CSC 344 – Algorithms and Complexity

Lecture #5 – Searching

Why Search?

- Everyday life -We are always looking for something – in the yellow pages, universities, hairdressers
- Computers can search for us
- World wide web provides different searching mechanisms such as yahoo.com, bing.com, google.com
- Spreadsheet – list of names – searching mechanism to find a name
- Databases – use to search for a record
- Searching thousands of records takes time the large number of comparisons slows the system

Sequential Search

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- Best case?
- Worst case?
- Average case?

Sequential Search

```
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 Sequential Search

```
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 (A Decrease and Conquer Algorithm)

- Very efficient algorithm for searching in sorted array:
  - K vs A[0] . . . A[m] . . . A[n-1]
- If K = A[m], stop (successful search);

  otherwise, continue searching by the same method in:
  - A[0..m-1] if K < A[m]
  - A[m+1..n-1] if K > A[m]

Binary Search (A Decrease and Conquer Algorithm)

$l \leftarrow 0$; $r \leftarrow n\text{-}1$

while $l \leq r$ do

  $m \leftarrow \lfloor (l+r)/2 \rfloor$

  if $K = A[m]$ return m

  else if $K < A[m]$ $r \leftarrow m\text{-}1$

  else $l \leftarrow m+1$

return -1

Analysis of Binary Search

- Time efficiency
- Worst-case recurrence:
  - $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$, $C_w(1) = 1$
    solution: $C_w(n) = \lceil log_2(n+1) \rceil$
  - This is VERY fast: e.g., $C_w(10^6) = 20$
- Optimal for searching a sorted array
- Limitations: must be a sorted array (not linked list)

binarySearch

```
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);
```

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

      return(-1);
}
```

Searching Problem

Problem: Given a (multi)set *S* of keys and a search key *K*, find an occurrence of *K* in *S*, if any.

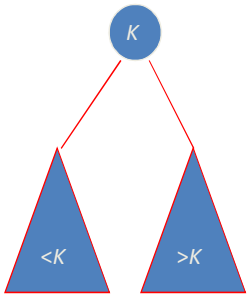
- Searching must be considered in the context of:
  - File size (internal vs. external)
  - Dynamics of data (static vs. dynamic)
- Dictionary operations (dynamic data):
  - Find (search)
  - Insert
  - Delete

Taxonomy of Searching Algorithms

- List searching
  - Sequential search
  - Binary search
  - Interpolation search
- Tree searching
  - Binary search tree
  - Binary balanced trees: AVL trees, red-black trees
  - Multiway balanced trees: 2-3 trees, 2-3-4 trees, B trees
- Hashing
  - Open hashing (separate chaining)
  - Closed hashing (open addressing)

Binary Search Tree

- Arrange keys in a binary tree with the *binary search tree property*:



Example: 5, 3, 1, 10, 12, 7, 9

Dictionary Operations on Binary Search Trees

- **Searching** – straightforward
- **Insertion** – search for key, insert at leaf where search terminated
- **Deletion** – 3 cases:
  - Deleting key at a leaf
  - Deleting key at node with single child
  - Deleting key at node with two children

Dictionary Operations on Binary Search Trees

- Efficiency depends of the tree's height:
  $\lfloor \log_2 n \rfloor \leq h \leq n-1$,
  with height average (random files) be about $3\log_2 n$
- Thus all three operations have
  - worst case efficiency: $\Theta(n)$
  - average case efficiency: $\Theta(\log n)$
- **Bonus**: inorder traversal produces sorted list

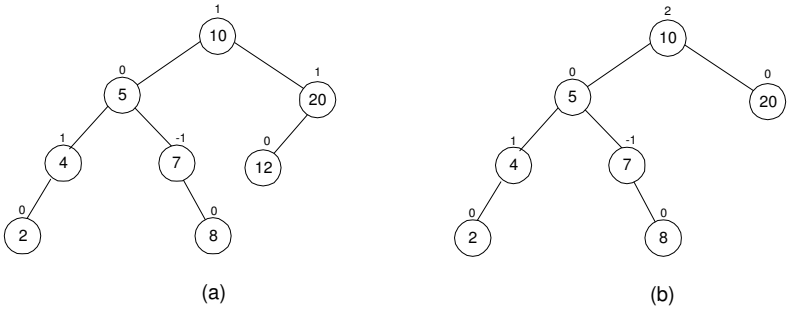
Balanced Search Trees

- Attractiveness of *binary search tree* is marred by the bad (linear) worst-case efficiency. Two ideas to overcome it are:
- to rebalance binary search tree when a new insertion makes the tree "too unbalanced"
  - *AVL trees*
  - *red-black trees*
- to allow more than one key per node of a search tree
  - *2-3 trees*
  - *2-3-4 trees*
  - *B-trees*

Balanced Trees: AVL trees

- Definition An *AVL tree* is a binary search tree in which, for every node, the difference between the heights of its left and right subtrees, called the *balance factor*, is at most 1 (with the height of an empty tree defined as -1)

Balanced Trees: AVL trees

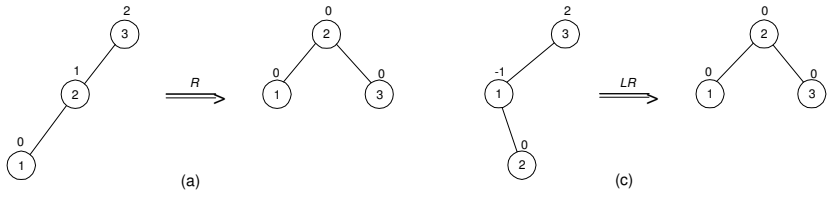


Tree (a) is an AVL tree; tree (b) is not an AVL tree

Rotations

- If a key insertion violates the balance requirement at some node, the subtree rooted at that node is transformed via one of the four *rotations*. (The rotation is always performed for a subtree rooted at an "unbalanced" node closest to the new leaf.)

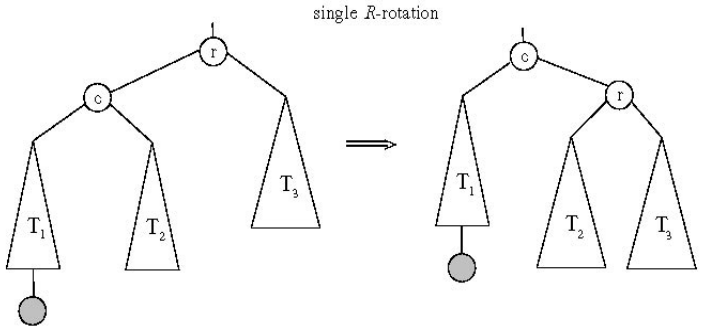
Rotations



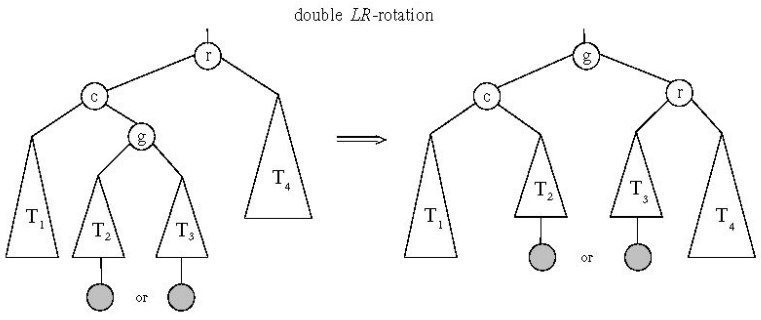
Single *R*-rotation

Double *LR*-rotation

General case: Single R-rotation

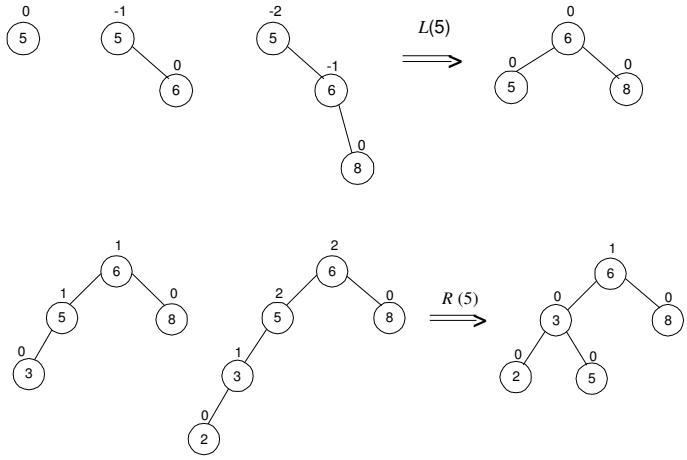


General case: Double LR-rotation

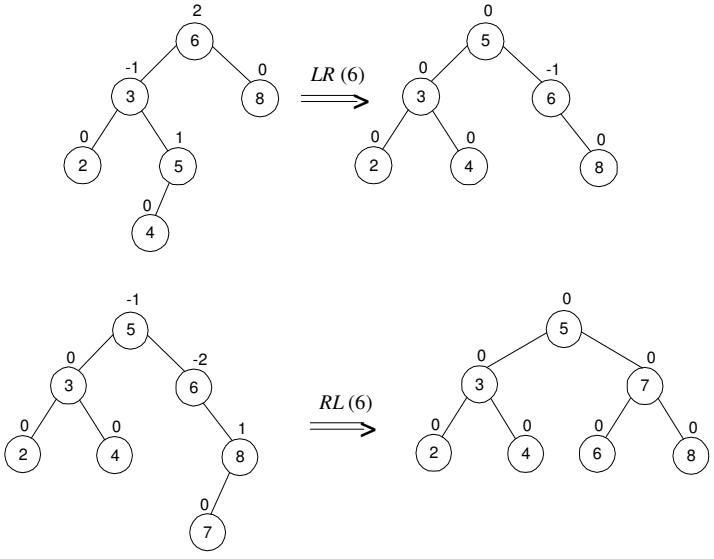


AVL Tree Construction - An Example

- Construct an AVL tree for the list 5, 6, 8, 3, 2, 4, 7



AVL Tree Construction - An Example (continued)



Analysis of AVL trees

- $h \leq 1.4404 \log_2 (n + 2) - 1.3277$
  - Average height: $1.01 \log_2 n + 0.1$ for large n (found empirically)
- Search and insertion are O(log n)
  Deletion is more complicated but is also O(log n)
- Disadvantages:
  - frequent rotations
  - complexity
- A similar idea: *red-black trees* (height of subtrees is allowed to differ by up to a factor of 2)

Analysis of AVL trees

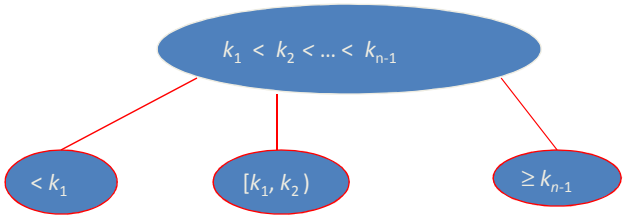
- $h \leq 1.4404 \log_2 (n + 2) - 1.3277$
  - Average height: $1.01 \log_2 n + 0.1$ for large n (found empirically)
  - Search and insertion are O(log n)
  - Deletion is more complicated but is also O(log n)
- Disadvantages:
  - frequent rotations
  - complexity
- A similar idea: *red-black trees* (height of subtrees is allowed to differ by up to a factor of 2)

Multiway Search Trees

Definition A *multiway search tree* is a search tree that allows more than one key in the same node of the tree.

Definition A node of a search tree is called an *n-node* if it contains *n*-1 ordered keys (which divide the entire key range into *n* intervals pointed to by the node's *n* links to its children):

Multiway Search Trees

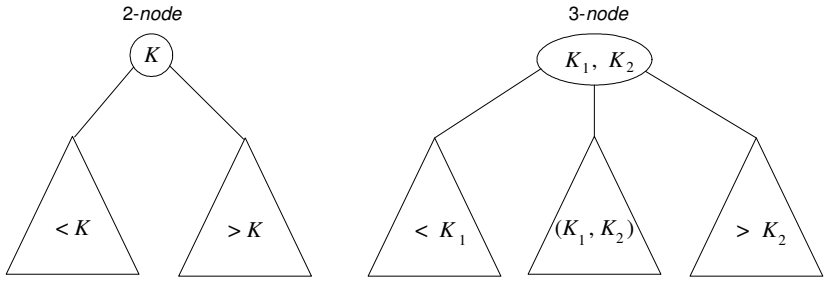


- ***Note***: *Every node in a classical binary search tree is a 2-node*

2-3 Tree

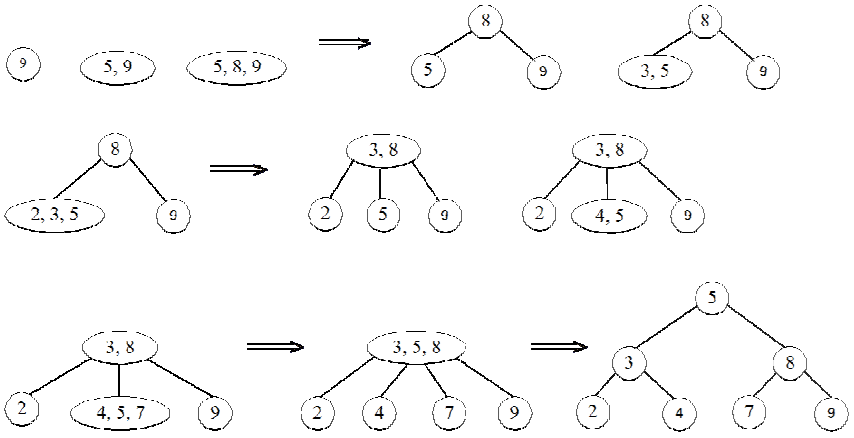
- Definition A 2-3 tree is a search tree that may have 2-nodes and 3-nodes height-balanced (all leaves are on the same level).
- A 2-3 tree is constructed by successive insertions of keys given, with a new key always inserted into a leaf of the tree. If the leaf is a 3-node, it's split into two with the middle key promoted to the parent.

2-3 Tree



2-3 Tree Construction – An Example

- Construct a 2-3 tree the list 9, 5, 8, 3, 2, 4, 7



Analysis Of 2-3 Trees

- $\log_3 (n + 1) - 1 \leq h \leq \log_2 (n + 1) - 1$
- Search, insertion, and deletion are in $\Theta(\log n)$
- The idea of 2-3 tree can be generalized by allowing more keys per node
- 2-3-4 trees
- B-trees

Why Hashing?

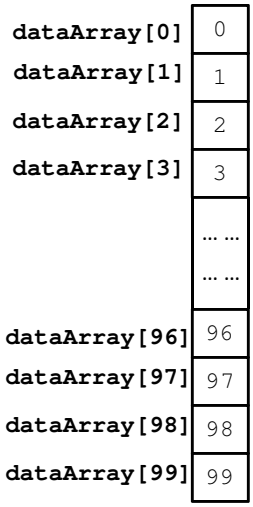
- The best average efficiency that we can do in any search is *log* (*n*). Is there a better way of organizing data.
- If the key is integer and within a small enough range 1 to *n* (e.g., *n* = 100), we can set up an array of *n* data items and store the data there.

Example – Array of 100 items

```
const int arraySize = 100;
typedef  struct   {
      int   key;
      // Other stuff goes here
}     dataType;

dataType    dataArray[arraySize];
```

Example – Array of 100 items (continued)

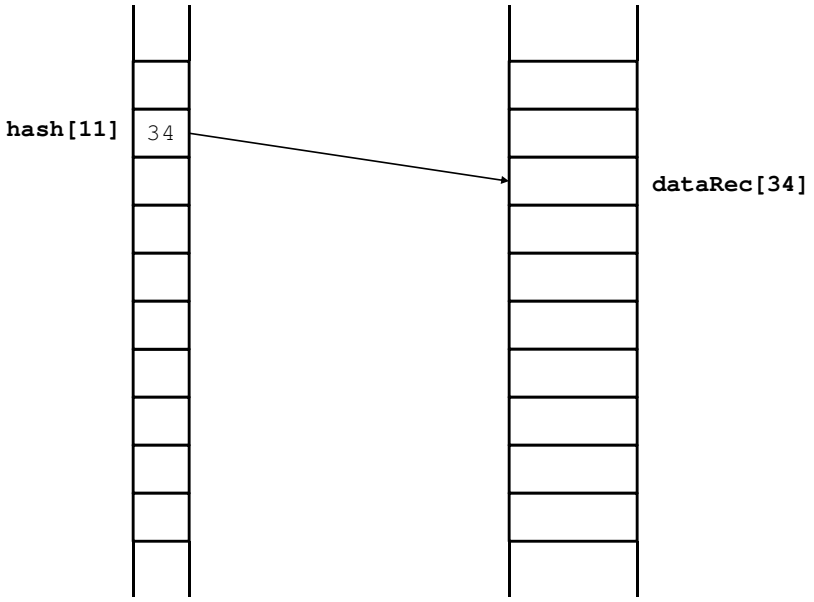


Why Hashing? (continued)

- This approach won't work if the key field covers too large a range of integers or is a character string.
  - E.g., a social security number, a name
- What do we do then?
  - We use a number that is generated out of the primary key. We call this the ***hash value*** and the function producing it the ***hash function***.

Hash Function

- A hash function looks to convert a value from large number or a character string into an integer value that can be a valid array index.
- If a social security number is used as a key, we seek to cut it down to a smaller number; there is no reason to have a billion entries in the hash table.



easyHash.h

```
const int stringLength = 20;
const int arraySize = 100;

typedef char    *keyString;

typedef struct  {
        keyString       *key;
        // Other Stuff can go here;
} dataRecord;

typedef dataRecord      *hashArray;

int     simpleHash(keyString value);
int     simpleHash2(keyString value);
```

easyHash.cc

```
#include        "easyHash.h"
#include        <iostream>
#include        <cstring>
using namespace std;

int     simpleHash(keyString value) {
        // A modular hashing function without
        // hash resolution

        int     i, sum = 0;
        for (i = 0;  value[i] != '\0'; i++)
                sum += value[i];
        return(sum % (arraySize+1));
}
```

```
float   frac(float x);

int     simpleHash2(keyString value)       {
      // Multiplicative hashing without
      //  hash resolution

      // Any positive real < 1 would do
      const float c = 0.301378663;
      int     i, sum = 0;

      for (i = 0;  value[i] != '\0';  i++)
             sum += value[i];

      return (int) ((arraySize + 1) * frac(c*sum));
}
```

```
float   frac(float x)   {
        float fraction = x - (int) x;
        return (fraction);
}

int     main(void)      {
        hashArray   x = new dataRecord[4];
        keyString   value = new char[stringLength];

        strcpy(value, "The quick brown fox");
        cout << value << endl;
        cout << simpleHash(value) << endl;
        cout << simpleHash2(value) << endl;
        return(1);
}
```

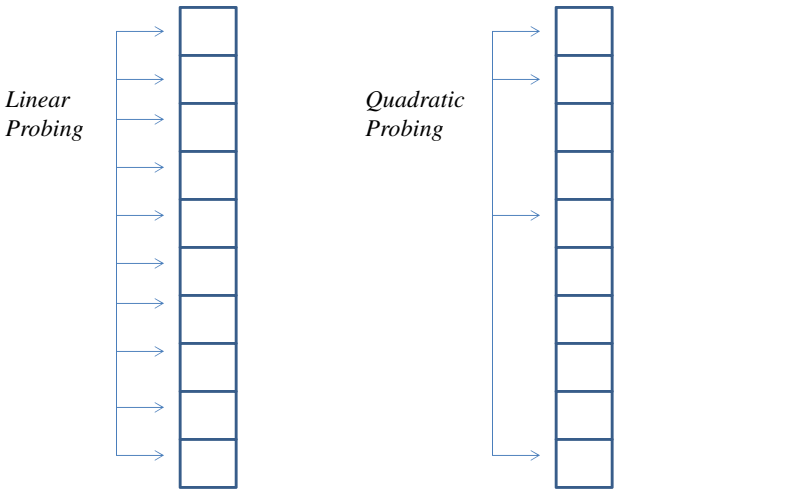
Resolving Hash Collision

- There are two ways that hash collision is usually resolved:
  - ***Rehashing*** – deriving an alternative hash value. It may take several rehashing attempts to avoid hash collision.
  - ***Chaining*** – each hash table entry points to the beginning of a linked list, all of these data entries for which this is the hash value.

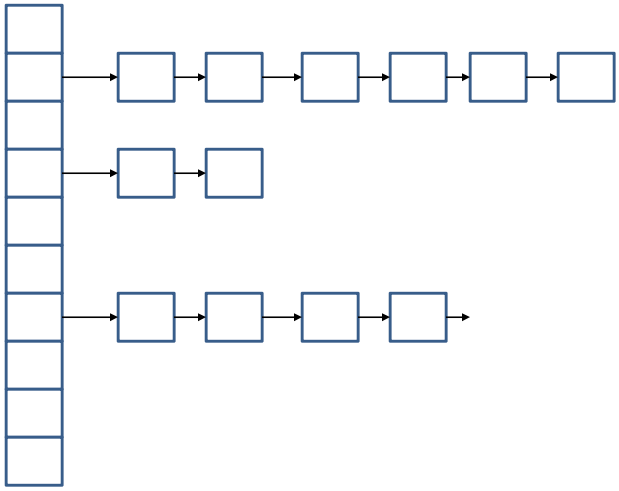
Linear vs. Quadratic Probing

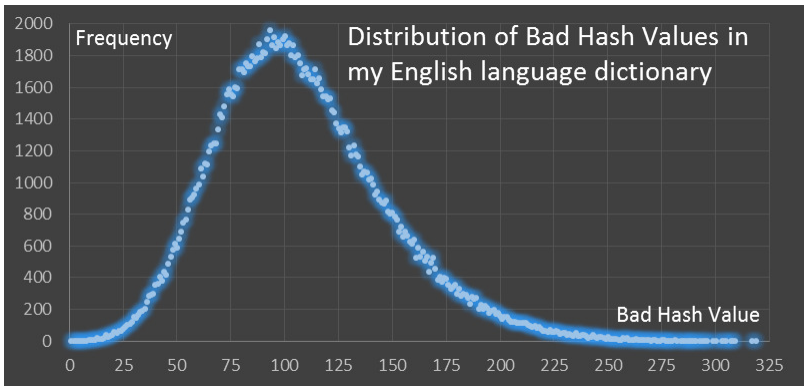
- Linear probing involves going down the entries in the hash table one entry at a time until we find one that is not in use.
- Quadratic Probing involves going down the entries in the hash table but the increment changes from 1, to 4, to 9, etc.

Linear vs. Quadratic Probing

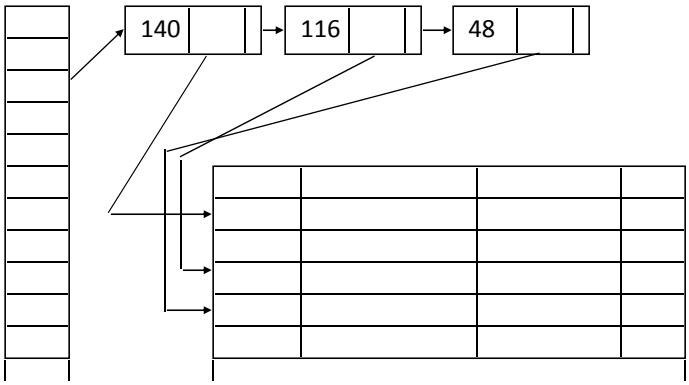


Chaining

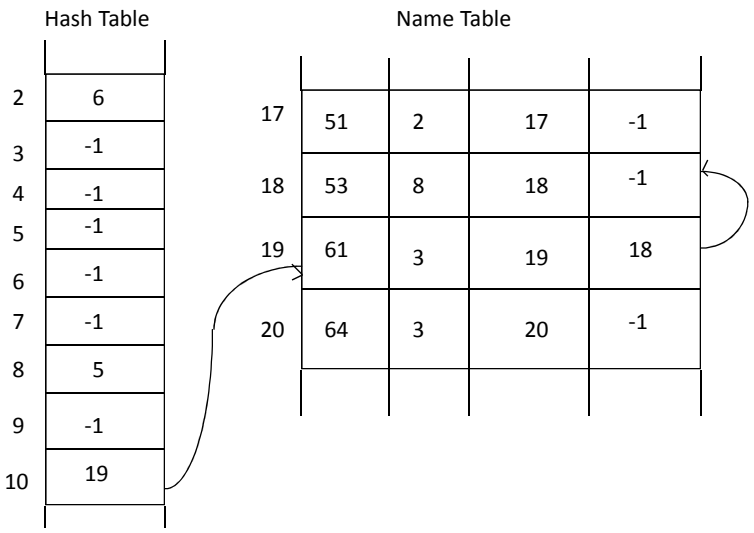




Separate Chaining



Hash Table and the Name Table



HashTable.h

```
#include    <cstring>

#pragma once
const int   dataArraySize = 50;
const int   hashTableSize = 10000;
const int   stringSize = 80;

typedef     struct      {
      int   index;
      char  keyValue[stringSize];
}     hashRecord;
```

```
typedef struct    {
      char  keyField[stringSize];
      // Other fields go here
} dataRecord;
```

```
class HashTable   {
public:
      HashTable(void);
      int   search(char *keyValue);
      void  insert(dataRecord data, dataRecord
dataTable[], bool &dup, bool &full);
      void  remove();
private:
      int   findHashCode(char *k);
      hashRecord  table[hashTableSize];
      int dataArrayLength;
};
```

HashTable.cpp

```
#include "HashTable.h"

// HashTable() - Setting all the indices to -1 and
//               the strings to blank
HashTable::HashTable(void)
{
      int   i;

      for (i = 0;  i < hashTableSize;  i++)     {
            table[i].index = -1;
            table[i].keyValue[0] = '\0';
      }
      dataArrayLength = 0;
}
```

```
// hash() -  calculating the hash function
int     HashTable::findHashCode(char *k)      {
        unsigned i;
        int     sum = 0;

        for (i = 0;  i < strlen(k); i++)
                sum += k[i];

        return (sum % (hashTableSize + 1));
}
```

```
// hashSearch() - Inserting into a Hash Table
//                resolving Hash Collision using
//                Linear Probing
int   HashTable::search(char *k)    {
        int     ix, oldIx;
        bool    found = false;

        ix = findHashCode(k);
        oldIx = ix;
```

```
        do      {
          if (strcmp(table[ix].keyValue, k)== 0)
              found = true;
          else
              ix = (ix +1 ) % hashTableSize;
        } while (!found
                  && table[ix].keyValue[0] != '\0'
                  && ix != oldIx);

        if (found)
           return (ix);
        else
           return(ix);
}
```

```
//hashInsert() - Inserting into a hash table and
//               resolving hash collision using
//               linear probing
void    HashTable::insert(dataRecord data,
                          dataRecord dataTable[],
                          bool &dup,  bool &full) {
        int ix, oldIx;
        int idx;

        dataTable[idx = dataArrayLength++] = data;

        // Presuming that it is neither full n
        // or a duplicate
        full = false;
        dup = false;
```

```
        // Set up the starting place
        // for the search of the hash table
        ix = findHashCode(data.keyField);
        oldIx = ix;
        do      {
           if (strcmp(table[ix].keyValue,
                        data.keyField) == 0)
              dup = true;
           else if(table[ix].keyValue[0] != '\0')
              ix = (ix + 1) % hashTableSize;
        } while (!dup
                  && table[ix].keyValue[0] != '\0'
                   && ix == oldIx);
```

```
        if (!dup)
            if (table[ix].keyValue[0] == '\0') {
                strcpy(table[ix].keyValue,
                         data.keyField);
                table[ix].index = idx;
            }
            else
                full = true;

}
```

TestHash.cpp

```
#include        "HashTable.h"
#include        <cstring>
#include        <iostream>

using namespace std;

const int numberBooks = 39;
```

```
char       *bibleBooks[numberBooks]
      = {"Genesis", "Exodus", "Leviticus",
         "Numbers", "Deuteronomy", "Joshua",
         "Judges", "First Samuel", "Second Samuel",
         "First Kings", "Second Kings", "Isaiah",
         "Jeremiah", "Ezekiel", "Hosea", "Joel",
         "Amos", "Obadiah", "Jonah", "Micah",
         "Nahum", "Habakkuk", "Zephaniah",
         "Haggai", "Zechariah", "Malachi", "Psalm",
         "Proverbs", "Job", "Song of Songs", "Ruth",
         "Lamentations", "Ecclesiastes", "Esther",
         "Daniel", "Ezra", "Nehemiah",
         "First Chronicles", "Second Chronicles"
        };
```

```
int     main(void)      {
        int     i, index;
        bool    full, dup;
        dataRecord      data;
        dataRecord      dataArray[dataArraySize];
        HashTable       ht;

        for (i = 0;  i < numberBooks;  i++)     {
           strcpy(data.keyField, bibleBooks[i]);
           ht.insert(data, dataArray, dup, full);
           index = ht.search(bibleBooks[i]);
           cout << "The index for " << bibleBooks[i]
                 << " is " << index << endl;
        }
        return(0);
}
```