

CSE1303 Part A
Data Structures and Algorithms
Summer Semester 2003

Lecture A13 – Binary Search Trees
(Information Retrieval)

Kymerly Fergusson

Overview

- Binary Search Trees.
- Hash Tables.



2

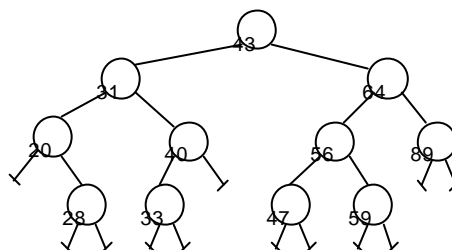
Recall - Binary Search Tree

A Binary Tree such that:

- Every node entry has a **unique** key.
- **All** the keys in the **left subtree** of a node are **less** than the key of the node.
- **All** the keys in the **right subtree** of a node are **greater** than the key of the node.

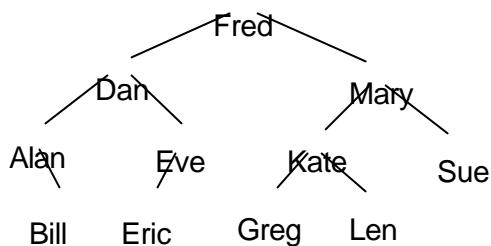
3

Example 1: *key is an integer*



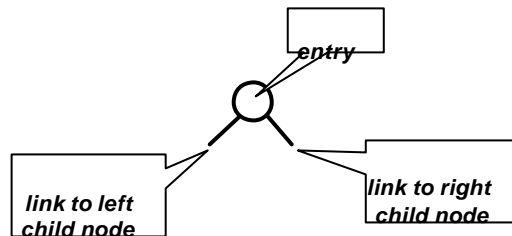
4

Example 2: *key is a string*



5

Binary Tree Node



6

Binary Search Tree Node

Example 1:

```
struct TreeNodeRec
{
    int    key;

    struct TreeNodeRec* leftPtr;
    struct TreeNodeRec* rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

7

Binary Search Tree Node

Example 2:

```
#define MAXLEN 15

struct TreeNodeRec
{
    char    key[MAXLEN];

    struct TreeNodeRec* leftPtr;
    struct TreeNodeRec* rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

maximum
string length
is fixed

8



Recall:

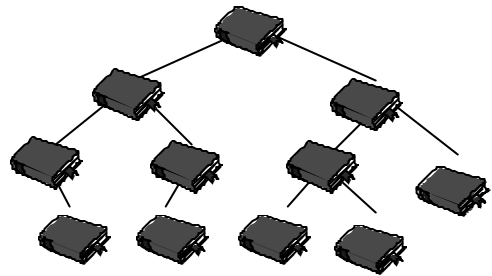
```
struct TreeNodeRec
{
    char* key;

    struct TreeNodeRec* leftPtr;
    struct TreeNodeRec* rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

- Allows strings of arbitrary length.
- Memory needs to be allocated dynamically before use.
- Use strcmp to compare strings.

9



10

Example 3: Binary Search Tree Node

```
struct BookRec
{
    char* author;
    char* title;
    char* publisher;

    /* etc.: other book information. */
};

typedef struct BookRec Book;

struct TreeNodeRec
{
    Book info;
    struct TreeNodeRec* leftPtr;
    struct TreeNodeRec* rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

key

11

Tree Node

```
struct TreeNodeRec
{
    float key;

    struct TreeNodeRec* leftPtr;
    struct TreeNodeRec* rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

12

```
#ifndef TREEH
#define TREEH

struct TreeNodeRec
{
    float          key;
    struct TreeNodeRec* leftPtr;
    struct TreeNodeRec* rightPtr;
};

typedef struct TreeNodeRec TreeNode;

TreeNode* makeTreeNode(float value);
TreeNode* insert(TreeNode* nodePtr, float item);
TreeNode* search(TreeNode* nodePtr, float item);
void printInorder(const TreeNode* nodePtr);
void printPreorder(const TreeNode* nodePtr);
void printPostorder(const TreeNode* nodePtr);

#endif
```

13

MakeNode

- parameter: item to be inserted
- steps:
 - allocate memory for the new node
 - check if memory allocation is successful
 - if so, put item into the new node
 - set left and right branches to NULL
- returns: pointer to (i.e. address of) new node

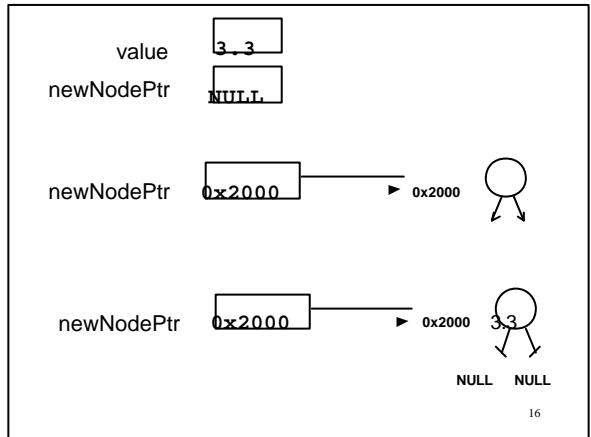
14

```
TreeNode* makeTreeNode(float value)
{
    TreeNode* newNodePtr = NULL;

    newNodePtr = (TreeNode*)malloc(sizeof(TreeNode));

    if (newNodePtr == NULL)
    {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    else
    {
        newNodePtr->key = value;
        newNodePtr->leftPtr = NULL;
        newNodePtr->rightPtr = NULL;
    }
    return newNodePtr;
}
```

15



16

Inorder

- Inorder traversal of a Binary Search Tree **always** gives the sorted order of the keys.

```
void printInorder(TreeNode* nodePtr)
{
    // traverse left sub-tree
    // visit the node
    // traverse right sub-tree
}
```

initially, pointer to root node

17

Inorder

- Inorder traversal of a Binary Search Tree **always** gives the sorted order of the keys.

```
void printInorder(TreeNode* nodePtr)
{
    if (nodePtr != NULL)
    {
        printInorder(nodePtr->leftPtr);
        printf("%f", nodePtr->key);
        printInorder(nodePtr->rightPtr);
    }
}
```

18

Inorder

```

void printInorder(TreeNode* nodePtr){
    if (nodePtr != NULL){
        printInorder(nodePtr->leftPtr);
        printf("%f", nodePtr->key);
        printInorder(nodePtr->rightPtr);
    }
}
    
```

19

Search

Example: 59

20

Search

Example: 61

21

Search: Checklist

- if target key is less than current node's key, search the left sub-tree.
- else, if target key is greater than current node's key, search the right sub-tree.
- returns:
 - if found, or if target key is equal to current node's key, a pointer to node containing target key.
 - otherwise, NULL pointer.

(Recall binary search)

22

```

TreeNode* search(TreeNode* nodePtr, float target)
{
    if (nodePtr != NULL)
    {
        if (target < nodePtr->key)
        {
            nodePtr = search(nodePtr->leftPtr, target);
        }
        else if (target > nodePtr->key)
        {
            nodePtr = search(nodePtr->rightPtr, target);
        }
    }
    return nodePtr;
}
    
```

23

Function Call to Search

```

/* ...other bits of code omitted... */

printf("Enter target ");
scanf("%f", &item);

if (search(rootPtr, item) == NULL)
{
    printf("Item was not found\n");
}
else
{
    printf("Item found\n");
}

/* ...and so on... */
    
```

24

Search Find 0.7

```

TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
    }
    return nodePtr;
}
    
```

25

Insert

Example:

26

Insert

- Create new node for the item.
- Find a parent node.
- Attach new node as a leaf.

27

Insert: Recursive

- parameters:
 - pointer to current node (initially: root node).
 - item to be inserted.
- If current node is NULL
 - Create a new node and return it.
- Else if item's key is less (greater) than current node's key:
 - otherwise, let the left (right) child node be the current node, setting the parent left (right) link equal that node, and repeat recursively.

28

```

TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
    {
        nodePtr = makeTreeNode(item);
    }
    else if (item < nodePtr->key)
    {
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    }
    else if (item > nodePtr->key)
    {
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    }
    return nodePtr;
}
    
```

29

Function Call to Insert

```

/* ...other bits of code omitted... */
printf("Enter number of items ");
scanf("%d", &n);

for (i = 0; i < n; i++) {
    scanf("%f", &item);
    rootPtr = insert(rootPtr, item);
}

/* ...and so on... */
    
```

30

Insert

Insert 0.9

```

TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
    return nodePtr;
}
    
```

31

Sorting

To sort a sequence of items:

- Insert items into a Binary Search Tree.
- Then Inorder Traverse the tree.

32

Sort

Sort the following list into a binary search tree

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

33

Sorting: Analysis

- Average Case: $O(n \log(n))$

- Insert $(i+1)^{th}$ item: $\sim \log_2(i)$ comparisons

34

Sorting: Analysis

- Worst Case: $O(n^2)$

Example:
Insert: 1, 3, 7, 9, 11, 15

- Insert $(i+1)^{th}$ item: $\sim i$ comparisons

35

Revision

- Binary Search Tree
- Make Tree Node, Insert item, Search for an item, and Print Inorder.
- Tree sort.

Revision: Reading

- Kruse 9
- Standish 9
- Langsam 5
- Deitel & Deitel 12.7

Preparation

Next lecture: Hash Tables

- Read Chapter 8.6 in Kruse et al.

36