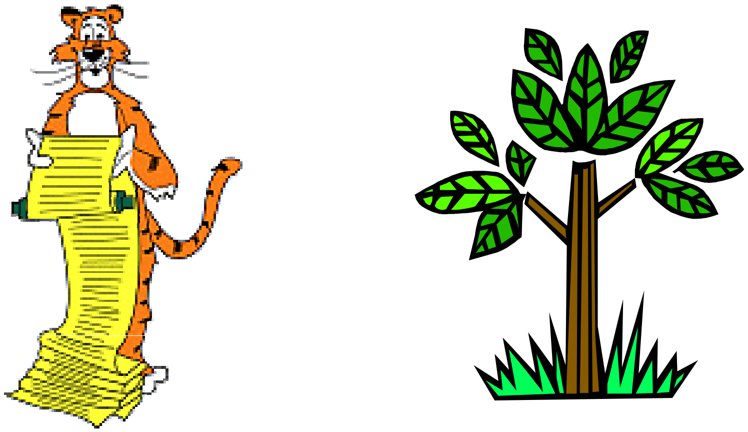
Lecture P9: Trees



Overview

Culmination of the programming portion of this class.

- Solve a database searching problem.

Trees

- Versatile and useful data structure.
- A naturally recursive data structure.
- Application of stacks and queues.

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 Name e
1920342006	Alam
2012121991	Baer
2021230087	Bagyenda
1779999898	Balestri
2328761212	Benjamin
1229993434	Berube

search key

Searching a Database

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.

Representing the Database Entries

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

- Don't want to use internals of item type when we write database code.
  - want our insert and search code to 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 print(Item);
```

Item.c

```
#include “ITEM.h”
int eq(Key A, Key B) {
  return A == B;
}
int less(Key A, Key B) {
  return A < B;
}
Key key(Item A) {
  return A.ID;
}
void print(Item A) {
  printf(“%9d %20s”,
  A.ID, A.name);
}
```

Symbol Table ADT

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

- Make it a true ADT.

ST.h (Sedgewick 12.1)

```
Item STsearch(Key);
void STinsert(Item);
void STprint(void);
int  STcount(void);
void STdelete(Item);
```

Sorted Array Representation of Database

Maintain array of Items.

- Store in sorted order.
- Use BINARY SEARCH to find database `Item` with designated `Key`.

STarray.c (Sedgewick 12.6)

```
#define MAXSIZE 10000
Item st[MAXSIZE];

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

Array of database Items.

Key k not found.

Key k found.

Divide and conquer.

Sorted Array Representation of Database

Maintain array of Items.

- Store in sorted order.
- Use BINARY SEARCH to find database `Item` with designated `Key`.

“Wrapper” for search function.

STarray.c (Sedgewick 12.6)

```
Item STsearch(Key k) {
  int N = Stcount();
  return search(0, N-1, k);
}
```

Cost of Binary Search

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

- $\lceil \log_2 N \rceil$ = number of digits in binary representation of N.
- Divide list in half each time.

5000 ⇒ 2500 ⇒ 1250 ⇒ 625 ⇒ 312 ⇒ 156 ⇒ 78 ⇒ 39
⇒ 18 ⇒ 9 ⇒ 4 ⇒ 2 ⇒ 1

The log functions grows very slowly.

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

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

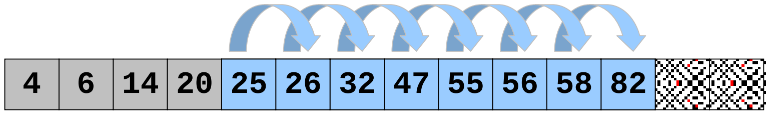
Without binary search (or if unsorted), may need to look at EVERY Item.

- Savings is enormous for large files.

Insert Using Sorted Array Representation

Problem 1: insertion is slow.

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

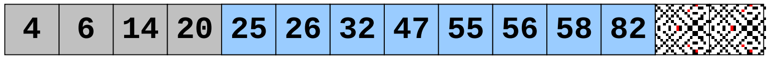


Demo: inserting 25 into a sorted array.

Insert Using Sorted Array Representation

Problem 1: insertion is slow.

- Want to keep entries in sorted order.
- Have to move larger keys over one position to right.
- Exercise: write code for insertion.



Demo: inserting 25 into a sorted array.

Problem 2: need to fix maximum database size ahead of time.

Linked List Representation of Database

Keep items in a linked list.

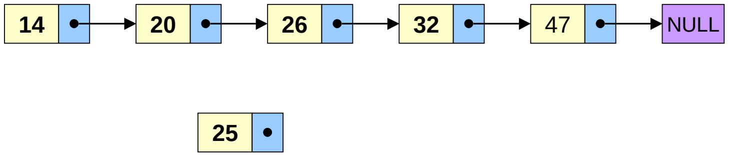
- Store in sorted order.

STlist.c

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

Insert.

- Only need to change links.
- No need to “move” large amounts of data.



Linked List Representation of Database

Keep items in a linked list.

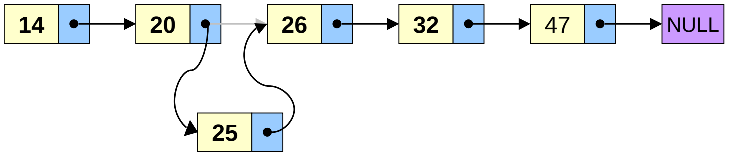
- Store in sorted order.

Insert.

- Only need to change links.
- No need to “move” large amounts of data.

STlist.c

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



Linked List Representation of Database

Search.

- Can’t use binary search since no DIRECT access to middle element.
- Use sequential search.
  - may need to search entire linked list to find desired Key
  - much slower than binary search

STlist.c

```
Item STsearch(Key k) {
  link x;
  for (x = head; x != NULL; x = x->next)
    if (eq(k, key(x)) return x->item;
  return NULLitem;
}
```

Summary

Database entries.

- Names and social security numbers.

Desired operations.

- Insert, delete, search.

Goal.

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

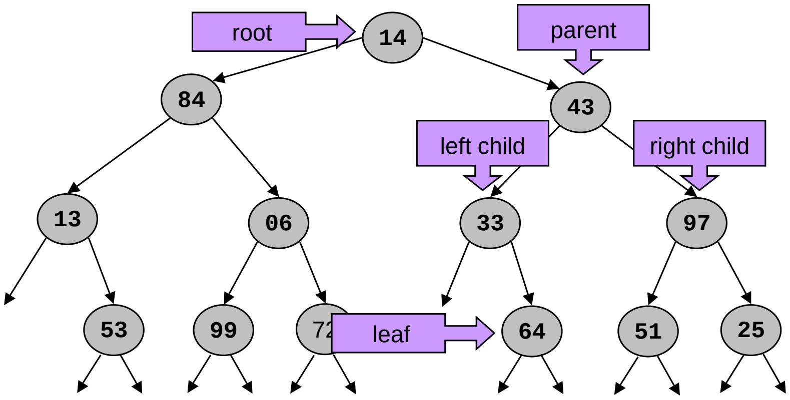
Tradeoff.

- ARRAY: fast search, slow insert/delete.
- LINKED LIST: fast insert/delete, slow search.

Is there any way to have fast insert AND search?

Binary Tree

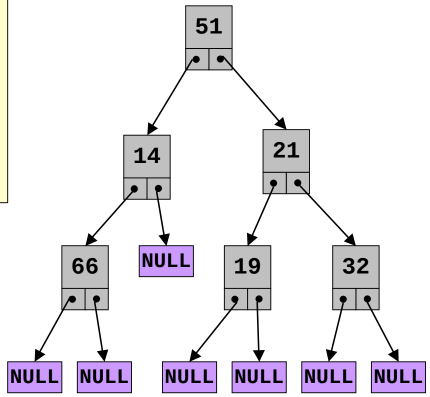
Yes. Use TWO links per node.



Binary Tree in C

STbst.h

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



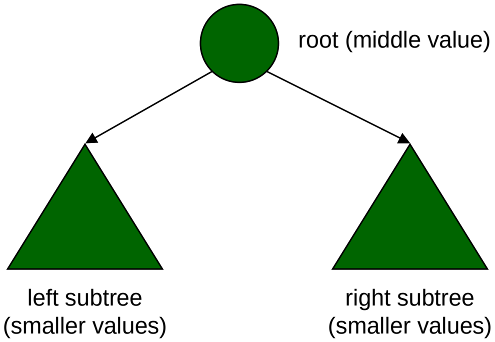
Represent in C with two links per node.

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

Binary Search Tree

Binary tree in “sorted” order.

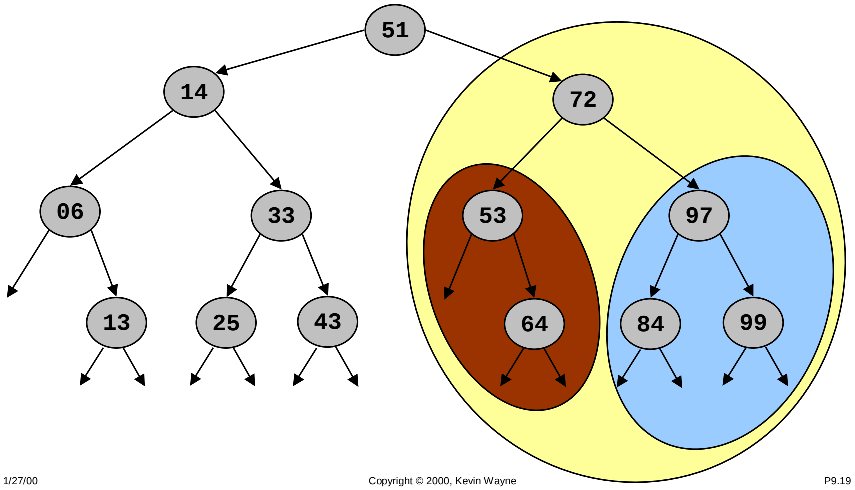
- Maintain ordering property for ALL subtrees.



Binary Search Tree

Binary tree in “sorted” order.

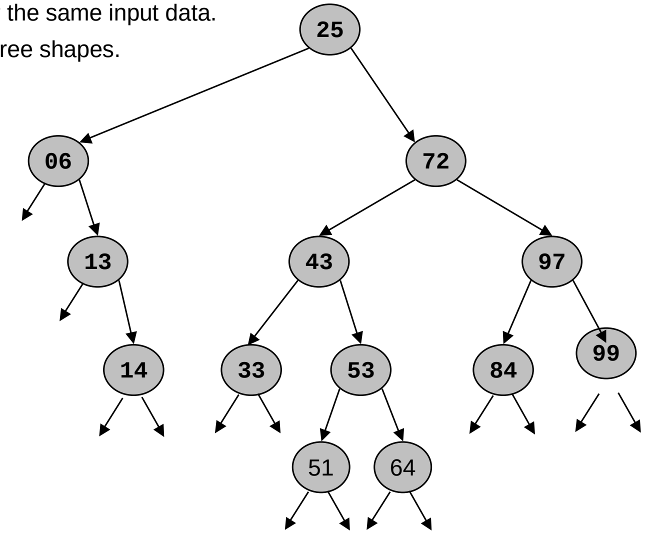
- Maintain ordering property for ALL subtrees.



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 h, Key k) {
  if (h == NULL)
    return NULLitem;
  if (eq(k, key(h->item))
    return h->item;
  if (less(k, key(h->item))
    return search(h->left, k);
  return search(h->right, k);
  }

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

Key k not in tree.

Found Key k.

Look for Key k in left subtree.

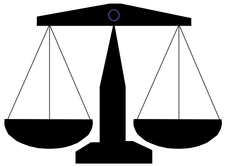
Look for Key k in right subtree.

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 (see COS 226), can assure tree is always 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 h, Item item) {
  Key k  = key(item);
  Key k2 = key(h->item);
  link x;
  if (h == NULL) {
    link x = malloc(sizeof *x);
    x->item = item;
    x->left = x->right = NULL;
    return x;
  }
  if (less(k, k2))
    h->left = insert(h->left, item);
  else h->right = insert(h->right, item);
  return h;
}

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

Allocate memory and insert here.

Divide-and-conquer.

Wrapper function.

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
- With cleverness, can ensure tree is always balanced.
  - see COS 226

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 A, Key B) {
  return strcmp(A, B) == 0;
}

int less(Key A, Key B) {
  return strcmp(A, B) < 0;
}

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

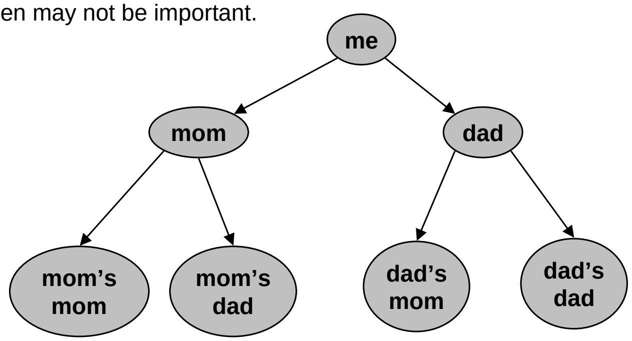
Other Types of Trees

Trees.

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

Examples.

- Family tree.



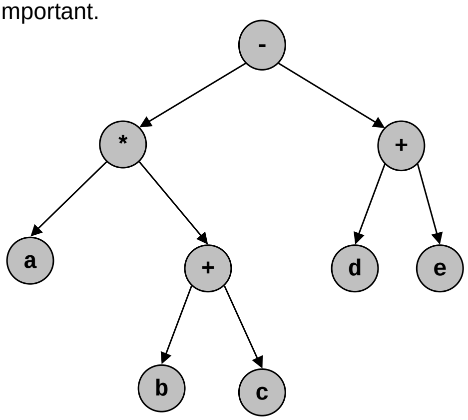
Other Types of Trees

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)



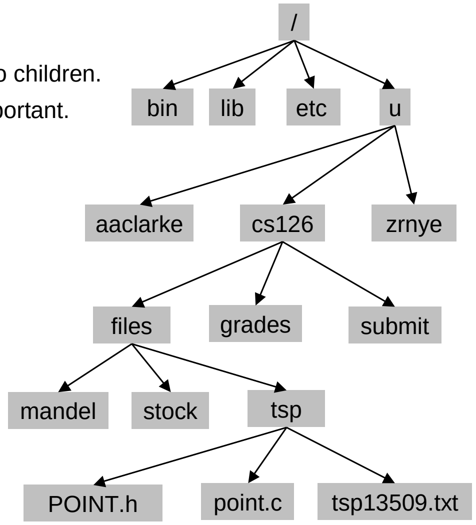
Other Types of Trees

Trees.

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

Examples.

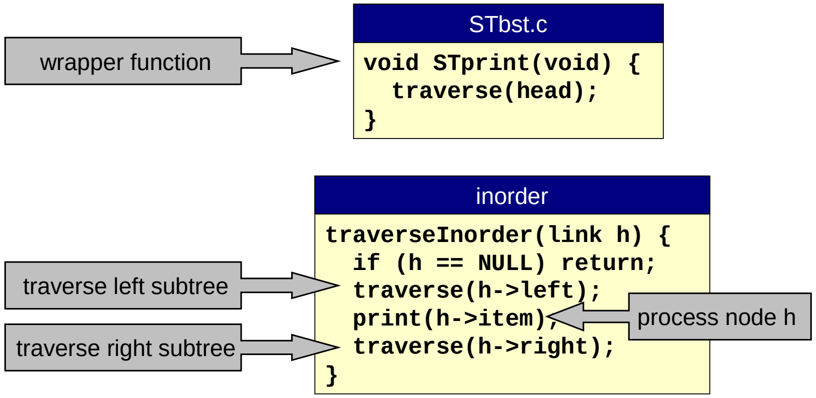
- Family tree.
- Parse tree.
- Unix file hierarchy.
  - not binary



Traversing Binary Trees

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

- “Tree traversal.”



STbst.c

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

inorder

```
traverseInorder(link h) {
  if (h == NULL) return;
  traverse(h->left);
  print(h->item);
  traverse(h->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

inorder

```
traverseInorder(link h) {
  if (h == NULL) return;
  traverse(h->left);
  print(h->item);
  traverse(h->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

```
traversePreorder(link h) {
  if (h == NULL) return;
  print(h->item);
  traverse(h->left);
  traverse(h->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

```
traversePostorder(link h) {
  if (h == NULL) return;
  traverse(h->left);
  traverse(h->right);
  print(h->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.

preorder traversal with stack

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

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

Works for general trees. Generalizes to DEPTH-FIRST-SEARCH in graphs.

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

```
traverse(link h) {
  QUEUEput(h);
  while (!QUEUEempty()) {
    h = QUEUEget();
    print(h->item);
    if (h->left != NULL)
      QUEUEput(h->left);
    if(h->right != NULL)
      QUEUEput(h->right);
  }
}
```

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

Summary

How to insert and search a database using:

- Arrays.
- Linked lists.
- Binary search trees.

Performance characteristics using different data structures.

The meaning of different traversal orders and how the code for them works.