Data Structures

Searching

Searching Efficiencies

• Best case: \( O(1) \)
• Worst Case: \( O(n) \)
• We can easily achieve: \( O(\log n) \)
Linear Search

```c
int linearsearch(int x[], int n, int key) {
    int i;
    for (i = 0; i < n; i++)
        if (x[i] == key)
            return(i);
    return(-1);
}
```

Improved Linear Search

```c
int linearsearch(int x[], int n, int key) {
    int i;
    //This assumes an ordered array
    for (i = 0; i < n && x[i] <= key; i++)
        if (x[i] == key)
            return(i);
    return(-1);
}
```
Binary Search

int binarysearch(int x[], int n, int key)
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (x[mid] == key)
            return(mid);
    }
    return(-1);
}

Binary Search (continued)

if (x[mid] > key)
    high = mid - 1;
else
    low = mid + 1;

return(-1);
A Binary Search Tree

Deleting A Node On A Search Tree - Before
Deleting A Node On A Search Tree - After

Is Balancing a Tree Important?
Is Balancing a Tree Important?

An AVL Tree

- An AVL Tree is a search tree where the depth of the left and right subtree can differ by no more than one level.
An AVL Tree

Inserting A Node On An AVL Tree
Inserting A Node On An AVL Tree

After Rotating
AVL.h

```cpp
#include <iostream>

using namespace std;

typedef int infotype;

struct nodetype {
    infotype info;
    int balance;
    struct nodetype *left, *right;
};

typedef struct nodetype *nodeptr;

class AVLTree {
public:
    AVLTree(void);
    AVLTree(infotype key);
    nodeptr maketree(infotype key);
    void traverse(void);
    nodeptr insert(infotype key);
private:
    nodeptr tree;
    void rightRotation(nodeptr p);
    void leftRotation(nodeptr p);
    void intrav(nodeptr tree);
};
```
AVL.cpp

#include "AVL.h"

// AVLTree() - Default constructor creating a null tree
AVLTree::AVLTree(void) {
    tree = NULL;
}

// AVLTree() - Initializing constructor creating a one-node tree
AVLTree::AVLTree(infotype key) {
    tree = maketree(key);
}

// traverse() - Does an in-order traversal using the recursive method
void AVLTree::traverse(void) {
    intrav(tree);
}

// maketree() - Creates a node for the AVL tree
nodeptr AVLTree::maketree(infotype key) {
    nodeptr p;

    p = new struct nodetype;
    p -> info = key;
    p ->balance = 0;
    p ->left = p ->right = NULL;
    return p;
}
/** rightRotation() - Rotates an AVL subtree to the right **/ void AVLTree::rightRotation(nodeptr p) {
    nodeptr q, hold;
    q = p -> left;
    hold = q -> right;
    q -> right = p;
    p -> left = hold;
}

pageTitle: // leftRotation() - Rotates an AVL subtree to the left
pageTitle: void AVLTree::leftRotation(nodeptr p) {
    nodeptr q, hold;
    q = p -> right;
    hold = q -> left;
    q -> left = p;
    p -> right = hold;
}

pageTitle: // intrav() - A recursive method to traverse the tree in order
pageTitle: void AVLTree::intrav(nodeptr tree) {
    static level = 0;
    level++;
    if (tree != NULL) {
        intrav(tree -> left);
        cout << level << ' ' << tree -> info << endl;
        intrav(tree-> right);
    }
    --level;
}
// insert() - Inserts a record into its proper place on an AVL tree
nodeptr AVLTree::insert(infotype key) {
    nodeptr fp, p, q, ya, fya, s;
    int imbalance;
    // Part I - Search and insert into the binary tree
    fp = NULL;
    p = tree;
    fya = NULL;
    ya = p;

    // ya points to the youngest ancestor which may become unbalanced. fya points to ya's father and fp points to the father of p
    while (p != NULL) {
        if (key == p -> info)
            return p;
        q = (key < p -> info)?
            p -> left: p -> right;
        if (q != NULL)
            if (q -> balance != 0) {
                fya = p;
                ya = q;
            }
        fp = p;
        p = q;
    } // while p!= NULL

    // Insert new record
    q = maketree(key);
    q -> balance = 0;
    (key < fp -> info)?
        fp -> left = q: fp -> right = q;
// The balance on all nodes between
// ya and q must be changed from 0
p = (key < ya -> info)?
    ya -> left: ya -> right;

s = p;
while (p != q) {
    if (key < p -> info) {
        p -> balance = 1;
        p = p -> left;
    }
    else {
        p -> balance = -1;
        p = p -> right;
    } // if key < p -> info
} // while p != q

// Part II - Ascertain whether or not the tree
// is unbalanced.
// If it is, q is the newly inserted node, ya
// is its youngest unbalanced ancestor, fya is
// ya's father and s is ya's son in the
// direction of the imbalance.
imbalance = (key < ya -> info)? 1 : -1;
if (ya -> balance == 0) {
    // Another level has been added to the tree
    // The tree remains balanced
    ya -> balance = imbalance;
    return q;
} // if (ya -> balance == 0)
if (ya -> balance != imbalance) {
    // The added node has been placed in
    // the opposite direction of the imbalance
    // The tree remains balanced
    ya -> balance = 0;
    return q;
}
// Part III - the additional node has
// unbalanced the tree
// Rebalance it by performing the required
// rotations and then adjust the balances of
// the nodes involved.
if (s -> balance == imbalance) {
    // ya nd s have been unbalanced in the same
    // direction
    p = s;
    if (imbalance == 1)
        rightRotation(ya);
    else
        leftRotation(ya);
    ya -> balance = 0;
    s -> balance = 0;
}

else {
    // ya and s are unbalanced in opposite
    // directions
    if (imbalance == 1) {
        p = s -> right;
        leftRotation(s);
        ya -> left = p;
        rightRotation(ya);
    }
    else {
        p = s -> left;
        ya -> right = p;
        rightRotation(s);
        leftRotation(p);
    }
}
// Adjust balance field for involved nodes
if (p ->balance == 0) {
    // p was inserted node
    ya -> balance = 0;
    s -> balance = 0;
}
else
    if (p -> balance == imbalance) {
        ya -> balance = -imbalance;
        s -> balance = 0;
    }
    else {
        ya -> balance = 0;
        s -> balance = imbalance;
    }
    p -> balance = 0;
}

// Adjust pointer to the rotated subtree
if (fya == NULL)
    tree = p;
else (ya == fya -> right)? fya -> right = p:
    fya -> left = p;
return q;
Inserting an AVL Tree: An Example

```c
int main(void) {
    AVLTree myTree(5);
    nodeptr z;
    int i;

    for (i = 6; i < 250; i++) {
        z = myTree.insert(2*i);
    }
    myTree.traverse();
    return(0);
}
```

General Search Trees

![Diagram with node 22 and two children](image-url)
B Trees

B Trees (continued)
B Trees (continued)

```
52  
  4 22  60 99
52  
  4 22  60 83 99
```

B Trees (continued)

```
52  
  4 22  60 83 99
52  
  4 22  60 81 83 99
```
B Trees (continued)

```
  52
 /  \\  \\
4 13 22 60 81 83 99
```

B Trees (continued)

```
  52
 /  \\  \\
4 13 22 51 60 81 83 99
```
B Trees (continued)

52

2 4 13 22 51

60 81 83 99

13 52

2 4

22 51

60 81 83 99

B Trees (continued)

13 52

2 4 7 10

22 51

60 81 83 99
B Trees (continued)

13, 52

2, 4, 7, 10, 17, 22, 25, 51, 60, 70, 81, 83, 99

13, 52, 81

2, 4, 7, 10, 17, 22, 25, 51, 60, 70, 83, 99

B Trees (continued)

13, 52, 81

2, 4, 7, 10, 17, 22, 25, 49, 51, 60, 70, 83, 99

13, 25, 52, 81

2, 4, 7, 10, 17, 22, 49, 51, 60, 70, 83, 99
B Trees (continued)
Hashing

- Hashing is at best an O(1) searching process, searching a large array by compressing the key into a smaller value called a hash value.
- Hash values will not be unique which leads to *hash collision* or *hash clash*. This is resolved by one of two means, one of which is called *rehashing*. 

Hash Table
Resolving Hash Collision

- Resolving hash collision is done by a process called **rehashing**.
- The simplest strategy for rehashing is to simply add one to the hash value and store the record in the next available slot. If this is occupied keeping incrementing until an available slot is found. This is called **linear probing**.

---

**Hash.h**

```cpp
#include <iostream>  // Necessary for input
                   // and output
#include <string.h> // Necessary for
                   // manipulating C-style
                   // strings
using namespace std;

// The necessary constants
const int hashTableSize = 10000;
const int keySize = 20;
const int recordTableSize = 500;

// A prototypical record type to be stored
typedef struct {
    char key[keySize];
} recordStuff;
```
// A class for a table to be searched via hashing
class HashTable {
public:
    HashTable(void); // The initialization
        // constructor
    int hashValue(char *skey); // Finds the hash value
    bool insertRecord(recordStuff myRecord); // Inserts a new record in the table
    bool findRecord(recordStuff &foundRecord, char *skey); // Searches for a record by key

private:
    // The predecessor value of the index
    inline int pred(int index)
    {return (index == 0)?
        hashTableSize - 1: index - 1;}

    // The successor value of the index
    inline int succ(int index)
    {return (index == hashTableSize - 1)?
        0: index + 1;}

    int recordTableLength; // Number of entries in the table so far
    int hashTable[hashTableSize];
    recordStuff recordTable[recordTableSize];
};
Hash.cpp

#include "Hash.h"

// HashTable() - An initialization constructor
HashTable::HashTable(void) {
    int i;

    // The whole table should be -1 because
    // there are no entries yet
    for (i = 0; i < hashTableSize; i++)
        hashTable[i] = -1;

    // The table is initially empty
    recordTableLength = 0;
}

// hashValue() - Returns the hash value
int HashTable::hashValue(char *skey) {
    int i;
    long sum = 0;

    // Add the ASCII code of the character
    // and left shift one place
    for (i = 0; i < strlen(skey) && i < 8*sizeof(int)-8; i++)
        sum = (sum << 1) + (int)(skey[i]);
    return sum % hashTableSize;
}
bool HashTable::insertRecord(recordStuff myRecord) {
    int index, firstIndex;

    // Make sure there's room for the new record
    if (recordTableLength == recordTableSize - 1)
        return(false);

    // Place the record in the next available slot
    recordTable[recordTableLength] = myRecord;

    // This is the expected hash value
    index = firstIndex = hashValue(myRecord.key);

    // If there is a hash collision, use linear
    // probing to find a spot for it in the hash
    // table
    while (hashTable[index] != -1
           && index != pred(firstIndex))
        index = succ(index);

    // We searched the whole hash table
    // and could not find room for it
    if (hashTable[index] != -1)
        return(false);

    // Don't forget to increment the length;
    // the table is one record larger
    hashTable[index] = recordTableLength++;
    return(true);
}
// FindRecord() - Look for the record, return it
// as a reference parameter and
// return whether or not the
// record was there
bool HashTable::FindRecord
    (recordStuff &foundRecord, char *skey) { int index, firstIndex, recordIndex;

    // Find the proper hash value
    index = firstIndex = hashValue(skey);

    // Get the corresponding index in the record's table
    recordIndex = hashTable[index];

    // This may not be it; we may have had hash
    // collision that was resolved by linear
    // probing
    while
        (strcmp(recordTable[recordIndex].key,
                skey) != 0 && index != pred(firstIndex)) {
            index = succ(index);
            recordIndex = hashTable[index];
        }

    // If the record was here, return an empty
    // record and a result of false (ie, the
    // record wasn't found
    if (strcmp(recordTable[recordIndex].key,
                skey) != 0) {
        strcpy(foundRecord.key, "" );
        return false;
    }
// Return the record and a result of true
// (ie the record was found)
foundRecord = recordTable[recordIndex];
return true;
}

A Driver To Test Hash.cpp

int main(void) {
    HashTable h;
    recordStuff rs;
    bool found;

    strcpy(rs.key, "act");
    h.insertRecord(rs);
    found = h.findRecord(rs, "act");
    if (found)
        cout << strlen(rs.key) << "\"
            << rs.key << \"\" << endl;
    else
        cout << \"\" << "act"
            << \"\" was not found." << endl;
    return(0);
}
Separate Chaining