RB-Tree Augmentation

- Augment x with **Size(x)**, where
  - Size(x) = size of subtree rooted at x
  - Size(NIL) = 0

OS-Rank

```
OS-RANK(x,y)
// Different from text (recursive version)
// Find the rank of x in the subtree rooted at y
r = size[left[y]] + 1
if x = y then return r
else if ( key[x] < key[y] ) then
     return OS-RANK(x,left[y])
else return r + OS-RANK(x,right[y] )
```

Time Complexity O(log n)

OS-Select

```
OS-SELECT(x,i) //page 304
// Select the node with rank i
// in the subtree rooted at x
r = size[left[x]]+1
if i = r then
       return x
elseif  i < r then
       return OS-SELECT (left[x], i)
else   return OS-SELECT (right[x], i-r)
```

Time Complexity O(log n)

Augmented Data Structures

- Why is it needed?
  - Because basic data structures not enough for all operations
  - storing extra information helps execute special operations more efficiently.
- Can any data structure be augmented?
  - Yes. Any data structure can be augmented.
- Can a data structure be augmented with any additional information?
  - Theoretically, yes.
- How to choose which additional information to store.
  - Only if we can maintain the additional information efficiently under all operations. That means, with additional information, we need to perform old and new operations efficiently maintain the additional information efficiently.

How to augment data structures

1. choose an underlying data structure
2. determine additional information to be maintained in the underlying data structure,
3. develop new operations,
4. verify that the additional information can be maintained for the modifying operations on the underlying data structure.

Augmenting RB-Trees

Theorem 14.1, page 309

Let f be a field that augments a red-black tree T with n nodes, and f(x) can be computed using only the information in nodes x, left[x], and right[x], including f[left[x]] and f[right[x]].

Then, we can maintain f(x) during insertion and deletion without asymptotically affecting the O(lgn) performance of these operations.

For example,

size[x] = size[left[x]] + size[right[x]] + 1

rank[x] = ?

Examples of augmenting information for RB-Trees

- Parent
- Height
- Any associative function on all previous values or all succeeding values.
- Next
- Previous

Greedy Algorithms

- Given a set of activities (s_i, f_i), we want to schedule the maximum number of non-overlapping activities.
- GREEDY-ACTIVITY-SELECTOR (s, f)

```
1. n = length[s]
2. S = {a1}
3. i = 1
4. for m = 2 to n do
5.     if s_m is not before f_i then
6.         S = S U {a_m}
7.         i = m
8. return S
```


Example

- [1,4], [3,5], [0,6], [5,7], [3,8], [5,9], [6,10], [8,11], [8,12], [2,13], [12,14] -- Sorted by finish times
- [1,4], [3,5], [0,6], [5,7], [3,8], [5,9], [6,10], [8,11], [8,12], [2,13], [12,14]
- [1,4], [3,5], [0,6], [5,7], [3,8], [5,9], [6,10], [8,11], [8,12], [2,13], [12,14]
- [1,4], [3,5], [0,6], [5,7], [3,8], [5,9], [6,10], [8,11], [8,12], [2,13], [12,14]
- [1,4], [3,5], [0,6], [5,7], [3,8], [5,9], [6,10], [8,11], [8,12], [2,13], [12,14]
- [1,4], [3,5], [0,6], [5,7], [3,8], [5,9], [6,10], [8,11], [8,12], [2,13], [12,14]

Why does it work?

- THEOREM
  Let A be a set of activities and let a_1 be the activity with the earliest finish time. Then activity a_1 is in some maximum-sized subset of non-overlapping activities.
- PROOF
  Let S' be a solution that does not contain a_1. Let a'_1 be the activity with the earliest finish time in S'. Then replacing a'_1 by a_1 gives a solution S of the same size.
  Why are we allowed to replace? Why is it of the same size?

Then apply induction! How?

Greedy Algorithms – Huffman Coding

- Huffman Coding Problem
  Example: Release 29.1 of 15-Feb-2005 of TrEMBL Protein Database contains **1,614,107** sequence entries, comprising **505,947,503** amino acids. There are 20 possible amino acids. What is the minimum number of bits to store the compressed database?
  ~2.5 G bits or 300MB.
- How to improve this?
- Information: Frequencies are not the same.

Ala (A) 7.72	Gln (Q) 3.91	Leu (L) 9.56	Ser (S) 6.98
Arg (R) 5.24	Glu (E) 6.54	Lys (K) 5.96	Thr (T) 5.52
Asn (N) 4.28	Gly (G) 6.90	Met (M) 2.36	Trp (W) 1.18
Asp (D) 5.28	His (H) 2.26	Phe (F) 4.06	Tyr (Y) 3.13
Cys (C) 1.60	Ile (I) 5.88	Pro (P) 4.87	Val (V) 6.66

Huffman Coding

2 million characters in file.
A, C, G, T, N, Y, R, S, M

IDEA 1: Use ASCII Code
Each need at least 8 bits,
Total = 16 M bits = 2 MB

IDEA 2: Use 4-bit Codes
Each need at least 4 bits,
Total = 8 M bits = 1 MB

Percentage Frequencies

IDEA 3: Use Variable Length Codes

A	22	11
T	22	10
C	18	011
G	18	010
N	10	001
Y	5	00011
R	4	00010
S	4	00001
M	3	00000

How to Decode?
Need Unique decoding!
Easy for Ideas 1 & 2.
What about Idea 3?

110101101110010001100000000110

110101101110010001100000000110

2 million characters in file.
Length = ?
Expected length = ?
Sum up products of frequency times the code length, i.e.,
(.22x2 + .22x2 + .18x3 + .18x3 + .10x3 + .05x5 + .04x5 + .04x5 + .03x5) x 2 M bits =
3.24 M bits = .4 MB

Huffman Coding

- **Idea:** Use shorter codes for more frequent amino acids and longer codes for less frequent ones.

Greedy Algorithms – Other examples

- Minimum Spanning Trees (Kruskal's & Prim's)
- Matroid Problems
- Several scheduling problems

Dynamic Programming

- Activity Problem Revisited: Given a set of activities (s_i, f_i), we want to schedule the maximum number of non-overlapping activities.
- New Approach:

  A_i = Best solution for intervals $\{a_1, \ldots, a_i\}$ that includes interval a_i

  B_i = Best solution for intervals $\{a_1, \ldots, a_i\}$ that does not include interval a_i
- Does it solve the problem to compute A_i and B_i?
- How to compute A_i and B_i?