CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

LEC04
DICTIONARY DATA STRUCTURES (77-98)

Lecture slide courtesy of Prof. Steven Skiena

PROBLEM OF THE DAY 1: COMPARING DICTIONARY IMPLEMENTATIONS

Problem: What is the asymptotic worst-case running times for each of the seven fundamental dictionary operations when the data structure is implemented as

- A singly-linked unsorted list.
- A doubly-linked unsorted list.
- A singly-linked sorted list.
- A doubly-linked sorted list.

Dictionary operation	Singly unsorted	Double unsorted	Singly sorted	Doubly sorted
Search(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Delete(L, x)	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Successor(L, x)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(n)$	$O(n)^*$	$O(1)$
Minimum(L)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Maximum(L)	$O(n)$	$O(n)$	$O(1)^*$	$O(1)$

The four data structures allow fast search or flexible update, but not fast search *and* flexible update.

PROBLEM OF THE DAY 2

A common problem for compilers and text editors is determining whether the parentheses in a string are balanced and properly nested. For example, the string ((())())() contains properly nested pairs of parentheses, which the strings)()(and
()) do not. Give an algorithm that returns true if a string contains properly nested and balanced parentheses, and false if otherwise.

For full credit, identify the position of the first offending parenthesis if the string is not properly nested and balanced.

REVIEW: DICTIONARY / DYNAMIC SET OPERATIONS

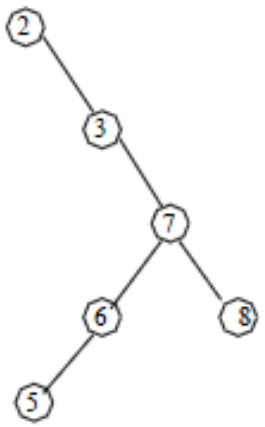
Perhaps the most important class of data structures maintain a set of items, indexed by keys.

- *Search(S,k)* – A query that, given a set S and a key value k, returns a pointer x to an element in S such that key[x] = k, or nil if no such element belongs to S.
- *Insert(S,x)* – A modifying operation that augments the set S with the element x.
- *Delete(S,x)* – Given a pointer x to an element in the set S, remove x from S. Observe we are given a pointer to an element x, not a key value.
- *Min(S), Max(S)* – Returns the element of the totally ordered set S which has the smallest (largest) key.
- *Next(S,x), Previous(S,x)* – Given an element x whose key is from a totally ordered set S, returns the next largest (smallest) element in S, or NIL if x is the maximum (minimum) element.

- There are a variety of implementations of these *dictionary* operations, each of which yield **different** time bounds for various operations.

BINARY SEARCH TREES

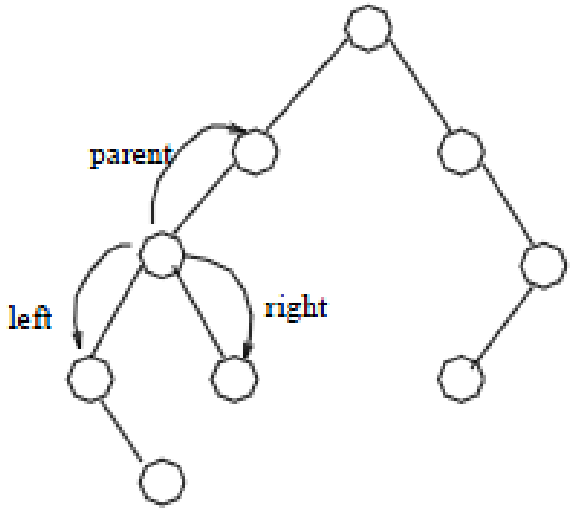
- A binary *search* tree labels each node *x* in a binary tree such that
  - all nodes in the left subtree of x have keys < x and
  - all nodes in the right subtree of x have key's > x.
- The search tree labeling enables us to find where any key is.



* The leaves contain no key and have no structure to distinguish them from one another. Leaves are commonly represented by a special leaf or nil symbol)

BINARY SEARCH TREES

- Binary search trees provide a data structure which efficiently supports all six dictionary operations.
- A binary tree is a rooted tree where each node contains at most two children.
  - Internal nodes of the binary search tree contains a key (and value)
  - Each have two distinguished sub-trees, denoted *left* and *right*.



IMPLEMENTING BINARY SEARCH TREES

```
typedef struct tree {
        item_type item;                 /* data item */
        struct tree *parent;            /* pointer to parent */
        struct tree *left;              /* pointer to left child */
        struct tree *right;             /* pointer to right child */
} tree;
```

- The parent link is optional, since we can store the pointer on a stack when we encounter it.

SEARCHING IN A BINARY TREE: IMPLEMENTATION

Start at the root. Unless it contains the query key *x*, proceed either left or right depending upon whether *x* occurs before or after the root key

recursive search algorithm

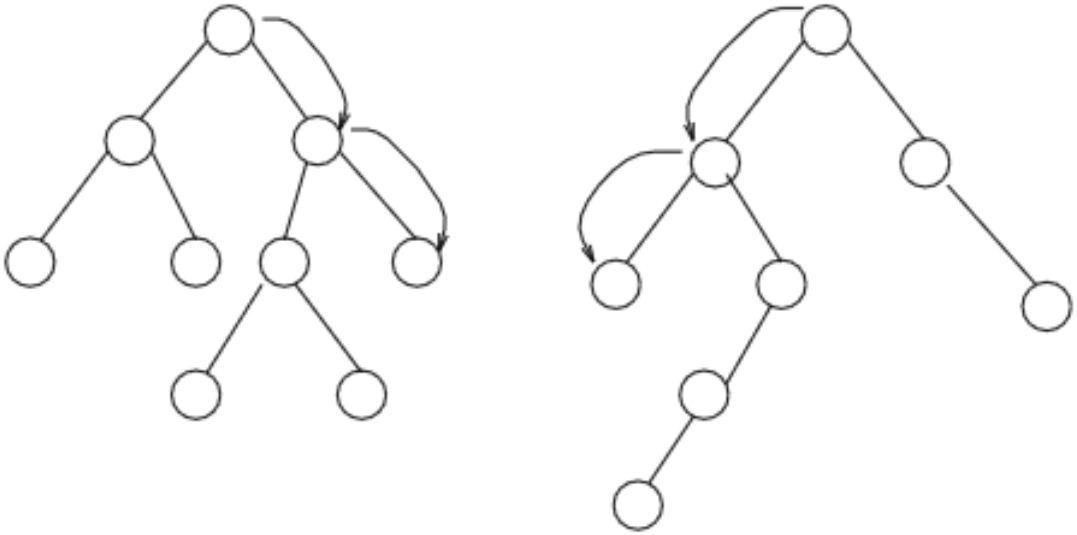
```
tree *search_tree(tree *l, item_type x)
{
        if (l == NULL) return(NULL);

        if (l->item == x) return(l);

        if (x < l->item)
                return( search_tree(l->left, x) );
        else
                return( search_tree(l->right, x) );
}
```

SEARCHING IN A BINARY TREE: HOW MUCH

- The algorithm works because both the left and right subtrees of a binary search tree *are* binary search trees
  - Recursive structure, Recursive algorithm.
- This takes time proportional to the height of the tree, O(h).

MAXIMUM AND MINIMUM

- Where are the maximum and minimum elements in a binary search tree?

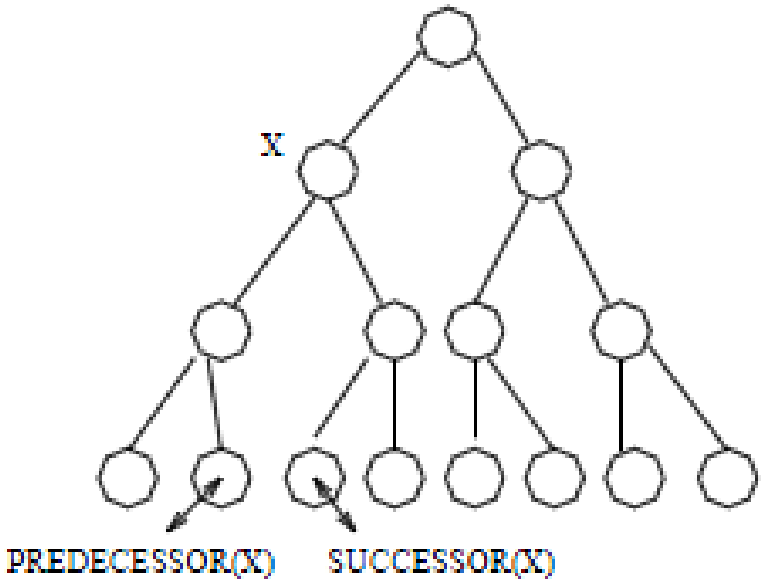


FINDING THE MINIMUM

```
tree *find_minimum(tree *t)
{
        tree *min; (* pointer to minimum *)

        if (t == NULL) return(NULL);

        min = t;
        while (min->left != NULL)
                min = min->left;
        return(min);
}
```

- Finding the max or min takes time proportional to the height of the tree, O(h).

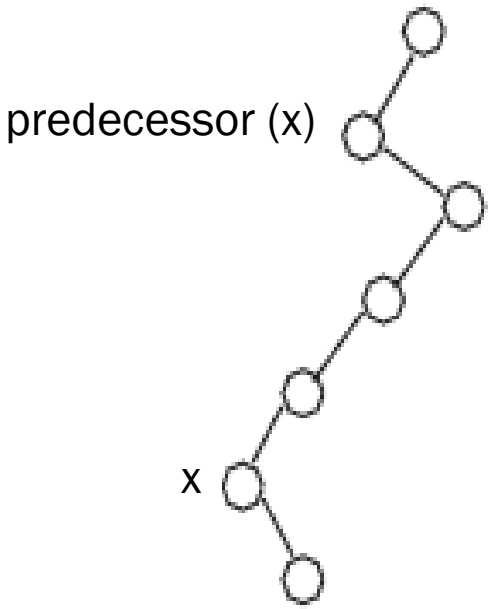
WHERE IS THE PREDECESSOR: INTERNAL NODE



If X has two children, its predecessor is the maximum value in its left subtree and its successor the minimum value in its right subtree.

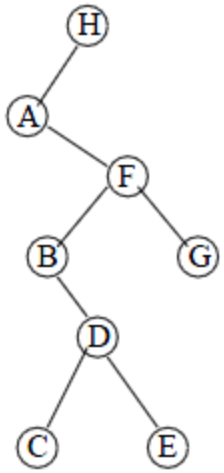
WHERE IS THE SUCCESSOR: LEAF NODE

- If it does **not** have a left child, a node's predecessor is its first left ancestor.
- The proof of correctness comes from looking at the *in-order traversal* of the tree.



IN-ORDER TRAVERSAL

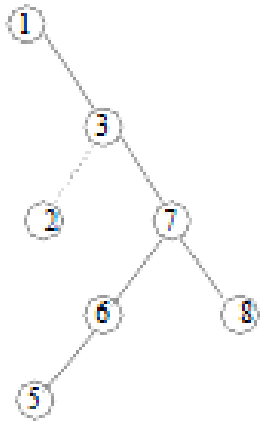
```
void traverse_tree(tree *l)
{
        if (l != NULL) {
                traverse_tree(l->left);
                process_item(l->item);
                traverse_tree(l->right);
        }

}
```



- Binary search trees make it easy to report the labels in sorted order at $O(n)$
- By definition, all the keys smaller than the root must lie in the left subtree of the root, and all keys bigger than the root in the right subtree.
- Thus, visiting the nodes recursively in accord with such a policy produces an *in-order* traversal (DFS) of the search tree.

TREE INSERTION

- Do a binary search to find where it should be,
- then replace the termination NULL pointer with the new item.
- Allocating the node and linking it in to the tree is a constant-time operation a fter the search has been performed in $O(h)$ time.



pointer l to the pointer linking the search subtree to the rest of the tree

key x to be inserted

parent pointer to the parent node containing l (leaf doesn't have storage for parent pointer)

```
insert_tree(tree **l, item_type x, tree *parent)
{
    tree *p;                          /* temporary pointer */

    if (*l == NULL) {
        p = malloc(sizeof(tree)); /* allocate new node */
        p->item = x;
        p->left = p->right = NULL;
        p->parent = parent;
        *l = p;                       /* link into parent's record */
        return;
    }

    if (x < (*l)->item)
        insert_tree(&((*l)->left), x, *l);
    else
        insert_tree(&((*l)->right), x, *l);
}
```

Insert location found

Still searching

This implementation uses recursion to combine the search and node insertion stages of key insertion.

TREE DELETION

- Deletion is trickier than insertion, because the node to die may not be a leaf, and thus effect other nodes.
- There are three cases:
  - Case (a), where the node is a leaf, is simple - just NULL out the parents child pointer.
  - Case (b), where a node has one child, the doomed node can just be cut out and link the grandchild directly to the parent.
  - Case (c), where a node two children, relabel the node as its immediate successor (which has at most one child when z has two children!) and delete the successor!
- Every deletion requires the cost of at most two search operations, each taking $O(h)$ time where h is the height of the tree, plus a constant amount of pointer manipulation

CASES OF DELETION

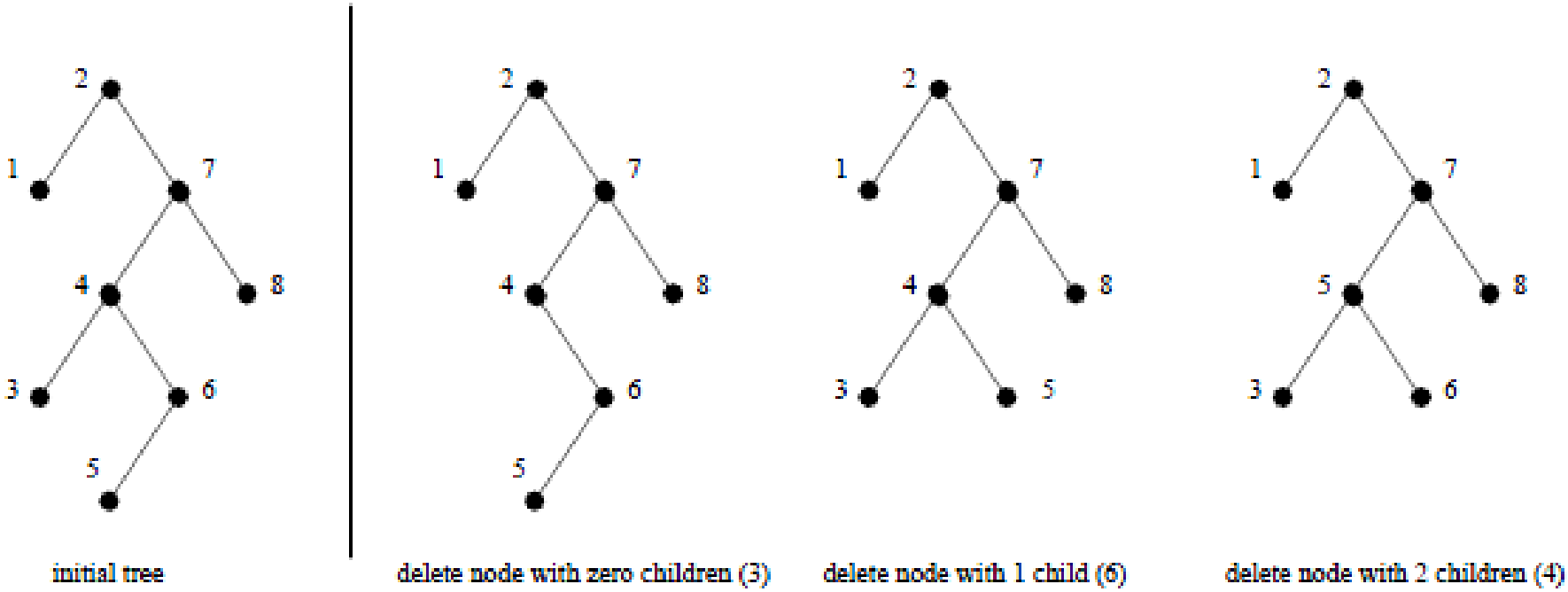


Figure 3.4: Deleting tree nodes with 0, 1, and 2 children

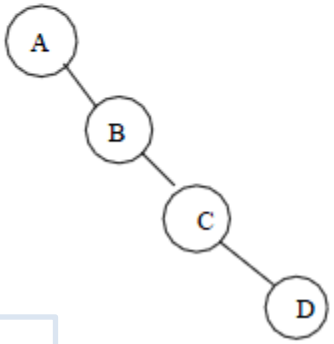
BINARY SEARCH TREES AS DICTIONARIES

- All six of our dictionary operations, when implemented with binary search trees, take O(h), where h is the height of the tree.
- The best height we could hope to get is lgn, if the tree was perfectly balanced, since $\sum_{i=0}^{\lfloor \lg n \rfloor} 2^i \approx n$
- But if we get unlucky with our order of insertion or deletion, we could get linear height!

WORST CASE AND AVERAGE HEIGHT

$\text{insert}(a)$
$\text{insert}(b)$
$\text{insert}(c)$
$\text{insert}(d)$

bad things can happen when building trees through insertion.



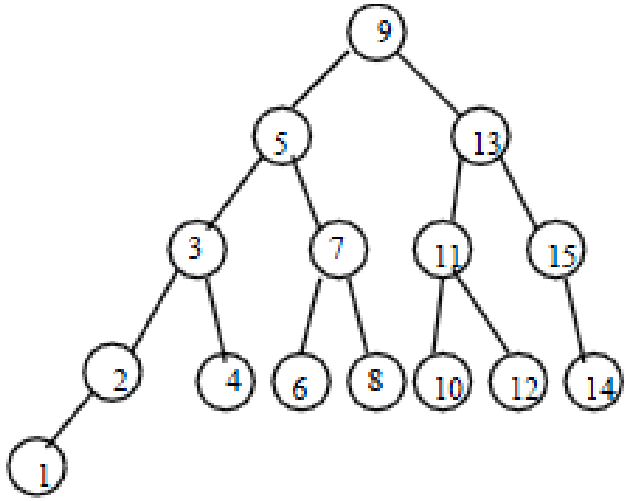
But with high probability the resulting tree will have O(log n) height ☺

TREE INSERTION ANALYSIS

- In fact, binary search trees constructed with random insertion orders *on average* have (lgn) height.
- The worst case is linear, however.
- Our analysis of Quicksort will later explain why the expected height is θ(lg n).

PERFECTLY BALANCED TREES

Perfectly balanced trees require a lot of work to maintain:



- If we insert the key 1, we must move every single node in the tree to rebalance it, taking (n) time.

BALANCED SEARCH TREES

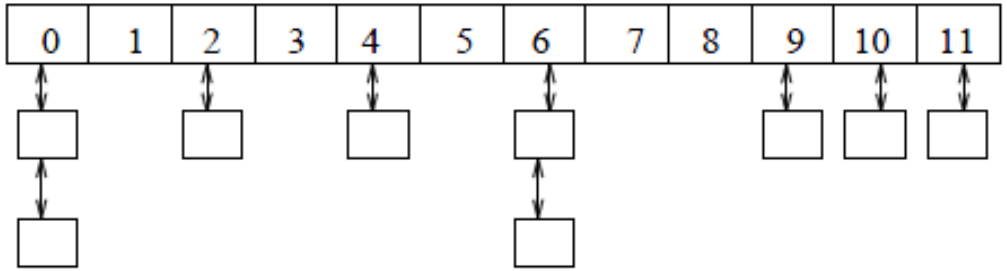
- Therefore, when we talk about "balanced" trees, we mean trees whose height is O(lg n),
  - so all dictionary operations (insert, delete, search, min/max, successor/predecessor) take O(lg n) time.
- Extra care must be taken on insertion and deletion to guarantee such performance, by rearranging things when they get too lopsided.
- Examples of balanced search trees used in practice
  - *Red-Black trees (discussed in Section 12.1)*, *AVL trees*, *2-3 trees*, *splay trees*, and *B-trees*

HASH TABLES

- Hash tables are a very practical way to maintain a dictionary.
- The idea is simply that looking an item up in an array is $\theta(1)$ once you have its index.
- A hash function is a mathematical function which maps keys to integers (index into an array).

COLLISIONS

- *Collisions* are the set of keys mapped to the same bucket.
- If the keys are uniformly distributed, then each bucket should contain very few keys!
- The resulting short lists are easily searched!



- Collision Resolution: Chaining, open addressing, etc (review your cse214 class notes)

HASH FUNCTIONS

- It is the job of the hash function to map keys to integers.
- A good hash function:
  - 1. Is cheap to evaluate
  - 2. Tends to use all positions from 0 … M with uniform frequency.
- The first step is usually to map the key (ex> a string) to a big integer, for example

$$h = \sum_{i=0}^{keylength} 128^i \times char(key[i])$$

MODULAR ARITHMETIC

- This large number must be reduced to an integer whose size is between 1 and the size of our hash table.
- One way is by h(k) = k mod M, where M is best a large prime not too close to 2^i - 1, which would just mask off the high bits.
- This works on the same principle as a roulette wheel!

PERFORMANCE ON SET OPERATIONS

- With either chaining or open addressing:
  - Search - O(1) expected, O(n) worst case
  - Insert - O(1) expected, O(n) worst case
  - Delete - O(1) expected, O(n) worst case
  - Min, Max and Predecessor, Successor θ(n+m) expected and worst case
- Pragmatically, a hash table is often the best data structure to maintain a dictionary.
- However, the worst-case time is unpredictable.
- The best worst-case bounds come from balanced binary trees.

RUNNING TIME

Chaining with doubly-linked lists to resolve collisions in an *m*-element hash table, the dictionary operations for *n* items

	Hash table (expected)	Hash table (worst case)
Search(L, k)	$O(n/m)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$
Delete(L, x)	$O(1)$	$O(1)$
Successor(L, x)	$O(n+m)$	$O(n+m)$
Predecessor(L, x)	$O(n+m)$	$O(n+m)$
Minimum(L)	$O(n+m)$	$O(n+m)$
Maximum(L)	$O(n+m)$	$O(n+m)$

SUBSTRING PATTERN MATCHING

- Problem: Substring Pattern Matching
- Input: A text string t and a pattern string p.
- Output: Does t contain the pattern p as a substring, and if so where?

- E.g: Is *Skiena* in the Bible?

BRUTE FORCE SEARCH

- The simplest algorithm to search for the presence of pattern string p in text t overlays the pattern string at every position in the text, and checks whether every pattern character matches the corresponding text character.
- This runs in $O(nm)$ time, where $n = |t|$ and $m = |p|$

STRING MATCHING VIA HASHING

- Suppose we compute a given hash function on both the pattern string p and the m-character substring starting from the ith position of t.
- If these two strings are identical, clearly the resulting hash values will be the same.
- If the two strings are different, the hash values will almost certainly be different.
- These false positives should be so rare that we can easily spend the O(m) time it take to explicitly check the identity of two strings whenever the hash values agree.

THE CATCH

- This reduces string matching to *n - m + 2* hash value computations
  - (the *n - m + 1* windows of t, plus one hash of p),
- plus what should be a very small number of O(m) time verification steps.
- The catch is that it takes *O(m)* time to compute a hash function on an *m*-character string, and *O(n)* such computations seems to leave us with an *O(mn)* algorithm again.

THE TRICK: RABIN-KARP ALGORITHM.

- Look closely at our string hash function, applied to the m characters starting from the jth position of string S:

$$H(S,j) = \sum_{i=0}^{m-1} \alpha^{m-(i+1)} \times char(s_{i+j})$$

- A little algebra reveals that

$$H(S,j+1) = (H(S,j) - \alpha^{m-1} char(s_j))\alpha + s_{j+m}$$

  - Thus once we know the hash value from the j position, we can find the hash value from the (j + 1)st position for the cost of two multiplications, one addition, and one subtraction.
  - This can be done in constant time.

- Linear expected-time algorithm for string matching,

HASHING, HASHING, AND HASHING

Udi Manber says that the three most important algorithms at Yahoo are hashing, hashing, and hashing.

Hashing has a variety of clever applications beyond just speeding up search, by giving you a short but distinctive representation of a larger document.

- *Is this new document different from the rest in a large corpus?* – Hash the new document, and compare it to the hash codes of corpus.
- *How can I convince you that a file isn't changed?* – Check if the cryptographic hash code of the file you give me today is the same as that of the original. Any changes to the file will change the hash code.