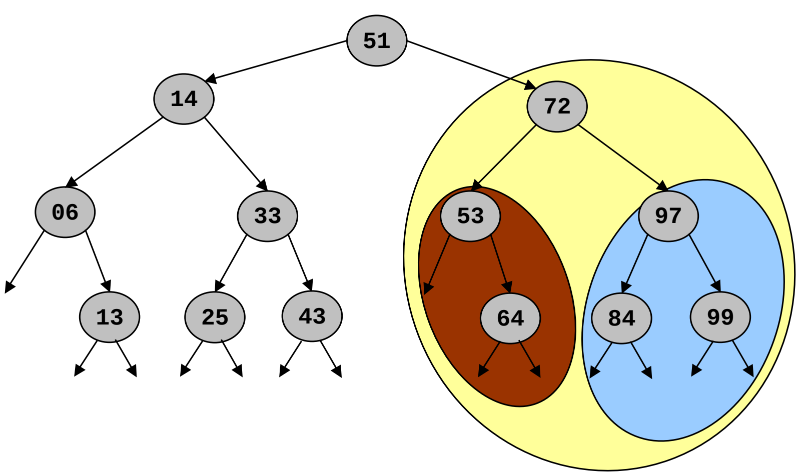
- **Maintain ordering property for ALL sub-trees.**



Binary Search Tree

Binary tree in "sorted" order.

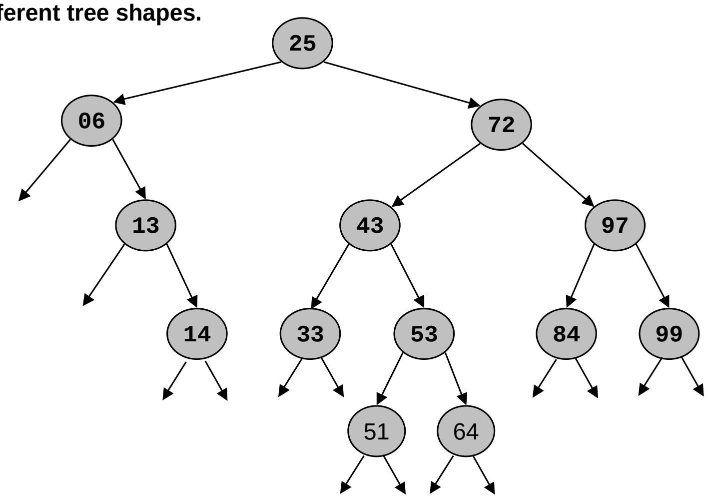
- **Maintain ordering property for ALL sub-trees.**



Binary Search Tree

Binary tree in "sorted" order.

- **Many BST's for the same input data.**
- **Have different tree shapes.**



Search in Binary Search Tree

Search for Key k in binary search tree.

- Analogous to binary search in sorted array.

Search algorithm:

- Start at head node.
- If Key of current node is k, return node.
- Go LEFT if current node has Key < k.
- Go RIGHT if current node has Key > k.

Search in BST's

Search for Key k.

STbst.c (Sedgewick 12.7)

```
Item search (link x, Key k) {
  if (x == NULL)
    return NULLitem;
  else if (eq(k, key(x->item))
    return x->item;
  else if (less(k, key(x->item))
    return search(x->left, k);
  else
    return search(x->right, k);
}

Item STsearch(Key k) {
  return search(root, k);
}
```

Key k not in tree.

Found Key k.

Look for Key k in left subtree.

Look for Key k in right subtree.

Search for Key k in BST tree rooted at head.

Cost of BST Search

Depends on tree shape.

- Proportional to length of path from root to Key.
- "Balanced."
  - $2 \log_2 N$ comparisons
  - proportional to binary search cost
- "Unbalanced."
  - takes N comparisons for degenerate tree shapes
  - can be as slow as sequential search

Algorithm works for any tree shape.

- With cleverness (e.g., "red-black trees" in COS 226), can ensure tree is always (roughly) balanced.

Insert Using BST's

How to insert new database Item.

- Search for key of database Item.
- Search ends at NULL pointer.
- New Item "belongs" here.
- Allocate memory for new Item, and link it to tree.

Insert Using BST's

BST.c (Sedgewick 12.7)

```
link insert(link x, Item item) {
  if (x == NULL)
     return NEWnode(item, NULL, NULL);

  else if (less(key(item), key(x->item))
     x->left = insert(x->left, item);
  else
     x->right = insert(x->right, item);
  return x;
}

void STinsert(Item item) {
  head = insert(root, item);
}
```

Insert new node here.

Divide-and-conquer.

Wrapper function.

Insert Using BST's

BST.c (Sedgewick 12.7)

```
link NEWnode(Item item, link left, link right) {
   link x = malloc(sizeof *x);
   if(x == NULL) {
      printf("Error allocating memory.\n");
      exit(EXIT_FAILURE);
   }
   x->item  = item;
   x->left  = left;
   x->right = right;
   return x;
}
```

Allocate memory and initialize.

Insertion Cost in BST

Depends on tree shape.

- Cost is proportional to length of path from root to node.

Tree shape depends on order keys are inserted.

- Insert in "random" order.
  - leads to "well-balanced" tree
  - average length of path from root to node is $1.44 \log_2 N$
- Insert in sorted or reverse-sorted order.
  - degenerates into linked list
  - takes N -1 comparisons

Algorithm works for any tree shape.

- With cleverness (e.g., red-black trees in COS 226), can ensure tree is always balanced.

Summary

Database entries.

- Names and social security numbers.

Desired operations.

- Insert, delete, search.

Goal.

- Make all of these operations FAST even for huge databases.

	asymptotic time			computer time		
	search	insert	delete	search	insert	delete
sorted array	log N	N	N	instant	2 hour	2 hour
unsorted array	N	1	N	2 hour	instant	2 hour
BST	log N	log N	log N	instant	instant	instant

Question

Current code searches for a name given an ID number.

What if we want to search for an ID number given a name?

ITEM.h

```
typedef char Key[30];
typedef struct {
  int ID;
  Key name;
} Item;

Item NULLitem = {-1, ""};

int eq(Key, Key);
int less(Key, Key);
Key key(Item);
```

item.c

```
#include <string.h>
int eq(Key k1, Key k2) {
  return strcmp(k1, k2) == 0;
}

int less(Key k1, Key k2) {
  return strcmp(k1, k2) < 0;
}

Key key(Item item) {
  return item.name;
}
```

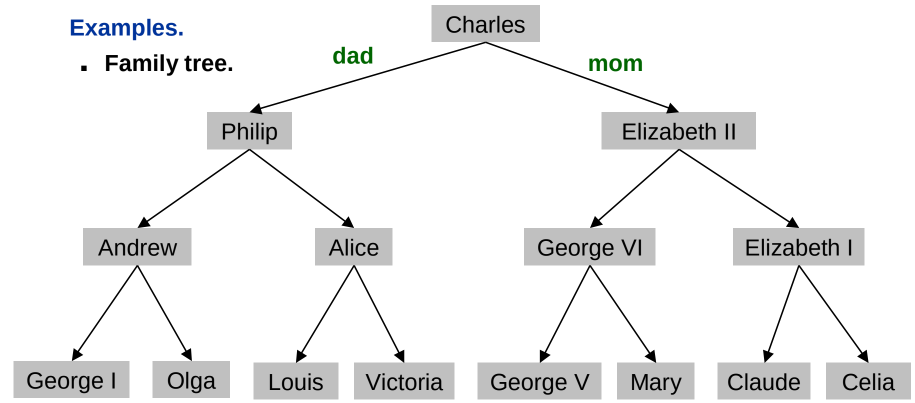

Other Types of Trees

Trees.

- Nodes need not have exactly two children.
- Order of children may not be important.

Examples.

- Family tree.



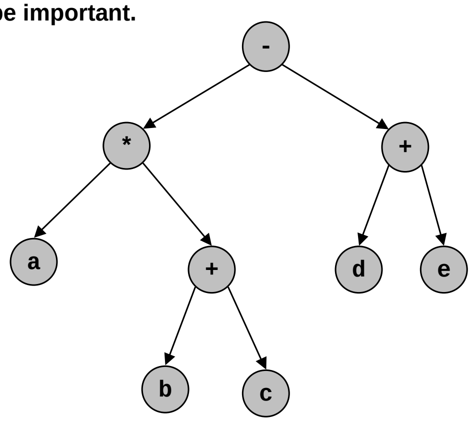
Other Types of Trees

Trees.

- Nodes need not have exactly two children.
- Order of children may not be important.

Examples.

- Family tree.
- Parse tree.
  (a * (b + c)) - (d + e)



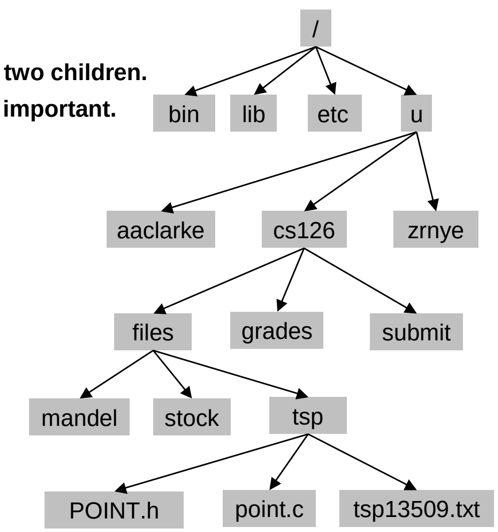
Other Types of Trees

Trees.

- Nodes need not have exactly two children.
- Order of children may not be important.

Examples.

- Family tree.
- Parse tree.
- Unix file hierarchy.
  - not binary



Traversing Binary Trees

Goal: visit (process) each node in tree in some order.

- "Tree traversal."

wrapper function →

STbst.c

```
void STprint(void) {
  traverse(root);
}
```

inorder

```
void traverseInorder(link x) {
  if (x == NULL)
     return;
  traverse(x->left);
  show(x->item);
  traverse(x->right);
}
```

traverse left subtree → `traverse(x->left);`

traverse right subtree → `traverse(x->right);`

process node h → `show(x->item);`

Traversing Binary Trees

Goal: visit (process) each node in tree in some order.

- "Tree traversal."
- Goal realized no matter what order nodes are visited.
  - inorder: visit between recursive calls

inorder

```
void traverseInorder(link x) {
  if (x == NULL)
     return;
  traverse(x->left);
  show(x->item);
  traverse(x->right);
}
```


Traversing Binary Trees

Goal: visit (process) each node in tree in some order.

- "Tree traversal."
- Goal realized no matter what order nodes are visited.
  - inorder: visit between recursive calls
  - preorder: visit before recursive calls

preorder

```
void traversePreorder(link x) {
  if (x == NULL)
     return;
  show(x->item);
  traverse(x->left);
  traverse(x->right);
}
```


Traversing Binary Trees

Goal: visit (process) each node in tree in some order.

- "Tree traversal."
- Goal realized no matter what order nodes are visited.
  - inorder: visit between recursive calls
  - preorder: visit before recursive calls
  - postorder: visit after recursive calls

postorder

```
void traversePostorder(link x) {
  if (x == NULL)
     return;
  traverse(x->left);
  traverse(x->right);
  show(x->item);
}
```

Traversing Binary Trees

Goal: visit (process) each node in tree in some order.

- "Tree traversal."
- Goal realized no matter what order nodes are visited.
  - inorder: visit between recursive calls
  - preorder: visit before recursive calls
  - postorder: visit after recursive calls

Preorder Traversal With Explicit Stack

Visit the top node on the stack.

- Push its children onto stack.

Push right node before left, so that left node is visited first.

preorder traversal with stack

```
void traverse(link x) {
   STACKpush(x);
   while (!STACKempty()) {
      h = STACKpop();
      show(x->item);
      if (x->right != NULL)
         STACKpush(x->right);
      if (x->left != NULL)
         STACKpush(x->left);
   }
}
```


Level Traversal With Queue

Q. What happens if we replace stack with QUEUE?

- Level order traversal.
- Visit nodes in order from distance to root.

level traversal with queue

```
void traverse(link x) {
   QUEUEput(x);
   while (!QUEUEisempty()) {
      x = QUEUEget();
      show(x->item);
      if (x->left != NULL)
         QUEUEput(x->left);
      if(x->right != NULL)
         QUEUEput(x->right);
   }
}
```


Summary

How to insert and search a database using:

- Unsorted array.
- Sorted array.
- Binary search tree.

Performance characteristics using different data structures.

Preorder, inorder, postorder, levelorder tree traversals.