4 Dictionaries

Readings: CLRS chapters 11, 12, 13.2, 17.4

Not covered in textbook: AVL trees (see Problem 13-3 in CLRS), scapegoat trees, tries (see Problem 12-2 in CLRS), PATRICIA trees

Dictionary is an abstract data type that stores pairs (key,value). All keys must be distinct (i.e., no two elements have the same key). Dictionaries support the following operations:

- Insert(key,val): inserts pair (key,val) into the dictionary.
- Search(key): search for an element (key,val) in the dictionary and return val, if such element exists, or NULL otherwise.
- Delete(key): deletes element (key,val) from the dictionary, if such element exists.

Other names: dict (in Python), relation (in database context), map (in C++ standard template library and Java API), hash (Perl), associative array

Trivial implementation 1: linked lists

- Insert - $\Theta(1)$
- Search (succesful) - $\Theta(n)$
- Search (unsuccesful) - $\Theta(n)$
- Delete - $\Theta(n)$ (this is because we need to find the element first before removing it)

Trivial implementation 2: sorted array

- Insert - $\Theta(n)$
- Search - $\Theta(\log n)$ (binary search)
- Delete - $\Theta(n)$

```
Search(key):
  from:=1; to:=n
  while (from<=to):
    i = (from+to)/2
    if a[i]==key return i
    else if a[i]<key from:=i+1
    else if a[i]>key to:=i-1;
  return NULL;
```

4.1 Hashing

Direct address hashing:

- Let keys are integers from interval $[0, U)$ (called *universe of keys*)
- We can create a table $T[0 \ldots U-1]$ (called *hash table*) which will store for each key either a value associated with this key in the dictionary, or NULL, if the key does not exist in the dictionary

- Dictionary operations:
  - Insert(key,val): store val in T[key]; run time $\Theta(1)$
  - Search(key): check T[key]; run time $\Theta(1)$
  - Delete(key): replace T[key] with NULL; run time $\Theta(1)$

Problem: Universe of keys can be huge! Requires too much memory and too much time to initialize the data structure.

Solution: Use *hash function* $h : [0, U) \to [0, m)$ that maps universe of keys $[0, U)$ into a smaller interval $[0, m)$. Then we can use hash table that has only m slots instead of U slots.

Problem: Different keys **will** map into the same slot of the table! Such a situation is called **conflict**.

Two solutions:

- Hashing with chaining
- Open address hashing

Hashing with chaining: Instead of storing a single value in $T[k]$, store a linked list of (key,value) pairs.

- Insert(key,val): add (key,val) to the list $T[h(\text{key})]$
- Search(key): search for key in the list $T[h(\text{key})]$; if the search is succesful, return corresponding value; otherwise return NULL
- Delete(key): search for key in the list $T[h(\text{key})]$ and delete the corresponding (key,val) pair from the list

Worst-case running time:

- Insert: $\Theta(1)$
- Search: $\Theta(n)$ (all items hash to the same slot)
- Delete: Search + $\Theta(1)$

Regardless, hashing is good in practice!

- if hashing function is good, the entries will be “scattered” evenly over the hash table
- if the hash table is large enough, lists will be short

Definition 1 (Load factor of hash table)**.** *A hash table with m slots storing n keys has load factor $\alpha = n/m$.*

Definition 2 (Uniform hash functions)**.** *Hash function is* uniform with respect to some process *generating keys in “Insert” operations, if the i-th key in such sequence hashes in each slot with equal probability.*

More formally, if $X_{i,j}$ is an indicator random variable for condition that the key in the i-th insert operation hashes to j-th slot, than:

$$\Pr(X_{i,j} = 1) = \frac{1}{m}$$

What is the expected size n_j of the list in slot j after n insert operations? Clearly, $n_j = \sum_{i=1}^{n} X_{i,j}$, and therefore, assuming that the hash function is uniform:

$$E[n_j] = E\left[\sum_{i=1}^{n} X_{i,j}\right] = \sum_{i=1}^{n} E[X_{i,j}] = \frac{n}{m} = \alpha$$

Thus *expected number of comparisons* required for various operations in hashing with chaining is as follows:

- unsuccesful search: $1+\alpha$ comparisons (one is added for the final comparison with NULL)
- succesful search: clearly at most $1+\alpha$ comparisons (the textbook CLRS also shows lower bound $\Omega(1+\alpha)$)

Choosing a good hash function Ideally, we would like uniform hash function.

Example 1: Assume that the keys for Insert operation are drawn uniformly randomly independently from $[0, U)$. Then the following function may be a good hash function:

$$h(k) = \left\lfloor k\frac{m}{U} \right\rfloor$$

Example 2: Assume that odd numbers are drawn twice as often as even numbers. Then we want to have function that allocates twice as much space in the table to odd numbers as to even numbers.

Achieving ideal uniform hashing may not be possible in practice.

Common hash functions:

- **Division method:** $h(k) = k \bmod m$

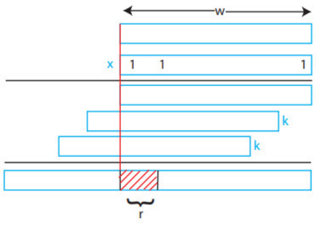
  Why $m = 2^p$ may not be the best choice?
  (only looking at last p bits)

  More difficult:Why $m = 2^{p-1}$ may not be the best choice?
  (in p-bit strings, rearrangement of characters does not change the hash value)

  Choose m that is a prime number not close to a power of 2 (or 10).

- **Multiplication method:** $h(k) = (ak \bmod 2^w) >> (w-r)$

  a is a random odd number s.t. $2^{w-1} < a < 2^w$

  (figure by Erik Demaine)

- **Universal hashing:**

  Example: $h(k) = (ak+b) \bmod p$

  p is a large prime ($> U$)
  a, b random numbers from $[0, p)$

Theorem 1. *With respect to a random choice of a, b, for any two keys $k_1 \neq k_2$*

$$\Pr[h(k_1) = h(k_2)] \leq 1/m.$$

This implies that number of collisions with key k is less than $n/m = \alpha$

See further discussion on advantages and disadvantages of particular choices of hash function in CLRS 11.3.

Open address hashing: If the conflict is encountered, and the slot in the hash table is already taken, try another slot. The hash function will now give the whole *sequence of slots* for one key, instead of a single value:

$$h : [0, U) \times [0, m) \to [0, m)$$

```
Insert(key,val):
  i:=0;
  while T[h(key,i)] is full
  | i:=i+1;
  T[h(key,i)]:=(key,val);

Search(key):
  i:=0;
  while T[h(key,i)] is full and T[h(key,i)].key != key
  | i:=i+1;
  return T[h(key,i)];
```

- Advantage: Simpler implementation, less memory (do not need to store as many pointers)
- Disadvantage: Operation Delete is really problematic

Typical hash functions for open address hashing

- **Linear probing:** $h(k, i) = (g(k) + i) \bmod m$
  ($g(k)$ is a regular hash function)

  Problem: *clustering*—long runs of occupied slots
- **Quadratic probing:** $h(k, i) = (g(k) + c_1 i + c_2 i^2) \bmod m$
  ($g(k)$ is a regular hash function, c_1 and c_2 are constants)

  Problem: *secondary clustering*—$g(k) = g(\ell) \Rightarrow h(k, i) = h(\ell, i)$
- **Double hashing:** $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$
  ($h_1(k)$ and $h_2(k)$ are regular hash functions)

In all hashing functions: we need to be careful for sequence $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ to cover the whole or at least major part of the hash table (i.e., elements in this sequence should not repeat).

More discussion on hashing functions for open address hashing can be found in CLRS 11.4.

Analysis of open adress hashing

Definition 3 (Uniform hashing functions for open address hashing)**.** *Hash function for open adress hashing is* uniform with respect to some process *generating keys in "Insert" operations, if sequence*

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$$

is always a permutation of numbers $\{0, 1, \ldots, m-1\}$, and every permutation is equally likely to occur with probability $1/m!$.

Note: Since each slot in the hash table holds at most one element, the load factor of hash table for open address hashing is always $\alpha \leq 1$.

Theorem 2. *Given an open address hash table with uniform hash function and load factor $\alpha < 1$, the expected number of comparisons required in Insert (or unsuccesful Search) operation is $1/(1-\alpha)$.*

Proof. (sloppy sketch) Let X denote the random variable representing the number of comparisons required for Insert operation. The uniformity condition implies that each slot in the hash table is occupied with probability $1/\alpha$.

$$\begin{aligned} E[X] &= 1.\Pr(X=1) + 2.\Pr(X=2) + 3.\Pr(X=3) + \ldots \\ &= \Pr(X \geq 1) + \Pr(X \geq 2) + \Pr(X \geq 3) + \ldots \\ &\approx 1 + \alpha + \alpha^2 + \ldots = 1/(1-\alpha) \end{aligned}$$

□

Theorem 3. *Given an open address hash table with uniform hash function and load factor $\alpha < 1$, the expected number of comparisons required in Search is at most $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$, assuming that* every key is equally likely to be searched.

Proof. The number of comparisons we have to do to find key k equals exactly to the number of comparisons that were needed to insert k into the dictionary. If the key k was inserted as $(i+1)$-st element, this number of comparisons is $1/(1-i/m) = \frac{m}{m-i}$.

Since every key is equally likely to be searched, we can get the expected number of comparisons by averaging over all keys that are in the dictionary:

$$\begin{aligned} \frac{1}{n}\sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1} \frac{1}{m-i} \quad [k := m-i] \\ &= \frac{1}{\alpha}\sum_{k=m-n+1}^{m} \frac{1}{k} \\ &\leq \frac{1}{\alpha}\int_{m-n}^{m} \frac{1}{k} dk \\ &= \frac{1}{\alpha}(\ln m - \ln(m-n)) \\ &= \frac{1}{\alpha} \ln \frac{m}{m-n} = \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned}$$

□

Size of the hash table and amortized analysis Ideally, we want $n \approx \alpha m$ most of the time.

Rehashing:

- change the size of the hash table m
- retrieve all the items from current hash table and re-instert them to the new hash table

Requires $\Theta(n)$ time.

How often to rehash?

- if $n > \alpha m$: double the size of m and rehash
- if $n < (1/4)\alpha m$: shrink the size of m to half and rehash

Amortized time complexity analysis: a common technique in developing and analyzing data structures.

- In each step, we have to pay for all work.
- In addition to the actual work performed, we can also choose to "save" something for later use (add to a balance). The cost of savings is added to the work performed.
- If we have enough balance, we can "withdraw" some of it and pay for part or all of the work.
  The cost of withdraval is subtracted from the work performed.

In our hashing example:

- for each Insert/Delete, 1 unit is spent on actual work and 2 units are added to the savings; thus each Insert/Delete will cost us 3 units instead of 1 (which is still a constant)
- when we are resizing from m to $2m$, we need αm units to cover the cost of resizing:
  - if this is the first resizing: we did at least αm Insert operations
  - if previous resizing was from $m/2$ to m: we did at least $(1/2)\alpha m$ Insert operations (after the last resize, the number of elements was $m/2$)
  - if previous resizing was from $2m$ to m: we did at least $(1/2)\alpha m$ Insert operations (after the last resize, the number of elements was $m/2$)

  So we can **always cover the full cost of rehash from the savings** ⇒ rehash is now effectively a free operation
- Similar analysis can be done for resizing from m to $m/2$.

We say that Insert and Delete have $\boxed{\Theta(1) \text{ amortized complexity}}$.

Note: Remember python resizable lists, they work in exactly the same way.

4.2 Binary Search Trees

Definition 4 (Binary search tree)**.** *A* binary search tree *is a binary tree, where for each node x of the tree:*

- *all elements in its left subtree are* smaller than *x,*
- *all elements in its right subtree are* larger than *x*

Simple implementation: Each node of the tree is represented as a record with following parts: **key**, **value**, **left** (pointer to the node representing left subtree), **right** (pointer to the node representing right subtree), **parent** (pointer to the node representing the immediate ancestor of the node, or NULL if the node is a root).

Dictionary operations:

- **Search:**

  ```
  Search(key,root):
    if root = NULL or root.key = key
    | return root

    if key < root.key
    | return Search(key,root.left)
    else
    | return Search(key,root.right)
  ```

- **Insert:**

  ```
  Insert(key,val,root):
    if root = NULL
    | x = new node(key,val)
    | return x

    if key < root.key
    | root.left = Insert(key,val,root.left)
    | root.left.parent = root
    else
    | root.right = Insert(key,val,root.right)
    | root.right.parent = root

    return root
  ```

- **Delete:** Little bit more complicated, but straightforward; refer to CLRS 12.3. (In the rest of this and the following section, we look only at Search and Insert; everything can be straightforwardly extended to Delete.)

Running time:

- Search: $\Theta(\textit{height})$
- Insert: $\Theta(\textit{height})$
- Delete: $\Theta(\textit{height})$

where *height* is the height of the binary search tree.

How large is height?

- **Ideally:** $\Theta(\log n)$
  (this is, in fact, expected height if keys are inserted in a random order—see CLRS 12.4)
- **Worst-case:** $\Theta(n)$
  (for example, insert elements in a sorted order)

4.3 AVL trees

[Adel'son–Vel'skii and Landis, 1962]

Goal: modify binary search trees so that the height is alway worst-case $\Theta(\log n)$

Definition 5 (AVL trees)**.** *A node x in a tree is* balanced *if the difference in height of its left and right subtrees is at most 1.*

A binary search tree is an AVL tree *if and only if every node is balanced.*

Note: Fibonacci numbers F_k are defined recursively as follows:

- $F_1 = 1$, $F_2 = 1$
- $F_{k+2} = F_{k+1} + F_k$

It was shown before (and it can be easily proven by induction), that

$$F_k = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right)$$

Note, that $\left| \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^k \right|$ is always smaller than 0.5; therefore this part of the formula only serves as a "rounding" factor, and therefore we can write:

$$F_k = \Theta \left(\left(\frac{1+\sqrt{5}}{2} \right)^k \right)$$

(i.e., F_k grows exponentially with k.)

Theorem 4 (Height of AVL trees)**.** *An AVL tree with n nodes has height $O(\log n)$,*

Proof. Let $S(h)$ be the smallest number of nodes in AVL tree of height h. We will show that $S(h) = F_{h+3} - 1$ by induction on h.

- **Base case:** $S(0) = 1 = F_3 - 1$, $S(1) = 2 = F_4 - 1$
- **Induction step:** Assume for all $i < h$, the claim is true. The AVL tree of height h must have one of its subtrees of height $h-1$, and the other subtree must be of height at least $h-2$. Therefore,

$$S(h) = S(h-1) + S(h-2) + 1 = (F_{h+2} - 1) + (F_{h+1} - 1) + 1 = F_{h+3} - 1$$

Therefore, the number of nodes of an AVL tree with height h is at least $\Theta \left(\left(\frac{1+\sqrt{5}}{2} \right)^i \right)$, and thus the height of an AVL tree with n nodes is at most $\Theta(\log n)$. □

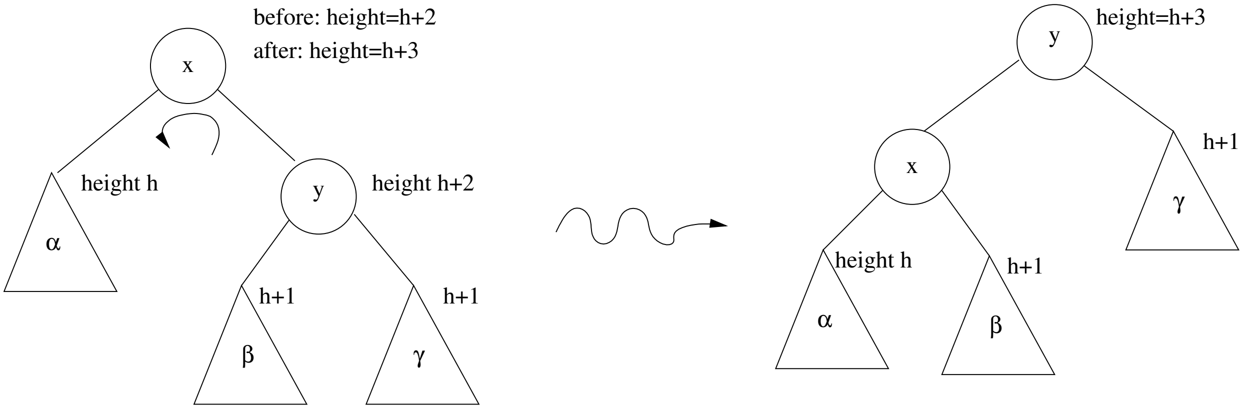
Inserting into AVL trees To facilitate "checking" the balance of nodes, we will keep additional entry for height of a subtree in each node of the tree. The code for Insertion will change as follows (the added lines are marked ***):

```
  AVL_Insert(key,val,root):
    if root = NULL
    | x = new node(key,val)
*** | x.height = 0
    | return x

    if key < root.key
    | root.left = Insert(key,val,root.left)
    | root.left.parent = root
    else
    | root.right = Insert(key,val,root.right)
    | root.right.parent = root

*** root.height = 1+max(root.left.height,root.right.height)
*** return Rebalance(root)
```

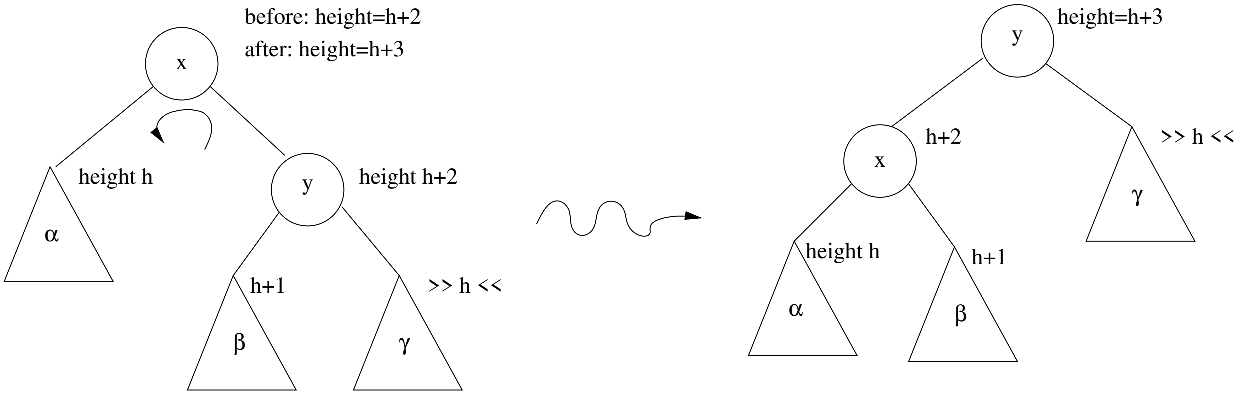
How to rebalance after insertion?

- Unbalance nodes can appear only on the path from root to the latest inserted element.
- Because we assumed that before the insertion, the tree was a valid AVL tree, and the height of each subtree may have increased by at most 1, the difference of the height will be at most 2.

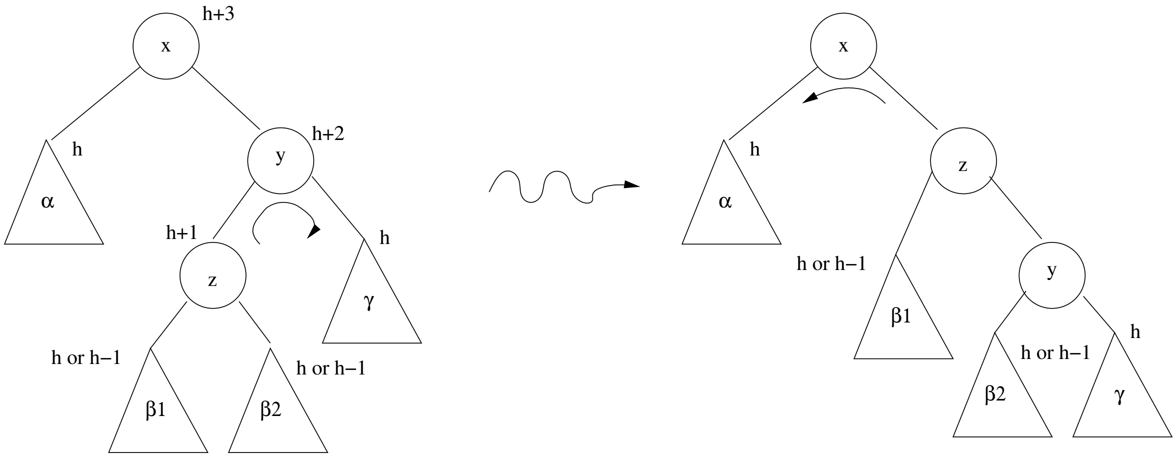
LeftRotation: Consider an unbalanced node x (the left tree is too small) as in the figure below. We can rebalance it by applying the left rotation as follows.

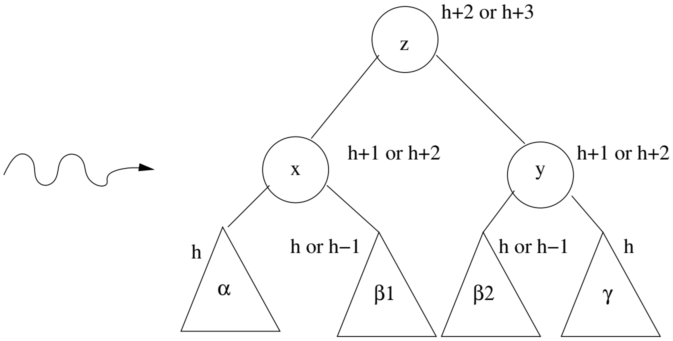


Similarly, we can have the right rotation, if the right subtree is too small. However, this technique is not universal; the following example results in ubalanced node y, because subtree γ is too small:



DoubleLeftRotation: To solve the case in the above figure, we need to apply double rotation as follows:





Similarly, we can design DoubleRightRotation for the symmetric case.

Pseudocode for rebalancing:

```
Rebalance(x):
  if x.left.height = x.right.height - 2
  | y = x.right
  | if y.right.height = y.height - 2
  | | y = RightRotate(y); y.parent = x; x.right = y
  | return LeftRotate(x)

  else if x.right.height = x.left.height - 2
  | y = x.left
  | if y.left.height = y.height - 2
  | | y = LeftRotate(y); y.parent = x; x.left = y
  | return RightRotate(x)

  else return x
```

4.4 Scapegoat trees

Scapegoat tree idea:

- We will keep inserting into the tree without rebalancing, until the tree height grows “too much”
- When things get too bad, we will find a node where things are very unbalanced (*scapegoat* node)
- We will rebuild scapegoat node subtree from scratch and everybody will be happy again
- Rely on amortized analysis to pay for the expensive rebuilds

```
Rebuild(x):
   A=empty array
   inorder(x,A)
   return buildBalanced(A,1,size_of(A))

inorder(x,A):
   if x = NULL return
   inorder(x.left,A)
   push(A,(x.key,x.val))
   inorder(x.right,A)

buildBalanced(A,from,to):
   mid = (from+to)/2
   root = new node(A[mid].key,A[mid].val)
   root.size = 1
   if (from<mid)
     // build left subtree
     left = buildBalanced(A,from,mid-1)
     root.left = left
     left.parent = root
     root.size += left.size
   if (to>mid)
     // build right subtree
     right = buildBalanced(A,mid+1,to)
     root.right = right
     right.parent = root
     root.size += right.size
   return root
```

Technical things:

- Will need to maintain size of all subtrees
- Will need to know where the node was inserted and at what depth

```
Insert(key,val,root):
  if root = NULL
    x = new node(key,val)
**  x.size = 1
**  return (x,x,0)

  if key < root.key
**  (root.left,x,depth) = Insert(key,val,root.left)
    root.left.parent = root
**  root.size += 1
```

```
      else
    **  (root.right,x,depth) = Insert(key,val,root.right)
        root.right.parent = root
    **  root.size += 1

    **return (root,x,depth+1)
```

- Will maintain maxDepth = $\log_{3/2} n$
  (note: "ideal" depth is $\log_2 n$)
- The *scapegoat node* will be a node whose subtree is so unbalanced that one of its children has **more than** 2/3 **of the nodes**
  (note: ideally, the nodes of a subtree are split evenly among the left and right child)

Insert with rebuild trigger:

```
Scapegoat_Insert(key,val,root):
  (root,x,depth) = Insert(key,val,root)
  if depth>maxDepth(root.size)
     scapegoat = findScapegoat(x)
     parent = scapegoat.parent
     y = Rebuild(scapegoat)
     if (parent = NULL)
        return y
     else
       if (parent.left = scapegoat)
          parent.left = y
       else
          parent.right = y
       y.parent = parent
   return root

findScapegoat(x):
  if (x.left and x.left.size>(2/3)*x.size)
    return x
  if (x.right and x.right.size>(2/3)*x.size)
    return x
  return findScapegoat(x.parent)
```

Theorem 5 (Existence of a scapegoat)**.** *If a node x is inserted at a depth greater than $\log_{3/2} n$, then there must always be a scapegoat on the path from root to x*

Proof. Assume there is no scapegoat on the path from root to x. This means that the size of the subtree in depth i on this path is at most $(2/3)^i n$. In particular, the size s of the subtree corresponding to the depth at which node x was inserted is at most:

$$s < n(2/3)^{\log_{3/2} n} = \frac{n}{(3/2)^{\log_{3/2} n}} = \frac{n}{n} = 1$$

So the size of the freshly inserted node x would have to be smaller than 1, which is a contradictiction. □

Amortized time analysis:

- We will associate a savings account with each of the nodes in the tree.
- Each insert operation will contribute 3 tokens to each node on the way to the place where the new node was inserted.
- The node rebuild operation will consume number of tokens equal to the size of the corresponding subtree and will take tokens from the account of the corresponding node.

Lemma 1 (Local account balance)**.** *The local account of node x at each point has at least* $3(|x.left.size - x.right.size| - 1)$ *tokens.*

Since local rebuild only happens when the difference between subtrees is greater than $(1/3)x$.size, we have always enough tokens to do the local rebuild.

Running time:

- Insert - $\Theta(n)$ worst-case, $\Theta(\log n)$ amortized
- Seach - $\Theta(\log n)$ worst-case
- Delete - $\Theta(n)$ worst-case, $\Theta(\log n)$ amortized

4.5 Dictionaries on words

We can apply all the methods from above chapters to words as well, however, the comparison of words cannot be considered a constant time operation. In particular, if the two words involved in comparison have lengths k and ℓ, then the comparison takes $O(\min(k, \ell))$ time.

For example, if we use AVL trees, this means the following worst-case running times (let us assume that all words are of length at most k):

- Insert: $\Theta(k \log n)$
- Search: $\Theta(k \log n)$
- Delete: $\Theta(k \log n)$

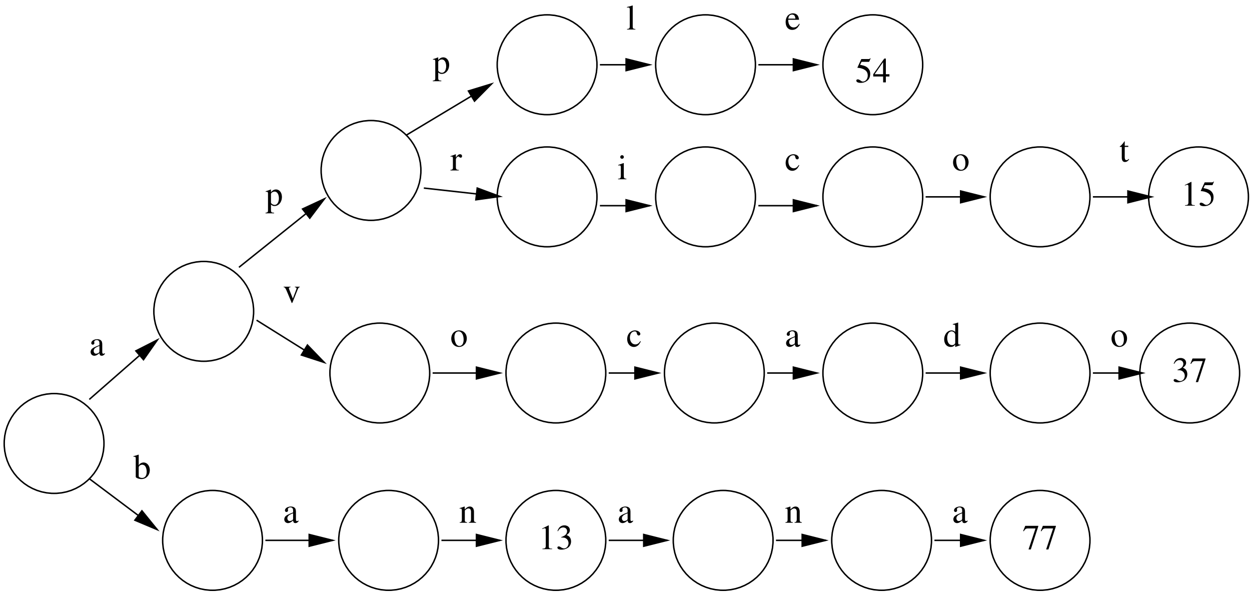
Perhaps we can do better, if we develop special dictionaries for words.

$\boxed{\textbf{Tries}}$ (trees for re**trie**val)

Assumption: words are from a fixed (and relatively small) alphabet Σ (for example, $\Sigma = \{a, b, \ldots, z\}$)

- tree (but not a binary one)
- each node has at most one child per symbol of alphabet
- edges are labeled by characters from the alphabet
- every node corresponds to a prefix of a word from the dictionary formed by concatenation of characters on the path from root to that node; the root of the tree corresponds to the empty word
- some nodes (those that correspond to the full words) have values associated with them

Example: Insert(banana,77); Insert(apple,54); Insert(avocado,37); Insert(apricot,15); Insert(ban,13)



Implementation: Every node is a record with **value** and an **array of pointers** to other nodes indexed directly by the symbols of alphabet.

```
Search(key):
  node := root; char := 0;
  while node <> NULL and char<length(key)
    node := node[key[char]];
    char++;
  if node = NULL return NULL
  else return node.value;


Insert(key,val):
  node := root; char := 0;
  while char<length(key)
    if node[key[char]] = NULL
      node[key[char]] = new node
    node := node[key[char]];
    char++;

  node.value := val;


Delete(key,val):
  node := root;

  while node <> NULL and char<length(key)
    node := node[key[char]];
    char++;
```

```
if node = NULL return NULL
else node.value = NULL;
```

Note: The version of delete here is a very simple version. To keep the size of the trie small, we would also delete all the nodes up the way we came until we encountered a node that had at least two outgoing edges.

Running times: If k is the length of the key, then the running time of the corresponding operations is:

- Insert: $\Theta(k)$
- Search: $\Theta(k)$
- Delete: $\Theta(k)$

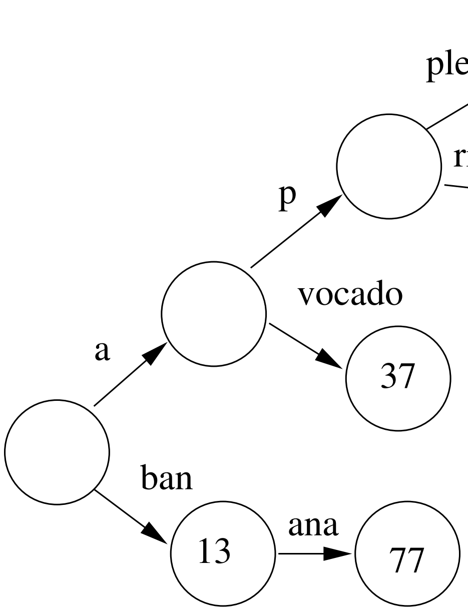
Disadvantages of tries:

- Usually there are many nodes, and only few of them store a value.
- Each node has many pointers (dependent on the size of the alphabet).
- Thus it is not memory efficient data structure.
- However, it is fast.

Compressed tries (or PATRICIA trees) from Practical Algorithm to Retrieve Information Coded in Alphanumeric

- To save space, we will have edges that will represent more than one character of a string: **compressed edges**.
- However, we will still have at most one outgoing edge for every letter of alphabet.
- We will "omit" nodes that have only a single outgoing edge and no value.

Example:



Implementation: For each character in the array of outgoing edges, we also store **word** (pointer to a string), **from**, and **to**, where word[from..to] is a corresponding string that is associated with the edge. Of course, for the edge indexed under character c, word[from] $= c$.

Implementation details: Operations Insert, Search, and Delete can be still done all in $O(k)$ time. They are little bit more complicated than pseudocodes listed above for uncompressed tries.

Details can be found in the paper: Donald R. Morrison, PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric, Journal of the ACM, 15(4):514-534, October 1968. `http://doi.acm.org/10.1145/321479.321481`

4.6 Summary

Method	Insert	Search	Delete	Analysis	Note
Unsorted linked list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	worst-case	
Sorted arrays	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$		
Hashing with chaining	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	worst-case	
(load factor α)	$\Theta(1)$	$\Theta(1+\alpha)$	$\Theta(1+\alpha)$	expected	uniform hashing
Open-address hashing	$\Theta(n)$	$\Theta(n)$	N/A	worst-case	
(load factor $\alpha < 1$)	$\Theta(\frac{1}{1-\alpha})$	$\Theta(\frac{1}{\alpha}\ln\frac{1}{1-\alpha})$ or $\Theta(\frac{1}{1-\alpha})$	N/A	expected	uniform hashing and equiprobable key search
Binary-search trees	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	worst-case	
	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	average	
AVL trees	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	worst-case	
Scapegoat trees	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	amortized	
Tries (string of length k)	$\Theta(k)$	$\Theta(k)$	$\Theta(k)$	worst-case	when using other data structures for strings, multiplicative factor of k must be added to the above times