Chapter 5 Arrays and Strings

5.1 Arrays as abstract data types

5.2 Contiguous representations of arrays

5.3 Sparse arrays

5.4 Representations of strings

5.5 String searching algorithms

5.1 Arrays as abstract data types

An array is a list of elements whom can be accessed with their *indices*

An array is a function, e.g, defined as

$$X : \{0, 1, \ldots, n-1\} \rightarrow V$$

where V is the element value set and n is the array length.

In general,

l: index lower bound

u: index upper bound

length $= u - l + 1$

abstract operations on arrays

$Access(X, i)$: return $X[i]$

$Length(X)$: return $u - l + 1$

$Assign(X, i, v)$: replace X with a function whose value on i is v, and the rest remains unchanged; the same as $X[i] \leftarrow v$

$Initialize(X, v)$: assign v to every element of X

$Iterate(X, F)$: apply F to each element of X in order, the same as **for** $i = l$ **to** u **do** $F(X[i])$

An array is different from a table

array: a logical structure
table: a physical structure

String: a special type of array, e.g.,

$$S : \{0, 1, \ldots, n\} \to \Sigma$$

where Σ is the *alphabet* that can be any finite set, e.g., the English alphabet.

S is called *string over* Σ.

Each element in Σ is called a *character*

$Length(S) = n$

ϵ is a string of length 0

Two more operations for strings:

$Substring(w, i, m)$: return the string w' of length m that contains

the portion of w that starts at position i , $0 \leq i < |w|$, and $0 \leq m \leq |w| - i$, i.e.,

$$w'[k] = w[i + k] \qquad 0 \leq k < m$$

$Substring(w, 0, j)$ is called a *prefix* of w
$Substring(w, j, |w| - j)$ is called a *suffix* of w

$Concat(w_1, w_2)$: return a string w of length $|w_1| + |w_2|$,
which is w_1 followed by w_2, i.e.,
$w[k] = w_1[k]$ for $0 \leq k < |w_1|$
$w[k] = w_2[k - |w_1|]$ for $|w_1| \leq k < |w_1| + |w_2|$
$Concat(w, \epsilon) = w$, $Concat(\epsilon, w) = w$

Multi-dimensional arrays:

$$A : I_1 \times I_2 \times \ldots \times I_d \longrightarrow V$$

where I_i is the index set of the ith dimension.

5.2 Contiguous representations of arrays

Simply arrange memory locations for array elements in order.

If every element takes L memory locations, $X[i]$ is stored in the location $\chi + L(i - l)$, where χ is the start address of X.

For multi-dimensional arrays:

row major order:

column major order:

efficiency issues, pre-computation

can operation *Iterate* be made efficient?

Constant-time initialization

use enough memory: constant-time initialization and access (different from implementation with linked memory)

The idea:

(1) maintain a $Count(M)$ for the number of different elements modified since the last time the array M was initialized.

(2) Table $Which(M)$ remembers which elements of M have been modified, $0 \leq j \leq Count - 1$, $Which(M)[j] = i$ iff $M[i]$ has been modified since the last initialization

(3) $Data(M)[i]$ stores the value of $M[i]$

(4) $When(M)[i]$ gives the location in $Which(M)$ where i can be found.

M is represented as a record of five fields:

Data(M), Which(M), When(M), Count(M), Default(M).

procedure *Initialize*(**pointer** M, **value** v):
{Initialize each element of M to v}
Count(M) $\leftarrow 0$
Default(M) $\leftarrow v$

function *Valid*(**integer** i, **pointer** M): **boolean**
{Return **true** if $M[i]$ has been modified since the last *Initialize*}
return $0 \leq$When(M)$[i]$ $<$Count(M) **and**
Which(M)[When(M)$[i]$] $= i$

procedure *Access*(**integer** i, **pointer** M) : **value**
{Return $M[i]$}
  if $Valid(i, M)$ **then**
    return Data$(M)[i]$
  else
    return **Dafault(M)**

procedure *Assign*(**pointer** M, **integer** i, **value** v)**:**
{**Set** $M[i] \leftarrow v$}
  if not $Valid(i, M)$ **then**
    When$(M)[i] \leftarrow$ **Count**(M)
    Which$(M)[$**Count**$(M)] \leftarrow i$
    Count$(M) \leftarrow$ **Count**$(M) + 1$
  Data$(M)[i] \leftarrow v$

5.3 Sparse arrays

Arrays that have mostly zero entries

- Linked-list representations
  - advantages?
  - disadvantages?
- Example: 2-D array represented by doubly-linked lists
  - What fields does each record need?

Arrays with special shapes

5.4 Representations of Strings

Physical storage issues:

(1) contiguous memory

(2) encoding of characters in Σ

fixed length encoding: at least $\lceil \log_2 |\Sigma| \rceil$ binary bits are needed

variable length encoding:
short sequences to encode common characters
long sequences to encode rare characters

the "decodability" issue

Encoding tree (prefix code)

an encoding tree is a full binary tree.

Decoding with encoding tree

procedure $TreeDecode$(**pointer** T, **bitstream** b)**:**
    $P \leftarrow T$
    while not $Empty(b)$ **do**
        if $Nextbit(b) = 0$
            then $P \leftarrow LC(P)$
            else $P \leftarrow RC(P)$
        if $IsLeaf(P)$ **then**
            $OutputChar$(**Char**(P))
            $P \leftarrow T$

Huffman encoding tree

Algorithm HUFFMAN(Σ, f): **pointer**

(1) choose $a, b \in \Sigma$ with the lowest frequencies f_a and f_b,
(2) create new character $\hat{ab}$
$\Sigma = \Sigma - \{a, b\} \cup \{\hat{ab}\}$
$f_{\hat{ab}} = f_a + f_b$
(3) $T \leftarrow$HUFFMAN(Σ, f)
(4) create nodes N_a and N_b, and in T do
$LC(N_{\hat{ab}}) \leftarrow N_a$
$RC(N_{\hat{ab}}) \leftarrow N_b$
(5) return T

Huffman encoding tree is the optimal

How to measure the optimality?

Weighted path length (WPL) of an encoding tree T:

$$WPL(T) = \sum_{c \in Leaves(T)} Depth_T(c) f(c)$$

Definition: An encoding T for Σ is called the *optimal* if its $WPL(T)$ is the smallest.

Why should WPL be the right measure?

consider the following

Let F be a text file containing n characters
T be an encoding tree for character set Σ with frequency function f

Then the *size of the encoded file* F is

$$\sum_{c \in \Sigma} n f(c) Depth_T(c) = n \sum_{c \in \Sigma} Depth_T(c) f(c)$$

proportional to $WPL(T)$. So the smaller it is, the smaller the encoded file will be.

Lemma: Given a encoding tree T with characters a, b being the two with the lowest frequencies. Then there is another encoding tree T' constructed from T such that

(1) the leaves of T' are the same as the leaves of T, except that a, b are not leaves of T' and T' has a new leaf c.

(2) the frequency of $f(c) = f(a) + f(b)$ and frequencies of all other leaves of T' are the same as their frequencies in T.

(3) $WPL(T') \leq WPL(T) - f(c)$ with equality if a, b are siblings.

PROOF:

(1) Case 1: a, b are sibling in T. Then delete them and designate their parent as the new leaf c in T'.

Let $f(c) = f(a) + f(b)$. Assume d is the depth of a and b. Then the difference between

$$WPL(T') - WPL(T) = (d-1)f(c) - d(f(a) + f(b)) = -f(c)$$

So $WPL(T') = WPL(T) - f(c)$ or $WPL(T') \leq WPL(T) - f(c)$

(2) Case 2: a, b are NOT sibling in T but they have the same depth. Simply exchange the sibling of a with b in T. This does not change $WPL(T)$. Then we use the argument for (1) to prove.

(3) Case 3: a, b are NOT sibling in T and $Depth_T(a) > Depth_T(b)$. Exchange the sibling of a with b. This will decrease $WPL(T)$ or keep it unchanged. Then we use the argument for (1) to prove.

Theorem (*Huffman Optimality*) Let T be the Huffman encoding tree and X be any encoding tree for a given Σ.
Then $WPL(T) \leq WPL(X)$.

PROOF:

By induction on the number of characters in Σ.

Base: when $|\Sigma| = 2$, The claim holds.

Assume for $|\Sigma| = k$, $WPL(T) \leq WPL(X)$.

Now $|\Sigma| = k + 1$ and assume T and X to be the Huffman encoding and an arbitrary encoding trees repsectively.

We use the Lemma to construct T' and X' that satisfy

$$WPL(T') = WPL(T) - f(a) - f(b)$$
$$WPL(X') \leq WPL(X) - f(a) - f(b)$$

Based on the assumption, $WPL(T') \leq WPL(X')$. So

$$WPL(T) \leq WPL(X)$$

5.5 String searching algorithms

Searching a target for a pattern:

INPUT: string p and t over the alphabet Σ
OUTPUT: k, such that $p = Substring(t, k, |p|)$

simple search algorithm

Knuth-Morries-Pratt algorithm

Boyer-Moore algorithm

A simple search algorithm

function $SimpleStringSearch(\textbf{string}\, p, t) : \textbf{integer}$

```
for k from 0 to Length(t) − Length(p) do
    i ← 0
    while i < Length(p) and p[i] = t[k + i] do
        i ← i + 1
    if i = Length(p) then return k
return −1
```

Time Complexity: $O(|p||t|)$

Knuth-Morris-Pratt Algorithm

Basic ideas:

forsee mismatches

$p = ABCD$ and $t = ABCEFGABCD$,
after a mismatch, move p rightward as many positions as possible.

a complex case: $p = XYXYZ$ and $t = XYXYXY...$

correct motion depends on mismatching character as well as
the location of mismatch.

example:

```
X Y X Y X Y c
X Y X Y Z
    X Y X Y Z
```

(1) if `c` is `X`, p is moved two positions rightward;

(2) if `c` is `Q`, p is moved five positions rightward, or

(3) if `c` is `Z`, pattern is found.

A more general case

```
       t[k]   t[k+1]      t[k+i]        t[k+m-1]
  ...  L          E ... S    c       ...     ...
       L          E ... S    D    E ... R    G
       p[0]       p[1]  p[i-1]p[i]p[i+1]     p[m-1]
```

$p[0] = t[k], \ldots, p[i-1] = t[k+i-1]$
but $p[i] \neq t[k+i]$

find d such that
  $p[0] = t[k+d]$, $p[1] = t[k+d+1]$, and up to $p[i-d] = t[k+i]$,
  and d is the smallest.

if no such d, $d = i+1$ always works.

	$t[k]$	$t[k+1]$			$t[k+i]$		$t[k+m-1]$
...	`L`	`E`	...	`S`	c	...	...
	`L`	`E`	...	`S`	`D`	`E`...`R`	`G`
	$p[0]$	$p[1]$		$p[i\text{-}1]$	$p[i]$	$p[i\text{+}1]$	$p[m\text{-}1]$

$p[0] = t[k], \ldots, p[i-1] = t[k+i-1]$
but $p[i] \neq t[k+i]$

Observations:

$p[0] = p[k+d]$ is equivalent to $p0] = p[d]$ because $p[d] = p[k+d]$

d depends on the pattern p, i, and mismatching character $t[k+i] = c$.

	$t[k]$	$t[k+1]$			$t[k+i]$			$t[k+m-1]$
...	`L`	`E`	...	`S`	c	...		...
	`L`	`E`	...	`S`	`D`	`E` ...	`R`	`G`
	$p[0]$	$p[1]$		$p[i\text{-}1]$	$p[i]$	$p[i{+}1]$		$p[m\text{-}1]$

$p[0] = t[k], \ldots, p[i-1] = t[k+i-1]$
but $p[i] \neq t[k+i]$

Define a function to yield d:

Define $KMPskip[p, i, c]$ to be the smallest d, $0 \leq d \leq i$, such that

(1) $p[i-d] = c$
(2) $p[j] = p[j+d]$, for every $0 \leq j \leq i-d-1$

and let $KMPskip[p, i, c] = i+1$, if no such d exists.

Compute $KMPskip[p, i, c]$ once for all for pattern p

compute an array $KMPskiparray[i, c]$ for all $0 \leq i < |p|$ and $c \in \Sigma$

assign $KMPskiparray[i, c] = KMPskip[p, i, c]$

example:

p =	A	B	C	D
A	0	1	2	3
B	1	0	3	4
C	1	2	0	4
D	1	2	3	0
other	1	2	3	4

p =	X	Y	X	Y	Z
X	0	1	0	3	2
Y	1	0	3	0	5
Z	1	2	3	4	0
other	1	2	3	4	5

Define $PrefSuf(w)$ to be the largest $j < |w|$ such that

$Substring(w, 0, j) = Substring(w, |w| - j, j)$

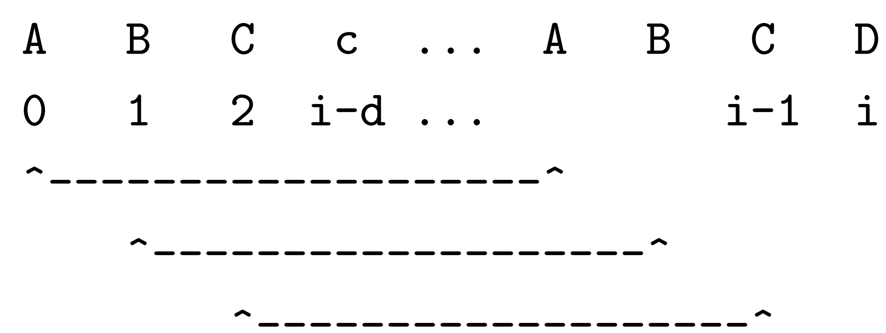
I.e., it is the length of the longest prefix of w that is the suffix of w.

e.g., $PrefSuf($`ABCAB`$) = 2$ and $PrefSuf($`AAAAA`$) = 4$

Note: $KMPskip[p, i, c]$ is the smallest d, $0 \leq d \leq i$, such that

(1) $p[i - d] = c$

(2) $p[j] = p[j + d]$, for every $0 \leq j \leq i - d - 1$



```
function AllPrefSuf(string p) : array
{ The result ps is an array of integers with the same indices as p}
    ps[0] ← 0
    for j from 1 to length(p) − 1 do
        ps[j] ← Extend(ps[j − 1], j)
    return ps

function Extend(integer i, j) : integer
{Chain through ps looking for an extendible value}
    if p[j] = p[i] then return i + 1
    if i = 0 then return 0
    return Extend(ps[i − 1], j)
```

Example to show how the algorithm works:

$p\ $ = `A B E A B X Y A B E A B E`

ps = `0 0 0 1 2 0 0 1 2 3 4 5 ?`

when $j = 13$, $ps[j-1] = 5$

call $Extend(ps[j-1], 13)$

then $i = 5$, $p[5] = `X' \neq `E' = p[13]$

then $ps[i-1] = ps[4] = 2$

call $Extend(2, 15)$

then $i = 2$, $p[2] = `E' = p[13]$

return $i + 1 = 3$

$KMPskiparray[i, c]$ can be computed using $ps[i]$.
(Questions 41 and 42 on page 172)

function $KMPSearch$(**string** p, t) : **integer**
{Find p in tl return its location or -1}

```
    KMPskiparray ← ComputeKMPskilarray(p)
    k = 0
    i = 0
    while k ≤ Length(t) − length(p) do
        if i = length(p) then return k
        d ← KMPskiparray[i, t[k + i]]
        k ← k + d
        i ← i + 1 − d
    return
```

Time complexity: $O(|t| + |p|)$**, why?**

Chapter 6. List and Tree Implementations of Sets

6.1 Sets and dictionaries as ADTs

Define SET to be an abstract data type with operations:

$Member(x, S)$: return **true** if and only if $x \in S$

$Union(S, T)$: return $S \cup T$, whose elements are either from S or T

$Intersection(S, T)$: return $S \cap T$, whose elements are in both S and T

$Difference(S, T)$: return $S - T$, whose elements are in S but T

$MakeEmptySet(S)$: return ϕ

$IsEmptySet(S)$:

$Size(S)$: return $|S|$, the number of elements in S

$Insert(x, S)$: add x to S

$Delete(x, S)$: remove x from S

$Equal(S, T)$: return **true** if and only if S and T are identical

$Iterate(S, F)$: perform F on every element in S

Set elements $\langle K, I \rangle$ are identified with their unique keys K along with data information I.

Dictionary is a set ADT that consists of of the following operations:

$MakeEmptySet(S)$, $IsEmptySet(S)$, $Insert$,
$Delete$, and $LookUp$ which is similar to $Member$

$LookUp(K, S)$: return an **info** I such that $\langle K, I \rangle \in S$, and Λ if there is no such member.

6.2 Unordered lists

Implementations for lists (chapter 3) can be used to implement unordered lists as well.

How difficult to do *LookUp* ?

worst case: n number of comparisons $\Theta(n)$

average case: (total num of comparisons) / (num of items)

$= \frac{(1+2+\ldots+n)}{n} = \frac{n+1}{2}$

or expected num of comparisons

$= 1 \times \frac{1}{n} + 2 \times \frac{1}{n} + \ldots + n \times \frac{1}{n} = \frac{n+1}{2}$

based on the equal probability distribution.

Consider general case where key K_i has probability p_i to be looked up where $\sum_{i=1}^{n} p_i = 1$ and the expected search time is $\sum_{i=1}^{n} ip_i$ which is the minimum when $p_1 \geq p_2 \geq \ldots \geq p_n$.

(**argument is the same as used for Hoffman encoding**)

6.3 Ordered lists

Binary search

function $BinarySearchLookup$(**key** K, **table** $T[0..n-1]$)**: info**
    $Left \leftarrow 0$
    $Right \leftarrow n - 1$
    repeat forever
        if $Right < Left$ **then**
            return Λ
        else
            $Middle \leftarrow \lfloor (Left + Right)/2 \rfloor$
            if $K = Key(T[Middle])$ **then return** $Info(T[Middle])$
            else if $K < Key(T[Middle])$ **then** $Right \leftarrow Middle - 1$
            else $Left \leftarrow Middle + 1$

Execution tree on table of size 10 (i.e., with indices 0..9**)**

Called it an *extended binary tree*

```
0, 4, 9
    0, 1, 3
        0, 0, 0
            0, , -1
            1, , 0
        2, 2, 3
            2, , 1
            3, 3, 3
                3, , 2
                4, , 3
    5, 7, 9
        5, 5, 6
            5, , 4
            6, 6, 6
                6, , 5
                7, , 6
        8, 8, 9
            8, , 7
            9, 9, 9
                9, , 8
                10, , 9
```

THEOREM (*Binary Search*) The binary search uses at most $O(\log_2 n)$ comparisons.

PROOF: use recurrence $T(n) = T(\lfloor \frac{n}{2} \rfloor) + c$ and $T(1) = 0$ for some constant $c > 0$ to prove. Assume that $n = 2^k$, and $k = \log_2 n$

Template: $T(m) = T(\frac{m}{2}) + c$

$$T(n) = T(\tfrac{n}{2}) + c$$
$$T(\tfrac{n}{2}) = T(\tfrac{n}{2^2}) + c$$
$$\cdots\cdots$$
$$T(\tfrac{n}{2^{k-1}}) = T(\tfrac{n}{2^k}) + c$$

Sum the equations:

left side $= T(n)$,
right side $= T(\frac{n}{2^k}) + c + c + \ldots + c = 0 + c \log_2 n = O(\log_2 n)$

Interpolation Search Assume

$$0 \leq \text{Key(T[0])} \leq \text{Key(T[1])} \leq \ldots \leq \text{Key(T[n}-1\text{])} \leq \text{N}-1$$

function $InterpolationSearchLookUp$(**key** K, **table** $T[0..n-1]$) : **info**

key($T[-1]$) $\leftarrow -1$

Key[$T(n)$] $\leftarrow N$

$Left \leftarrow 0$

$Right \leftarrow n-1$

repeat forever

if $Right < Left$ **then return** Λ

else

$p \leftarrow \frac{K - \text{Key(T[Left}-1]-1)}{\text{Key(T[Right}+1])-\text{Key(T[Left}-1])}$

$Middle \leftarrow \lfloor p \cdot (Right - Left + 1) \rfloor + Left$

if $K = \text{Key(T[Middle])}$ **then return** Info(T[Middle])

else if $K < \text{Key(T[Middle])}$ **then** $Right \leftarrow Middle - 1$

else $Left \leftarrow Middle + 1$

example: assume $n = 10, N = 1000,\ K = 316$

```
-1   0    1    2    3    4    5    6    7    8    9    10
-1   11   72   93   260  316  431  788  798  903  910  1000
```

$Left = 0,\ Right = 9$

$$p \leftarrow \frac{316 - (-1)}{1000 - (-1)} = 0.31668$$

$Middle \leftarrow \lfloor 0.31668(9 - 0 + 1) \rfloor + 0 = 3,\ T[3] = 260 < K,\ Left \leftarrow 4$

$$p \leftarrow \frac{316 - 260}{1000 - 260} = 0.07568$$

$Middle \leftarrow \lfloor 0.07568(9 - 4 + 1) \rfloor + 4$

worst case time: $O(n)$

average case time: $O(\log \log n)$

6.4 Binary search trees

recursive LookUp:

function $BinaryTreeLookUp$(**key** K, **pointer** P) : **info**
  if $P = \Lambda$ **then return** Λ
  else if $K = \text{Key(P)}$ **then return** Info(P)
  else if $K < \text{Key(P)}$ **then return** $BinaryTreeLookUp(K, LC(P))$
  else return $BinaryTreeLookUp(K, RC(P))$

---

non-recursive LookUp:

function $BinaryTreeLookUp$(**key** K, **pointer** P) : **info**
  while $P \neq \Lambda$ **do**
    if $K = \text{Key(P)}$ **then return** Info(P)
    else if $K < \text{Key(P)}$ **then** $P \leftarrow LC(P)$
    else $P \leftarrow RC(P)$
  return Λ

Insertion (at leaves):

```
procedure BinaryTreeInsert(key K, info I, pointer P) :
    R ← Λ
    while P ≠ Λ do
        if K = Key(P) then return Info(P) ← I
        else if K < Key(P)
            then R ← P, P ← LC(P)
            else R ← P, P ← RC(P)


    P ← NewCell(Node)
    Key(P) ← K,  Info(P) ← I
    LC(P) ← Λ,  RC(P) ← Λ
    if R ≠ Λ then
        if Key(R) > K
            then LC(R) ← P
            else RC(R) ← P
```

deletion:

Several cases to consider:

(1) deleting a leaf

(2) deleting an internal node that has only one child

(3) deleting an internal node that has two children

find inorder successor or predecessor

procedure $BinaryTreeDelete$(**key** K, **pointer** P) :

$R \leftarrow \Lambda$

while $P \neq \Lambda$ **and** Key(P) $\neq$ K **do**

$R \leftarrow P$

if $K <$ Key(P)

then $P \leftarrow LC(P)$**,** $C =' L'$

else $P \leftarrow RC(P)$**,** $C =' R'$

if $P = \Lambda$ **then return**

if $RC(P) = \Lambda$ **then**

if $C =' L'$ **then** $LC(R) \leftarrow LC(P)$

else $RC(R) \leftarrow LC(P)$

if $LC(P) = \Lambda$ **then**

if $C =' L'$ **then** $LC(R) \leftarrow RC(P)$

else $RC(R) \leftarrow RC(P)$

else...

else

{ find the inorder successor}

$Q \leftarrow RC(P)$, $T \leftarrow P$ {T records the parents of Q}

while $LC(Q) \neq \Lambda$ **do**

$T \leftarrow Q$

$Q \leftarrow LC(Q)$

{replace the node P to be deleted by its inorder successor Q}

$LC(T) \leftarrow RC(Q)$

$LC(Q) \leftarrow LC(P)$

$RC(Q) \leftarrow RC(P)$

if $C =' L'$ **then** $LC(R) = Q$

else $RC(R) \leftarrow Q$

6.5 Static binary search trees

Given a dictionary of n keys $K_1 < K_2 < \ldots < K_n$ with probability p_i of accessing K_i, $\sum_{i=1}^{n} p_i = 1$:

Optimal binary search tree T is defined to be such that the **cost** of the tree

$$C(T) = \sum_{i=1}^{n} p_i(Depth_T(K_i) + 1)$$

achieves the minimum.

But *how to construct such a tree* ?

Optimal binary search tree

Consider the following the more general case of construct an optimal binary search tree for keys $K_i < K_2 < \ldots < K_j$, where $i \leq j$

Define $c(i, j)$ to be the minimum cost of a binary search tree for keys $K_i < K_2 < \ldots < K_j$.

Then

$$c(i, j) = \min_{i \leq k \leq j} \{c(i, k-1) + c(k+1, j) + p_k\}$$

Base cases

$c(i, i) = p_i$

$c(i, i-1) = 0$

Directly compute $c(i, j)$ recursively would take too much time.

We use *dynamic programming* to compute $c(i, j)$ ....

The following procedure computes table $C_{n\times n}$ such that $C[i,j] = c(i,j)$. Our goal is $C[1,n]$

procedure *OptimalBinarySearchTree*(**set of real** P, **integer** n) :

```
for i = 1 to n do
    C[i, i − 1] ← 0
for d = 0 to n − 1 do
    for i = 1 to n − d do
        j = i + d
        for k = i to j do
            m = +∞
            if C[i, k − 1] + C[k + 1, j] + p_k < m
                then m = C[i, k − 1] + C[k + 1, j] + p_k
```

Table C, example

Chapter 7. Tree Structures for Dynamic Dictionaries

We will discuss tree implementation of dictionaries that dynamically change (i.e., involving *insertion*s and *deletion*s in addition to *lookup*s).

AVL trees
2-3 trees
B-trees and
red-black trees

7.1 AVL trees

Let T be any binary tree and v be a node in the tree define

$$LeftHeight(v) = \begin{cases} 0 & \text{if } LC(v) = \Lambda \\ 1 + Height(LC(v)) & \text{otherwise} \end{cases}$$

$$RightHeight(v) = \begin{cases} 0 & \text{if } RC(v) = \Lambda \\ 1 + Height(RC(v)) & \text{otherwise} \end{cases}$$

$LeftHeight(T) = LeftHeight(r)$ and
$RightHeight(T) = RightHeight(r)$ where r is the root of T

for any leaf l,

$$LeftHeight(l) = RightHeight(l) = 0$$

The *balance* of node v is

$$RightHeight(v) - Leftheight(v)$$

Tree T is an AVL tree if the balance of every node in T is -1, 0, or +1.

Figure 7.1 (page 220) examples of trees with $LeftHeight$ and $RightHeight$ for all nodes, some are AVL

THEOREM (*AVL Tree Height*) Any AVL tree with n nodes has height $O(\log n)$.

Representation of an AVL tree:

Fields include LC, RC and *Balance* for every node

Insertion by the following steps:

(1) Use the standard binary tree insertion – trace the path from root and insert the node as a new leaf.

(2) Retrace the path from the new leaf to the root, updating the balances along the way.

(3) If a node encountered has balance $+2$ or -2, re-adjust the subtrees of that node and its descendants by *rotation methods*.

Simple cases of insertions (Figure 7.3, page 223)

No change in the height of the tree, still AVL tree

Change in the height of the tree, still AVL tree.

Complex cases of insertions that cause balances out of range (Figure 7.4)

(1) Node A (balance $= +1$) has right child C (balance $= 0$).
When a leaf is inserted into the right subtree of C, the balance of A becomes $+2$.

Single left rotation: making A to be the left child of C; move the left subtree of C to be the right subtree of A

Parenthetical notation:

$$(T_1 A (T_2 C T_3)) \longrightarrow ((T_1 A T_2) C T_3)$$

Symmetrically, there is *single right rotation*

Complex cases of insertions that cause balances out of range (Figure 7.4)

(2) Node A (balance $= +1$) has right child C (balance $= 0$).
C has left child B (balance $= 0$).
When a leaf is inserted into either of the subtrees of B, the balance of A becomes $+2$

Double RL rotation:

First single right rotation from B to C (i.e., make C to be the right child of B; move the right subtree of B to be the left subtree of C)

Then single left rotation from B to A (i.e., make A to be the left child of B; move the left subtree of B to be the right subtree of A)

Symmetrically, there is *double LR rotation*

Are there double RR or double LL rotation?

Deletion by the following steps:

(1) Follow the **standard binary tree deletion**
  (a) deleting the node itself if it is a leaf
  (b) replacing it with its child if it has only one child
  (c) replacing it with its inorder successor and
    deleting the inorder successor

(2) Adjust the balance of the parent of the deleted node

(3) Adjust the balance of the parent of the deleted inorder successor

Examples (figure 7.6 on page 229)

(1) simple deletion of a leaf

(2) deletion of a node with only one child

(3) deletion of a node, which is replaced with its inorder successor

In the worst case, a rotation is needed.

7.2 2-3 trees and B-trees

2-3 trees

Goal: to construct trees of "perfectly balanced" – all leaves have the same depth and yet contain any number of leaves.

Definition: In a 2-3 tree:

(1) All leaves are the same depth and contain 1 or 2 keys

(2) An internal node either
  (a) contains one key and has two children (a *2-node*) or
  (b) contains two keys and has three children (a *3-node*)

(3) A key in an internal node is between the keys
  in the subtrees rooted at its adjacent children.

Examples (Figure 7.7)

Among all *2-3 trees* of height h:

(1) The one having all internal nodes containing one key and two children is of the fewest nodes and number of keys $n = 2^{h+1} - 1$. Thus $h = \lfloor \log_2 n \rfloor$.

(2) The one having all internals nodes containing two keys and three children is of the most nodes

$$\sum_{i=0}^{h} 3^i = (3^{h+1} - 1)/2$$

and the number of keys $n = 2 \times (3^{h+1} - 1)/2 = 3^{h+1} - 1$
so $h = \lfloor \log_3 n \rfloor$.

Insertion (by taking advantage of the “extra room” in a leaf)

(1) search for the leaf where the key belongs. Memorize the path.

(2) If there is a room, add the key and stop.

(3) Otherwise, split the 3-node into two 2-nodes and
pass the middle key to its parent.

(4) If the parent is a 2-node, it then becomes a 3-node and
the algorithm stops. Otherwise, go to step (3).

(5) The above is repeated up the tree until there is a room for
the new key or the root must be split.
In the latter case, the height of the tree is incremented.

Examples (Figure 7.8)

Deletion

(1) If the key to be deleted is in a leaf, remove it from the leaf.

(2) If the key is not in a leaf, then the key's inorder successor is in a leaf (*why*?). Replace the key with its inorder successor, and remove the key of the inorder successor.

(3) Let N be the node where a key has been removed. If node N still has one key, stop. Otherwise (it contains no key):

   (a) If N is the root, delete it. If it has a child, the child becomes the root.

   (b) (N has at least one sibling). If N has sibling N' that is immediately to its left or right and has two keys, then move the key of N's parent to N and replace it with the key of N' that is adjacent to N. If N and N' are internal nodes, move one child of N' to be a child of N. Stop.

   (c) (Now N' is a sibling of N and has only one child)
   Let P be the parent of N and N' and S be the key in P that separates them. Consolidate S and the one key in N' into a new 3-node, replacing N and N'. Let $N \leftarrow P$, repeat step (2).

Examples (Figure 7.9)

Red-black trees

A *red-black tree* is a binary search tree where:

(1) nodes are of red and black colors
(2) edges are of red and black colors
(3) the root is always black
(4) the color of any edge connecting a parent to a child is the same as the color of the child.

with additional constraints:

(5) On any path from the root to a leaf, the number of black edges is the same
(6) A red node that is not a leaf has two black children
(7) A black node that is not a leaf has either
   (a) two black children; or
   (b) one red child and one black child; or
   (c) a single child which is a red leaf.

Representing 2-3 trees with red-black trees which provide a straightforward implementation for 2-3 trees

Figure 7.10 (page 234):

a 3-node corresponds to two possible red-black tree

Figure 7.11

a red-black and the corresponding 2-3 tree

questions:

* what does a red node stand for?
* what does a black node stand for?
* what does a red edge stand for?
* what does a black edge stand for?
* why (5) is true?
* how about (6) and (7)?
* the height of a red-black tree is at most twice the height of the corresponding 2-3 tree (why?)

LookUp: is the ordinary binary tree search, ignoring colors

Insertion:

(1) Search the tree, insert the key as a new child of an old leaf

(2) Color the new node red

  (a) The parent is black
    (i) The other child is black/empty (i.e., a 3-node is formed)
    (ii) The other child is red: recoloring the children to black and the parent to red, check for the parent (i.e., a split)

  (b) The parent is red
    The grandparent is black, transform by a single rotation or a double rotation into case (a)(ii).

Figure 7.12 (page 236)

(a, b)-trees and B-trees

Extending 2-3 trees to (a, b)-trees

Let $a \geq 2$ and $b \geq 2a - 1$. An (a, b)-tree is a tree
where each node is either a leaf or has c children, $a \leq c \leq b$, and
all leaves have the same depth.

A (2, 3)-tree is a little different from 2-3 tree