Maths 3467, Michaelmas 2013: algorithms

Colm Ó Dúnlaing

March 27, 2014

Contents

1 Binary search

(1.1) The study of algorithms aims to make us better programmers in various ways.

- **Standardisation.** We learn standard methods applicable to well-known programming problems; this reduces the annoying 'problem-solving' component of programming.
- **Correctness.** Those methods which are not obvious are studied and explained.
- **Efficiency.** We evaluate the efficiency of the methods in terms of resources such as time consumed or memory space requirements.

These algorithms are usually non-numerical, meaning that exact results are easily defined and there is no notion of approximate solution.

In numerical programming accuracy is critical; in these lectures, efficiency is desirable but the level of efficiency is not usually of critical importance. We are content with 'order-of-magnitude' analysis, and hence the $O()$ notation is important.

(1.2) Definition *Let $f(n)$ and $g(n)$ be two functions defined on $\mathbb{N}$ such that*

$$0 \leq f(n), g(n) \in \mathbb{R}$$

for all n. In other words, they are nonnegative real-valued sequences. Informally,

$$f \text{ is } O(g) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

More formally,

$$f \text{ is } O(g) \iff (\exists C \geq 0)(\exists N \in \mathbb{N})(\forall n \geq N)(f(n) \leq Cg(n))$$

Roughly speaking, the $O()$ notation measures the *growth rate* of sequences.

(1.3) Searching. Frequently a program needs to search an array of keys to find the location of a particular key. The obvious way is to iterate through the array with a for-loop; if the array has size n then this will take n iterations, or time proportional to n: written $O(n)$. This is called *linear search.*

If the keys can be sorted, and are stored in the array in sorted order, then **binary search** allows a key to be located in time $O(\log n)$.

(1.4) The idea is simple: the range of possible locations of a given key is halved in each iteration, so after about $\log_2 n$ iterations either the key is found or the range is reduced to nothing.

```
int location ( int *a, int n, int key )
{
  int i,j,m;

  i = 0;
  j = n-1;

  while ( i <= j )
  {
    m = (i+j)/2;
    if ( a[m] == key )
      return m;
    else if ( a[m] > key )
      j = m-1;
    else
      i = m+1;
  }
  return -1; /* nothing found */
}
```

(1.5) Example.

0	1	2	3	4	5	6	7	8	9	i
-1	0	1	3	5	7	8	10	11	13	a[i]

```
search for 11:          Search for 12:
i   j   m   a[m]        i    j    m    a[m]
0   9   4     5         0    9    4      5
5   9   7    10         5    9    7     10
8   9   8    11         8    9    8     11
return 8                9    9    9     13
                        9    8
                        return -1
```

Correctness. The while-loop **preserves** the following **invariant condition**: *if* the key is stored in the array *then* its index is $\geq i$ and $\leq j$. As for **efficiency,**

(1.6) Lemma *Binary search runs in time* $O(\log n)$.

Proof. We measure the search range, namely $j - i + 1$, at each iteration. In a single iteration, if `a[m]` $\neq$ `key`, the variables i, j are replaced by

$$i, m-1 \quad \text{or} \quad m+1, j \qquad \text{where } m = \lfloor \frac{i+j}{2} \rfloor.$$

Let s be the search range before the iteration, i.e., $j - i + 1$. We assume $s > 0$ and `a[m]` $\neq$ `key`: otherwise we are at the last iteration.

The interval $i \ldots m-1$ has length

$$m - i = \lfloor \frac{i+j}{2} \rfloor - i = \lfloor \frac{j-i}{2} \rfloor \leq \lfloor s/2 \rfloor,$$

and $m + 1 \ldots j$ has length

$$j - m = j - \lfloor \frac{i+j}{2} \rfloor = \lceil \frac{j-i}{2} \rceil \leq \lfloor s/2 \rfloor.$$

The range is reduced to at most $\lfloor s/2 \rfloor \leq s/2$ (possibly a fraction), so after r iterations the search range is at most

$$\frac{n}{2^r}$$

(possibly a fraction). If r is sufficiently large so that this fraction is < 1, there can be no further iterations. How large should r be?

We want $n/2^r < 1$, i.e., $n < 2^r$. Take $r = \lceil \log_2(n+1) \rceil$. Then $2^r \geq n + 1$. Therefore there are at most

$$\lceil \log_2(n+1) \rceil$$

iterations. Each iteration takes bounded time, so the overall runtime is $O(\log n)$. **Q.E.D.**

Here is a binary search procedure written in Eiffel.

```
location ( a: ARRAY[ <comparable type>  ]; key: <same type> )
         : INTEGER is
  require
    normal_bounds: a.lower = 1
                -- just an example
  do
    local
      i,j,m : INTEGER
      found : BOOLEAN
  do
    from
      i = a.lower
      j = a.upper
    until
      i > j or found
    loop
      m := (i+j) div 2
      if a[m] = key then
        found = true
      else if a[m] > key then
        j = m - 1
      else
        i = m + 1
    end

    if found then
      Result := m
    else
      Result = a.lower - 1
    end
  end
```

2 Binary trees

(2.1) Trees are very useful in Computer Science; the idea is based on something like family trees.

(2.2) Definition *A* forest *consists of a finite set of* nodes*, together with a 'parent' function defined on some of the nodes.*

It is required that the 'parent' function — write it as $p()$ for brevity — is acyclic. This means that no node u is a proper ancestor of itself,[1] *i.e., there is no node u such that*

$$u = p(p(p \ldots p(u) \ldots))$$

A root *in the forest is a node with no parent.*
A tree *is a forest which either is empty or has exactly one root.*
The children *of a node u are those nodes whose parent is u. A* leaf *is a node with no children.*

This section is concerned with *binary trees.* A binary tree has three partial functions, parent, lchild (left child) , and rchild (right child). Also,

- For every node v, the children of v are the left and right child, where they exist.
- If the left child u of u exists, and also the right child v, then they are distinct: $u \neq v$.

Example. A binary tree can expose the 'structure' of an arithmetic expression. See Figure 1, which shows a conventional way of depicting binary trees.

A typical definition of a binary-tree node in C would be

```
typedef struct btree_node_tag {
  char * item; /* whatever is 'stored' in a node,
                * in this case a character string
                */
  struct btree_node_tag
    *lchild, *rchild, *parent;
} BTREE_NODE;
```

And typically in Eiffel,

```
class BTREE_NODE [G]
feature {BTREE}
  item: G -- stored at the node,
          -- Type G (a parameter: generic)
  lchild, rchild, parent: like Current

  .....
end -- class
```

(2.3) Definition *The* ancestors *of a node u in a tree are either*

[1] 'proper ancestor' is explained later (2.3)

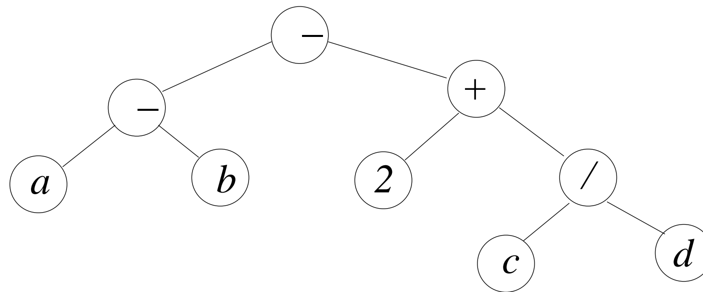


Figure 1: Binary tree representing $a - b - (2 + c/d)$.

- *u alone if u is a root, or otherwise*
- *u itself together with (recursively) the ancestors of its parent.*

The proper ancestors *of u are all ancestors except u itself.*
Similarly u has descendants *and* proper descendants.

Inorder, preorder, and postorder. There are three important linear orders defined recursively for binary trees.

- **Preorder:** root before left subtree before right subtree
- **Inorder:** left subtree before root before right subtree
- **Postorder:** left subtree before right subtree before root

If we take the above expression tree and list its nodes in the three orders we get preorder $--1b+2/cd$, inorder $a - b - 2 + c/d$, postorder $ab - 2cd/ + -$. For calculation purposes inorder is useless since the parentheses are missing, but preorder and postorder preserve the tree structure. Postorder, 'postfix notation,' is suitable for calculation using a pushdown stack. There is a very elegant algorithm to convert from (parenthesised) infix notation to postfix. It seems to be due to Dijkstra.

By the way, the Postscript language uses postfix notation. The above expression would be something like

```
a b sub 2 c d div add sub
```

in Postscript.

(2.4) Definition *The* depth *of a node u in a tree T is the number of proper ancestors of u, so the root has depth 0. The* height *of a node u is the maximum distance from u to a proper descendant of u.*

That is,

- The height of a leaf is 0;

- If u is not a leaf, then its height is $1+$ the maximum height of its left and right children, if they exist.

The *height* of a tree T is -1 if T is empty; otherwise it is the height of its root.

(2.5) Lemma (i) *The height of a nonempty tree T is the maximum depth of its nodes.*
(ii) *A binary tree of height h has between $h+1$ and $2^{h+1}-1$ nodes.*
(iii) *The height of a binary tree with n nodes is between* $\log_2(n+1)-1$ *and* $n-1$.

Proof. (i) We skip the proof: not because it is unimportant or obvious, but because it doesn't reward investigation.
(ii) If T is empty then $h=-1$, and $h+1=2^{h+1}-1=0$. Otherwise, T is nonempty.
For any $d \geq 0$, there are at most 2^d nodes at depth d: this is easily proved by induction on d, since every node at depth d has at most two children at depth $d+1$. Hence $n \leq \sum_{d=0}^{h} 2^d = 2^{h+1}-1$.
Choose a node of depth h (part (i)). It has $h+1$ ancestors including itself. Hence $h \leq n-1$.
(iii) is another version of (ii). **Q.E.D.**

(2.6) The Omega notation. When f is $O(g)$, f grows no faster than g. But f could grow slower than g; in that case we would write f is $o(g)$ ('little-o'). Informally,

$$f \text{ is } o(g) \iff \lim_n \frac{f(n)}{g(n} = 0.$$

The Ω notation is the *opposite* of this. It means that f does not grow slower than g. Informally

$$f \text{ is } \Omega(g) \iff \lim_n \frac{f(n)}{g(n} > 0.$$

(Possibly the limit is infinite.) With this notation, part (iii) of Lemma 2.5 can be simplified.

(2.7) Lemma *The height of an n-node binary tree is $O(n)$ and $\Omega(\log n)$.*

A partial strengthening will be useful:

(2.8) Lemma *Let T be a binary tree containing a subset S of k nodes. Then the deepest node in S has depth* $\Omega(\log k)$.

Proof. Let d be the maximum depth of all nodes in S. The total number of nodes of depth $\leq d$ in T is at most $2^{d+1}-1$, so

$$k \leq 2^{d+1}-1; \quad d \geq \log_2(k+1)-1,$$

so d is $\Omega(\log k)$. **Q.E.D.**

3 Enumerating binary trees

There are $\frac{1}{n+1}\binom{2n}{n}$ binary trees with n nodes. These are the *Catalan numbers.* There are at least two ways of arriving at this. One is given in Feller Volume 2.[2] The other uses generating functions.

Let b_n be the number of binary trees with n nodes. We allow the empty tree: $b_0 = 1$.

Given $n \geq 0$, by considering all possible sizes, i and j respectively, of left and right subtrees, in a tree with $n+1$ nodes, we arrive at the recurrence

$$b_{n+1} = \sum_{i+j=n} b_i b_j$$

Let $B(z) = \sum b_n z^n$. Then

$$B(z)^2 = \sum_{n\geq 0} \sum_{i+j=n} b_i b_j z^n$$

so the coefficient of z^n in $B(z)^2$ equals b_{n+1}, and

$$zB(z)^2 = B(z) - 1; \qquad zB(z)^2 - B(z) + 1 = 0.$$

Whence

$$B(z) = \frac{1 \pm \sqrt{1-4z}}{2z}$$

There are no negative powers of z in $B(z)$, which means we should take the minus sign.

$$B(z) = \frac{1-\sqrt{1-4z}}{2z} = \frac{1 - \sum_{r\geq 0}\binom{1/2}{r}(-4z)^r}{2z}.$$

The term in z^{-1} vanishes, and the others are simplified as follows.

$$\frac{1 - \sum_{r\geq 0}\binom{1/2}{r}(-4z)^4}{2z} = \sum_{r\geq 1}(-1)\binom{1/2}{r}(-4)^r \frac{1}{2} z^{r-1} = \sum_{r\geq 1}\binom{1/2}{r}(-1)^{r+1}2^{2r-1}z^{r-1} = \sum_{n\geq 0}\binom{1/2}{n+1}(-1)^n 2^{2n+1} z^n$$

[2] Feller, William: *An Introduction to Probability Theory and its applications.*

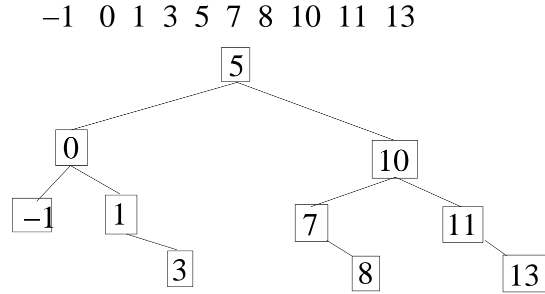


Figure 2: Binary search tree imitating binary search on array of a previous example.

so

$$b_n = \binom{1/2}{n+1}(-1)^n 2^{2n+1} =$$

$$(-1)^n \frac{\frac{1}{2}\left(\frac{1}{2}-1\right)\cdots\left(\frac{1}{2}-(n+1)+1\right)2^{2n+1}}{(n+1)!} =$$

$$\frac{\frac{1}{2}\left(1-\frac{1}{2}\right)\cdots\left(n-\frac{1}{2}\right)2^{2n+1}}{(n+1)!} =$$

$$\frac{(2n-1)(2n-3)\cdots(3)(1)(1)2^n}{(n+1)!} =$$

$$\frac{2n(2n-1)(2n-2)(2n-3)\cdots(3)(2)(1)}{n!(n+1)!} =$$

$$\frac{(2n)!}{n!(n+1)!} = \frac{1}{n+1}\binom{2n}{n}. \quad \blacksquare$$

4 Binary search trees

Binary search trees take their inspiration from binary search. While binary search is very efficient, it is inflexible; that is, there is no easy way to add or remove elements in the array being searched. Binary search trees aim to rival binary search in efficiency with greater flexibility.

(4.1) Binary search trees are binary trees whose nodes carry 'keys' of `COMPARABLE` type, arranged so that inorder traversal yields the keys in strictly sorted order.

Equivalently, at every node v, the keys stored in its left subtree are less than, and the keys stored in its right subtree are greater than, the key stored at v.

Figure 2 shows a binary search tree which corresponds exactly with binary search on the array given in §1.5.

(4.2) Inorder successor: non-recursive form. The inorder successor of p is,

- If p has a right child q, its inorder successor is q's leftmost descendant
- Otherwise its inorder successor is the lowest ancestor q of p, if it exists, such that p is descended from the left child of q.

(4.3) To locate a key in a binary search tree is straightforward:

- Initially, p is the root.
- Repeatedly, until p becomes NULL or the key is found, compare the key with `p->key`.
- If equal, return p.
- If low, replace p by `p->lchild` and continue.
- Otherwise, replace p by `p->rchild` and continue.

(4.4) Inserting a key is almost as simple. First follow the above procedure to locate a node containing the key — if the key is already stored, do nothing. Otherwise, suppose that u was the last non-null value p had. If the tree was empty, store the new key at the root. If p had been replaced by `p->lchild`, add the key as a left child for u. Else add it as a right child.

(4.5) The cost of searching for a key is proportional to its depth in the tree (or the depth of the last node inspected). Recall that the depth of a binary tree is $O(n)$ and $\Omega(\log n)$.

If the tree is shallow — meaning that its height is $O(\log n)$ — then searching the tree is maximally efficient.

If a binary search-tree is built beginning with the empty tree and making a sequence of insertions, it is possible that its depth is $n-1$. This happens, for instance, if the keys are presented for insertion in ascending order, in which case the tree resembles a linked list (there are no left children).

(4.6) The average cost of searching, however, is good, on average. This means that assuming that a tree is built by inserting a sequence of n keys in random order, then the average depth of a node in an average tree is proportional to $\log(n)$. This is proved as follows.

(4.7) First, let T be a binary search-tree with n nodes. The INTERNAL PATH-LENGTH (IPL) is the sum of the node depths. (Or, equivalently, the sum of lengths of all paths in the tree.) By convention, the root node has depth 0.

Thus the average node-depth, the average cost of accessing a node, assuming all nodes are accessed with equal likelihood, is $1/n$ times the IPL.

Next, let IPL(T) denote the internal path-length of a nonempty binary search-tree T; let T_ℓ and T_r be the left and right subtrees at the root, respectively; and suppose that i is the inorder rank of its root. Then

$$\text{IPL}(T) = i - 1 + \text{IPL}(T_1) + n - i + \text{IPL}(T_2),$$

because the depth of a node in T_1 or in T_2 is 1 less than its depth in T.

Let σ be an permutation of n input keys; we assume that all permutations have equal probability. A tree T built from keys presented in the order σ will have root with inorder rank i, where $i = \sigma(1)$.

If σ ranges over all permutations in which $\sigma(1) = i$, then the subtrees T_1 and T_2 will range independently over average search-trees with $i-1$ and $n-i$ nodes respectively.

Therefore

$$I = (n-1) + A(i-1) + A(n-i),$$

where I is the average IPL of trees created by all such σ, and $A(j)$ is the average IPL of a tree created from j random insertions. Averaging over all possible n values of i,

$$A(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{i=n}(A(i-1) + A(n-i)).$$

This recurrence is well-known and there are simple tricks for arriving at a solution. First, the two parts of the sum make the same contribution, so we get

$$\begin{aligned}
A(n) &= n - 1 + \frac{2}{n}\sum_{i=1}^{i=n} A(i-1) \\
A(n) &= n - 1 + \frac{2}{n}\sum_{i=0}^{i=n-1} A(i) \\
nA(n) &= n(n-1) + 2\sum_{i=0}^{i=n-1} A(i) \\
(n+1)A(n+1) &= n(n+1) + 2\sum_{i=0}^{i=n} A(i) \\
(n+1)A(n+1) - nA(n) &= 2n + 2A(n) \\
(n+1)A(n+1) - (n+2)A(n) &= 2n \\
\frac{A(n+1)}{n+2} - \frac{A(n)}{n+1} &= 2\frac{n}{(n+1)(n+2)}
\end{aligned}$$

Writing $U(n) = A(n)/(n+1)$ we get

$$\begin{aligned}
U(n+1) - U(n) = 2\frac{n}{(n+1)(n+2)} &= 2\frac{1}{n+1} - 4\frac{1}{(n+1)(n+2)} \\
U(n) &= 2\sum_{0}^{n-1}\frac{1}{n+1} - X.
\end{aligned}$$

The X part converges and can be ignored. Comparing the sum (called a harmonic series) with the integral of $1/x$, it emerges that $U(n) = 2\ln(n) + Y$ where Y is bounded. Hence $A(n)$ is roughly $2n \ln n$: it is $O(n \log n)$.

(4.8) Lemma *The average IPL of a binary search-tree built by a random sequence of key-insertions is $O(n \log(n))$ and hence the average cost of a successful search in such a tree is $O(\log(n))$.*

5 Average ipl of binary trees

This section uses generating functions to estimate the average depth of binary trees *as opposed to binary* **search** *trees.* The distributions are different.

The generating function for b_n, the number of binary trees with n nodes is

$$B(z) = \frac{1-\sqrt{1-4z}}{2z}.$$

Following Exercise 5 of group 2.3.4.5 in [Knuth, volume 1]. Let b_{np} be the number of trees with n nodes and IPL p.

Looking at the left and right subtrees at the root of one such tree, the left subtree has k nodes and IPL r, say, and the right subtree has ℓ nodes and IPL s, say; and $k+\ell+1=n$ and $r+s+n-1=p$. Therefore

$$b_{np} = \sum_{k+\ell=n-1;r+s+n-1=p} b_{kr}b_{\ell s}.$$

Let $B(w,z) = \sum_{n,p} b_{np}z^n w^p$. Then

$$zB(w,wz)^2 = B(w,z) - 1.$$

and $B(1,z) = B(z)$ as above.

In Knuth's notation, B_w is $\partial B/\partial w$, etcetera. Taking $\partial/\partial w$,

$$2zB(w,wz)(B_w(w,wz) + zB_z(w,wz)) = B_w(w,z).$$

Let $H(z) = \sum h_n z^n = B_w(1,z)$. Clearly the average ipl with n nodes is h_n/b_n.

$$\begin{aligned} 2zB(1,z)(H(z) + zH'(z)) &= H(z) \\ 2zB(z)(H(z) + zB'(z)) &= H(z) \\ H(z) &= \frac{2z^2BB'}{1-2zB} \end{aligned}$$

$$
\begin{gathered}
1-2zB=\sqrt{1-4z}\\
\frac{d}{dz}\sqrt{1-4z}=\frac{-2}{\sqrt{1-4z}}\\
B'(z)=\frac{-1}{2z^2}-\frac{1}{4z^2}\left[2z\times\frac{-2}{\sqrt{1-4z}}-\sqrt{1-4z}\times 2\right]=\\
\frac{-1}{2z^2}\left(1+\left(\frac{-2z}{\sqrt{1-4z}}-\sqrt{1-4z}\right)\right)\\
\sqrt{1-4z}H=\left(\frac{2z}{\sqrt{1-4z}}+\sqrt{1-4z}-1\right)\left(\frac{1-\sqrt{1-4z}}{2z}\right)\\
H=\left(\frac{2z}{1-4z}+1-\frac{1}{\sqrt{1-4z}}\right)\left(\frac{1-\sqrt{1-4z}}{2z}\right)=\\
\frac{1}{1-4z}+\frac{1}{2z}-\frac{1}{2z\sqrt{1-4z}}-\frac{1}{\sqrt{1-4z}}-\frac{\sqrt{1-4z}}{2z}+\frac{1}{2z}=\\
\frac{1}{1-4z}+\frac{1}{z}-\frac{1}{\sqrt{1-4z}}-\frac{1}{2z\sqrt{1-4z}}(1+1-4z)=\\
\frac{1}{1-4z}+\frac{1}{z}-\frac{1}{z\sqrt{1-4z}}-\frac{1}{\sqrt{1-4z}}+\frac{2}{\sqrt{1-4z}}=\\
\frac{1}{1-4z}+\frac{1}{z}-\frac{1}{z\sqrt{1-4z}}+\frac{1}{\sqrt{1-4z}}=\\
\frac{1}{1-4z}-\frac{1}{z}\left(\frac{1-z}{\sqrt{1-4z}}-1\right).
\end{gathered}
$$

or,

$$\frac{1}{1-4z}+\frac{1}{\sqrt{1-4z}}-\frac{1}{z\sqrt{1-4z}}+\frac{1}{z}.$$

Expand $1/\sqrt{1-4z}$.

$$\sum_{r\geq 0}\binom{-1/2}{r}(-4z)^r=\sum_r\binom{2r}{r}z^r.$$

The coefficient, call it h_n, of z^n, is

$$4^n+\binom{2n}{n}-\binom{2n+2}{n+1}=4^n-\frac{3n+1}{n+1}\binom{2n}{n}.$$

Divide by b_n, the n-th catalan number $\frac{1}{n+1}\binom{2n}{n}$, to get the average ipl of binary trees:

$$\frac{4^n}{\frac{1}{n+1}\binom{2n}{n}}-3n-1.$$

From Stirling's formula,

$$b_n\sim\frac{4^n}{(n+1)\sqrt{n\pi}}$$

so the dominant term in $A(n)$ is

$$n\sqrt{n\pi}$$

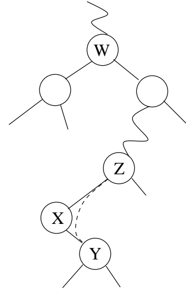


Figure 3: Hibbard's deletion strategy

and the average *depth* of average binary trees

$$\sim \sqrt{n\pi}.$$

6 Deletion from binary search trees

(6.1) To delete a node from u a binary search tree is easy if that node has fewer than two children. Suppose that w is its one child (default `Void`). If u is the root, w becomes the root; otherwise, u has parent p say; then p becomes the parent of w, and w becomes the left- or right- child of p, whichever u was.

If u has two children, the trick is to locate the inorder successor v of u, delete it from the tree (it has no left child), and re-attach it where u was in the tree. This is called *Hibbard's deletion strategy.* In Figure 3, if one wants to remove the node containing W, one actually replaces it by its inorder successor X. X is easily detached from the tree because it has no left child, and it can be moved to take W's place in the tree.

Lemma 4.8 does not hold when insertions and deletions are mixed together. Culberson (mid-80s)[3] showed that the average IPL of a tree subject to insertions and deletions (with Hibbard's strategy) could be $\Omega(n\sqrt{n})$.

Compare this with the average IPL of a binary tree.

There is a way to get guaranteed efficient insert and delete *on average*. This method is due to Michaela Heyer.[4]

To put it very briefly: together with a key at each node there is also a *timestamp,* giving the 'time' $(0 \ldots n-1)$ when the key was inserted. Of course, the keys and timestamps together can be used to construct the tree.

Insertion is basically the same as before. Deletion is more complicated. It is arranged to have the following property:

> Suppose a mixed sequence M of insertions and deletions is executed, ending up with a tree T. Let S be the subsequence consisting of those insertions which are not cancelled by a later deletion, and let T' be the tree which would be built from S.
>
> Then T and T' are isomorphic.

[3] Heyer gives different sources, referring to 'Knott's paradox.'

[4] Randomness preserving deletions on special binary search trees, *Electronic notes in theoretical computer science* **225**, 99–113.

The tree T' has good average depth; therefore so has T. (The reasoning may be quite subtle...)

7 Red-black trees

Here we introduce red-black trees, which are binary trees whose heights are guaranteed to be $O(\log n)$ The structure of such trees is subject to change, but the *inorder* sequence of nodes is preserved. They are suitable for storing data in a linear order (inorder). Usually, though not always, they are used as search-trees.

(7.1) In a binary tree, the *sibling* of a node is the other child of its parent, assuming parent and children exist. A root has no siblings.

(7.2) A *red-black* tree is a binary (search) tree satisfying

- It can be empty (no root).
- **Colour and rank:** every node is 'coloured' red or black. The *red-black rank* of a node is the maximum number of black nodes encountered in a branch from the node down (to a leaf); the node itself is counted if it is black.
- **No double red:** Every red node has a black parent.
- **Rank balance:** Siblings have equal rank, and a node without siblings, except the root, has rank zero.

(7.3) Lemma *The root, if it exists, is black, and any other node without siblings is a red leaf.*

Proof. Every red node has a black parent, so the root cannot be red.
Suppose v is a node without siblings. From rank balancing, its rank is zero, so all its descendants are red. By the 'no double red' condition, it has no proper descendants and is a red leaf. **Q.E.D.**

(7.4) Lemma *Let u be a node of rank r. Then u has at least $2^r - 1$ black descendants.*

Proof by induction on the height of u. If u is a red leaf then $r = 0$ and if it is a black leaf then $r = 1$; in either case the statement is true. If u has one child, then u is black and its child is a red leaf, and again $r = 1$ and the statement is true.
So we can assume u has two children v and w.
If u is red then both v and w have rank r and the statement is true by induction (in fact, strengthened).
If u is black then v and w have rank $r - 1$, with at least $2^{r-1} - 1$ black descendants each, by induction; adding u which is black, u has at least $2^r - 1$ black descendants. **Q.E.D.**

(7.5) Corollary *A red-black tree with n nodes has height $O(\log n)$.*

Proof. Suppose r is the red-black rank of its root.
The tree contains at least $2^r - 1$ black nodes, so $r \leq \log_2(n + 1)$.
Every red node has a black parent, which means that every path from root to a leaf contains at least as many black nodes as red. Therefore the tree has depth at most $2r - 1$, which is $O(\log n)$. **Q.E.D.**

(7.6) Keeping track of the ranks. The structure for a red-black tree should include the rank of its root. The rank of the root is enough: the rank of other nodes is easily deduced.

(7.7) An **imperfect** red-black tree is one which has a double red or a rank imbalance, violating exactly one of the two conditions exactly once, in the sense that there exists at most one red node whose parent is also red, or at most one node whose children are themselves rank-balanced but who differ in rank by 1. The *location* of a red-red imperfection is the red child of a red parent, and the location of a rank imbalance is at the heavier child. (This is in case the other child is non-existent.)

It is possible to convert an imperfect red-black tree into a red-black tree, while preserving inorder. The easier case is that of a double red imperfection.

(7.8) Rotating the links. An important operation here is that of 'rotating the links' at a node. Suppose that p is the left child of a node q. Let A and B be its left and right subtrees, and C the right subtree of q. So the subtree at q can be written $ApBqC$ in inorder. To rotate from p means to make q the right child of p, and B its left subtree. It is as if the pq link were rotated clockwise. This operation preserves inorder: the result would again be written $ApBqC$, though the parent-child relations are different.

Again, if q were the right child of p, the subtree at p being $ApBqC$, we can rotate from q, making q the parent: anticlockwise rotation.

(7.9) Fix double red. Suppose that a tree T satisfies the red-black conditions except for a possible double-red imperfection at p.

It is all right to make adjustments to the tree that lead to a new imperfection, so long as only one is introduced, and it is closer to the root. We avoid any action that might create an imperfection further down the tree. Bearing this in mind, the corrective strategy is as follows. (Note: as the corrections proceed up the tree, it is no longer certain that p have any parent, or a red parent.) See Figure 4.

- If p is the root, make it black, increase the rank (see 7.6), and stop. Otherwise, let q be its parent.
- If q is black, stop.
- If q is the root, make it black, increase the rank, and stop. Otherwise let r be its parent and s its sibling (s may be null initially, that is, q may have no sibling initially).
- If s is non-null and red, then 'promote:' make both s and q black, and make r red. The parent of p is no longer red, but there may be a double red at r: set $p := r$.
- Otherwise there are two cases, depending on whether the path pqr is 'zigzig' or 'zigzag.' In the zigzig case, p and q are both left children or both right children. Rotate from q, making q black and r red (this is all right since s is not red), and stop.
- In the zigzag case, rotate from p, then rotate from p again, making r red and p black, and stop.

(7.10) Fix rank deficit. This is, for some reason, more complicated. See Figure 5. The routine should begin with a node q whose rank is unbalanced, and p should be its 'deficient child.' It is possible that p be null, initially, but q should never become null. Iteratively,

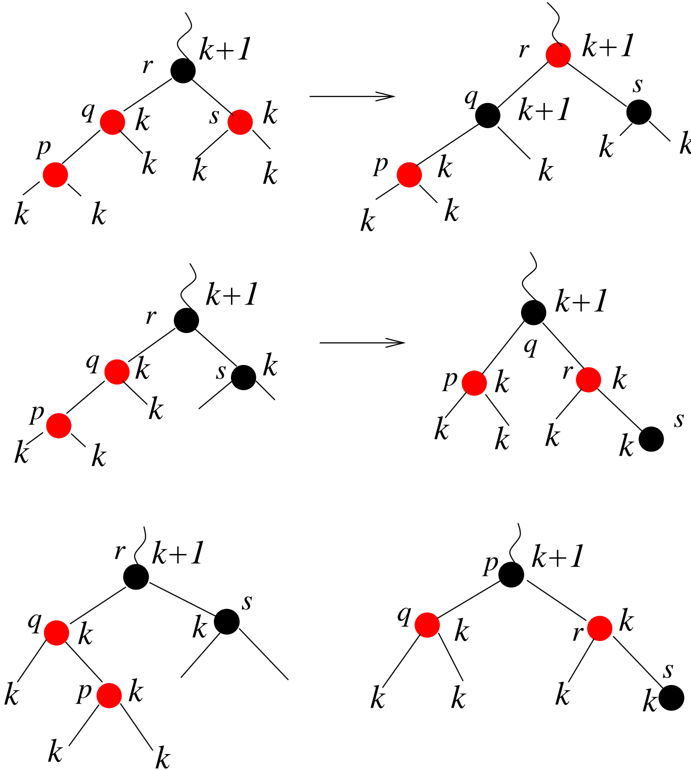
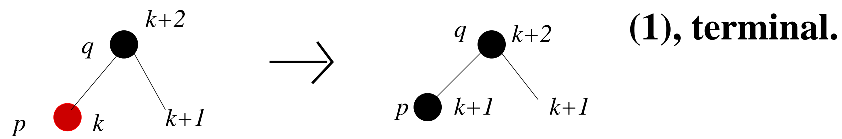


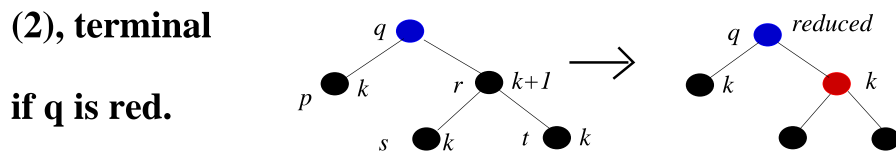
Figure 4: Fix double red

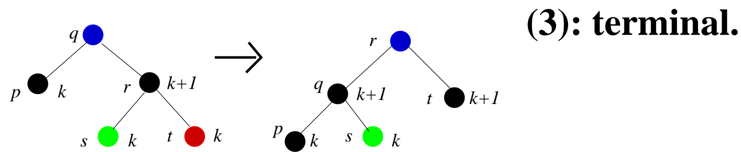
- If p is red, make it black, and stop. Otherwise, let r be the sibling of p and s and t its nearer and further children (i.e., if q is deficient on the left, then s is the left child of r).
- If r is black and s and t are null or black, then 'demote r.' Make r red. This fixes the imbalance at q, but reduces its rank. If q was red then make q black and stop. Else, if q is the root, decrease the rank (see 7.6), and stop. Otherwise replace p by q and q by the parent of q. Again there is an imbalance at q.
- If r is black, but t is (nonnull and) red, then rotate r, copy q's colour to r, make q and t both black, and stop.
- If r is black and t null or black but s is red, then rotate from s, make s black and r red, and continue. This will raise the previous case in the next iteration.
- Otherwise, r must be red and s, t both null or black. Rotate from r, make r black, and q red. At the next iteration one of the above cases will be raised, and there will be at most two more iterations.

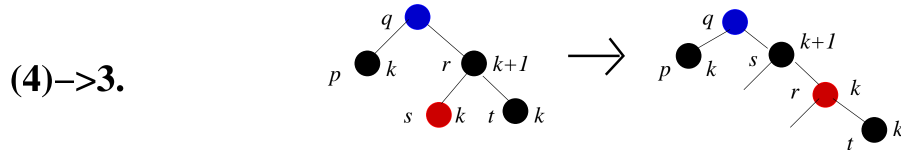
  (If the second case – sibling of p black with two black children – is raised, then it will terminate, because q is red.)

(7.11) Joining two red-black trees. Given two red-black trees T_L and T_R, and a node v, it is possible to form a tree T in which all nodes in T_L come first in inorder, then v, then all nodes in T_R. If T_R is empty, it is a matter of inserting appending v to T_L in time $O(\text{rank}(T_L))$, and similarly if T_L is empty.









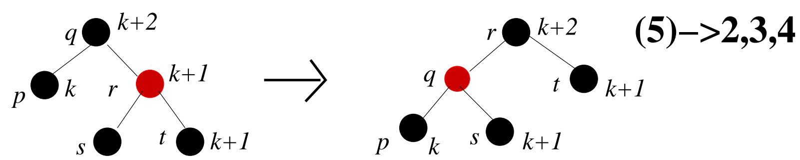


Figure 5: Fix rank deficit

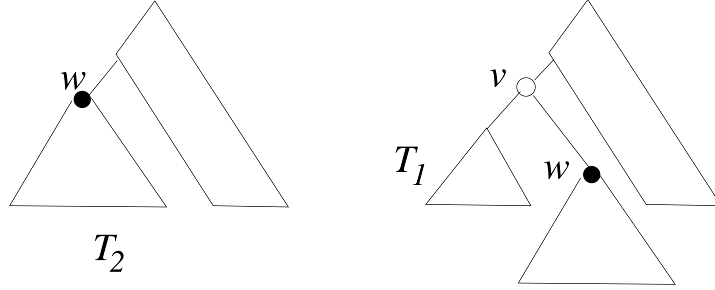


Figure 6: Joining one node and two trees. (The empty circle is a red node.)

Otherwise let r and s be the ranks of T_L and T_R respectively. If $r \leq s$, go down the leftmost branch of T_R until a black node w of rank r is found. Then replace w by v in T_R, make w its right child, and make T_R its left subtree. Make v red, and apply 'fix double red' at v to cure any double red situation that arises. See Figure 6.

If $r > s$, go down the rightmost branch of T_L to locate a black node w of rank s, and perform similar actions. The runtime, assuming that r and s are known in advance, is $O(|r - s|)$. The rank of the combined tree is $\max(r, s)$ or $\max(r, s) + 1$. Clearly,

(7.12) Lemma *If the ranks of T_L and T_R are known in advance, then the cost of joining is $O(|r-s|)$. In any event the cost is $O(\log n)$, where n is the total number of nodes in both trees.* ∎

(7.13) Splitting a tree. Given a node v in a red-black tree T with n nodes, it is possible to construct, in time $O(\log n)$, trees T_L and T_R containing all nodes preceding v (respectively, following v) in T in inorder, and preserving inorder in each tree, as follows.

- First calculate the rank of v, by counting the number of black proper ancestors and subtracting from the tree's rank.
- Let $x = v$ initially, T_L its left subtree, and T_R its right subtree. Possibly they are empty. Make sure the roots (where they exist) of these subtrees are black, adjusting their ranks if necessary. If x is the root then stop. Otherwise let y be its parent.
- If x was the left child of y then let S be the right subtree of y, making sure its root (if it exists) is black and adjusting its rank if necessary. Then join T_R, y, and S.

  Otherwise let S be the left subtree of y, adjusting its rank, and join S, y, and T_L.

  If y was the root, then stop. Otherwise let $x = y$, let y be its parent, and repeat.

Crudely put, splitting involves $O(\log n)$ operations each costing $O(\log n)$, yielding $O(\log^2 n)$. The sharp estimate $O(\log n)$ is based on the fact that the cost of joining is proportional to rank difference. First,

(7.14) Lemma *Suppose L and R are two (nonempty) red-black trees and x is another node. Let r and s be the tree ranks before the join. Without loss of generality, $r \leq s$ and x becomes a left descendant of the root of R. Suppose $L, x,$ and R are joined to form a tree T. Then T has rank s or $s + 1$.*

If the rank is $s + 1$ then either T has root x, with two black children, or its root u (which was that of R) had two red children in R before the join and two black children after.

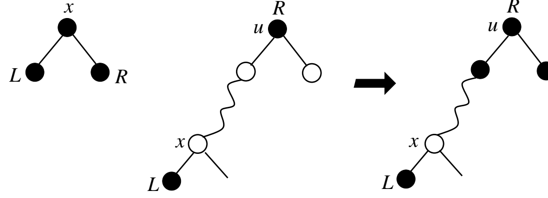


Figure 7: two ways in which joining L and R increases rank.

Proof. See Figure 7. Suppose the rank is $s + 1$. This can only have happened during fix double red where the root was red and then black. This is possible if x is made the root with L and R its left and right subtrees. Otherwise, the root must have become red and then been changed to black again, increasing the rank. The root can only have become red through 'promoting' its children, which must have been red and are made black. In this case there are no rotations and the root of T was the root of R. **Q.E.D.**

(7.15) Runtime of Split. In splitting a tree T into T_L, v, and T_R, T_R is constructed by a sequence of joins involving a sequence

$$T_0, x_1, T_1, x_2, \ldots, x_\ell, T_\ell$$

of subtrees and nodes where the ranks of the trees T_j are nondecreasing. If the root of T_j was red (as a child of x_j), then it is changed to black. Let r_j be the rank of x_j in T before splitting. These ranks are available as part of the computation.

Let U_k be the tree constructed after joining $T_0, x_1, \ldots, x_k, T_k$. We note that T_0 was the right subtree of v, but for $j \geq 1$ T_j was the right subtree of x_j in T. Here $U_0 = T_0$ is defined also.

(7.16) Lemma $r_k - 1 \leq \text{rank}(U_k) \leq r_k + 1$.

Proof. For $k \geq 1$, the rank of U_k is at least $r_k - 1$, since T_k had rank $\geq r_k - 1$ before adjustment, and the adjustments do not reduce rank.

The upper bound is harder.

We prove the upper bound by contradiction. Assuming the contrary, fix $k \geq 0$ minimal subject to

$$\text{rank}(U_{k+1}) > r_{k+1} + 1.$$

By interpreting x_0 as v we can include the case $k = 0, r_0 = \text{rank}(x_0) = \text{rank}(v)$. Also, let $U_0 = T_0$.

We note that $\text{rank}(T_j) \leq r_j$ for all j, because if the root, call it z, of T_k changes from red to black, then its parent x_j was black, so z had rank $r_j - 1$ before adjusting. The following holds (at any step of the construction, not just the k-th)

$$\text{rank}(U_{k+1}) \leq 1 + \max(\text{rank}(U_k), \text{rank}(T_{k+1})). \quad (7.17)$$

Note also that the '$1+$' term applies when there is a rank increase, as discussed in Lemma 7.14.

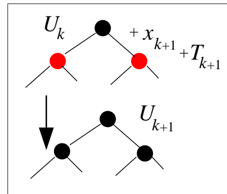
By choice of k, the left-hand side is $\geq 2 + r_{k+1}$, $\text{rank}(U_k) \leq 1 + r_k$: also, $\text{rank}(T_{k+1}) \leq r_{k+1}$. Therefore

$$2 + r_{k+1} \leq 1 + \max(1 + r_k, r_{k+1})$$

This is only possible if

$$r_{k+1} = r_k \quad \text{and} \quad \text{rank}(U_k) = 1 + r_k.$$

Also the second case of the rank-increase lemma applies to U_k, U_{k+1}:



Note

$$\text{rank}(U_1) \leq 1 + \max(\text{rank}(T_0), \text{rank}(T_1)) \leq 1 + r_1.$$

which shows that $k + 1 \geq 2$ and therefore U_{k-1} exists.
Since $r_k = r_{k+1}$, x_{k+1} was red and x_k black, so $r_{k-1} < r_k$.

$$\text{rank}(U_k) = 1 + r_k \leq 1 + \max(\text{rank}(U_{k-1}), r_k)$$

Now $\text{rank}(U_{k-1}) \leq 1 + r_{k-1} \leq r_k$, so in this equation too, the '1+' term applies, and the rank of U_k increased when it was built. It is then impossible (Lemma 7.14) that the root of U_k has two red children: a contradiction. **Q.E.D.**

(7.18) Corollary *Splitting a tree takes $O(\log n)$ time.*

Proof. Cost of building U_{k+1} is proportional to the difference in height between U_k and T_{k+1}. This is proportional to the difference in rank **(exercise).** Adding,

$$\sum |\text{rank}T_{k+1} - \text{rank}U_k| \leq \sum (2 + r_{k+1} - r_k).$$

This is twice the height of the tree plus its rank (at most): hence $O(\log n)$ ∎

8 Splay trees

(8.1) Amortised analysis. To implement red-black trees is complicated; there are alternatives which guarantee good asymptotic *amortised* runtimes for a sequence of operations, and which are much easier to program.

The idea of amortisation is to evaluate the total cost of a sequence of operations, and, if this total is low, not to worry about some of the operations being expensive. Repetitive usage of the same data-structure is the norm rather than the exception.

The approach to amortisation followed here uses the idea of a *potential function.* We suppose that a set U of n nodes is fixed in advance. At any time during processing, there is a collection of binary search-trees whose nodes are from U, and this set has a certain 'potential.' Specifically,

- Each one of the n nodes x has a fixed positive *weight* $w(x)$.
- During the processing, any node x has a definite set of descendants at any time (though its descendants may change over time); the *rank* of x is

$$\log_2 \left(\sum_{y \text{ descendant of } x} w(y) \right).$$

- The potential of the system is the sum of the node-ranks. Φ denotes the potential at fixed time.

(8.2) Amortised cost. The *amortised cost* of an operation on the trees is (some measure of its actual cost) + (new potential) - (old potential). The lemma below is trivial.

(8.3) Lemma *Suppose that a sequence of k operations is applied to the system with corresponding potentials* $\Phi_0 \ldots \Phi_k$*. Then the total actual cost is*
(total amortised cost) + $\Phi_0 - \Phi_k$*.*

(8.4) Splay heuristic. Suppose that x is a node, not currently the root of a tree. Let y be the parent of x. To *splay from* x means to raise x in the tree by one or two rotations as follows:

- 'Zig' case. If y is a root, rotate from x; then x becomes the root of the tree containing it. Otherwise let z be the parent of y.
- 'Zigzig' case. If x and y are both left children or both right children, Rotate from y, then rotate from x. Otherwise
- 'Zigzag' case. Rotate from x, and rotate from x again.

(8.5) See Figure 8 Splaying from x either makes x the root (first case) or brings it two steps closer to the root. The number of rotations is taken as the actual cost, so a 'zig' operation costs 1 and the other two cost 2. The following lemma gives a useful estimate of the amortised cost of a single splay operation.

(8.6) Lemma *Suppose that* r *and* r' *are the rank functions before and after a splay operation. Then the amortised cost is bounded by* $1 + 3(r'(x) - r(x))$ *for a zig, and* $3(r'(x) - r(x))$ *for a zigzig or zigzag.*

Proof. Zig. The descendants are unchanged except for x and for y, so the ranks are unchanged except at x and y.. Hence the amortised cost of the splay is

$$1 + (r'(x) - r(x) + r'(y) - r(y)).$$

Now, $r'(y) < r(y) = r'(x) > r(x)$, so the amortised cost is less than $1 + (r'(x) - r(x))$, which is less than $1 + 3(r'(x) - r(x))$.

In the other two cases the ranks are unchanged except at x, y, and z, and the amortised cost of the splay operation is

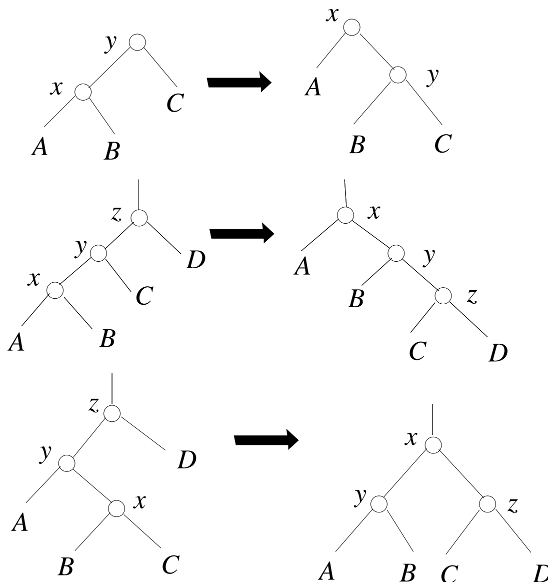


Figure 8: The three splay operations.

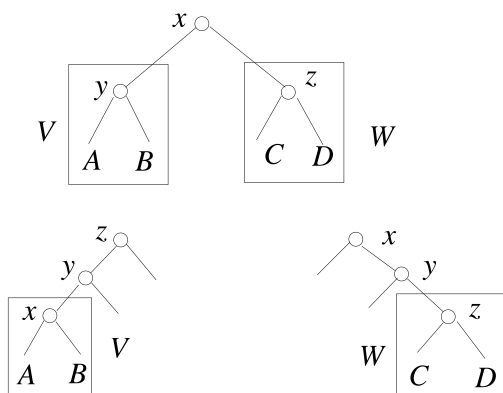


Figure 9: Amortised splay costs.

$$2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z).$$

Subtracting this cost from $3(r'(x) - r(x))$, we get

$$(8.1) \qquad 2r'(x) + r(y) + r(z) - r'(y) - r'(z) - 2r(x) - 2,$$

which we want to show is nonnegative. We shall consider two disjoint subsets V and W of nodes which together include all but one of the tree nodes. See Figure 9

The Zigzag case is the more natural, and will be considered first. Choose V and W to be the descendants of y and z after the splay, so $r'(y) = \log_2(|V|)$, $r'(z) = \log_2(|W|)$, and $r'(x) > \log_2(|V| + |W|)$. Bearing this in mind, we want to convert 8.1 to an expression involving just these three ranks, so $r(x), r(y)$, and $r(z)$ must go. Since $r(y) + r(z) - 2r(x)$ is positive, because both are ancestors of x before the splay, we can remove this term without increasing the result, leaving

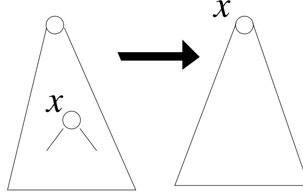


Figure 10: lookup — amortised cost $O(\log n)$.

$$2r'(x) - r'(y) - r'(z) - 2,$$

Write w for $|W|$ and v for $|V|$: the expression is greater than

$$2\log_2(v+w) - \log_2(v) - \log_2(w) - \log_2(4),$$

which equals

$$\log_2\left(\frac{(v+w)^2}{vw}\right).$$

Now, $(v-w)^2 \geq 0$, so $(v+w)^2 \geq 4vw$; the fraction in the above expression is at least 1, and the expression is nonnegative. Therefore 8.1 is nonnegative (actually, positive).

For the Zigzig case we choose V to be the descendants of x before the splay and W to be the descendants of z after the splay, so we want to remove $r(y)$, $r'(y)$, and $r(z)$.

We achieve this by subtracting $r(y) - r'(y) + r(z) - r(x)$ from the expression, a term which is clearly positive, and we are left with

$$2r'(x) - r(x) - r'(z) - 2,$$

which is positive by the same arguments as before. **Q.E.D.**

(8.7) Corollary *Suppose that a node x is brought to the root of a tree by a sequence of splay operations. Then the amortised cost of the sequence is bounded by $1 + 3(r(t) - r(x))$, where t was the root and r the ranks before the sequence.*

Proof. Apply the previous bounds iteratively, forming a telescoping sum. ∎

(8.8) Using splaying for insert, delete, join, and split. Splay operations involve only rotations and thus preserve inorder. They can be applied where only inorder need be preserved. They are applied as follows

- We assume that each node in the population of n nodes has weight $1/n$. Then the rank of a node with k descendants is $\log_2(k/n)$. The potential Φ is always between $-n\log_2 n$ and 0.

- **Look up** a key k in tree T: first search for k; let x be the last node visited — it contains K if k is in T. Then bring x to the root by repeated splaying. See figure 10.

  Taking the depth of x as the actual cost, the amortised cost is $O(\log n)$.

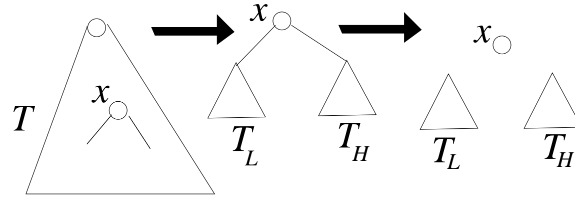


Figure 11: split — amortised cost of splay is $O(\log n)$, then removing the links from x reduces the potential further.

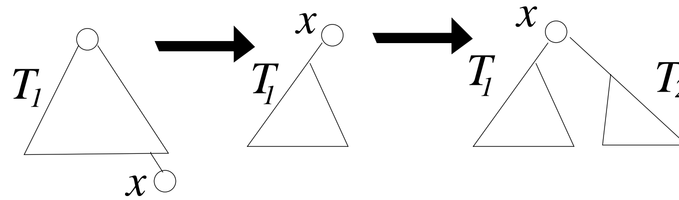


Figure 12: join — amortised cost of splay is $O(\log n)$. Making T_2 the right subtree at x increases potential and hence the amortised cost, but by $O(\log n)$.

- **Split** T at a node x: bring x to the root by splaying. Then the left subtree T_L of x contains all keys $< k$ and the right subtree T_H contains all keys $> k$; remove the links from x to these trees. See Figure 11.

  The amortised splay cost is $O(\log n)$. Removing the links from x further reduces the potential and the amortised cost.

- **Join** T_1 with T_2, where T_1 is nonempty: Access the rightmost node, call it x, in T_1, bringing it to the root; and make T_2 the right subtree at x.

  Amortised cost of accessing x and splaying is $O(\log n)$. Making T_2 the right subtree increases the rank of x, hence the potential, but by at most $\log_2 n$.

- **Delete** a node x: Access x, bringing it to the root, and let T_1 and T_2 be its left and right subtrees; delete the links from x to these subtrees. Join these trees if both are nonempty.

  Amortised cost of accessing x is $O(\log n)$. Deleting the links reduces potential and amortised cost. Joining the trees is $O(\log n)$ amortised.

- **Insert** a key k: Search for k; this search ends at a node x which contains either k, or the highest key $< k$ or the lowest key $> k$; if k not found, create[5] a new node y containing k. Bring x to the root by splaying, and insert y between x and its left or right subtrees as appropriate.

  Amortised cost of search and splaying is $O(\log n)$. Inserting y between x and one of its children increases the potential, and the amortised cost, but by $O(\log n)$.

(8.9) Corollary *The actual cost of a sequence of k operations of the above kinds is $O(k \log n)$.*

Proof. The amortised cost is $O(k \log n)$. Since the initial potential is at its minimal level, the amortised cost bounds the actual cost. **Q.E.D.**

[5] for the analysis, y is a node without children selected from the population of n nodes.

9 Heaps

A *heap* is a data-structure containing sortable keys, allowing for the *minimum* key to be accessed efficiently.

Representation. They are stored in an array which has been given the structure of a shallow binary tree: n items are in the first n array locations; the children of an index i are $2i + 1$ and $2i + 2$, where defined, i.e., where they are $< n$; 0 is the root; the parent of i (if $i \neq 0$) is $\lfloor (i-1)/2 \rfloor$.

(This is specific to C. If array indexing starts at 1 then the definition is different).

Heap order: keys must be in non-increasing order up the tree, that is, if $i \neq 0$ then the entry at i is $\geq$ the entry at $(i-1)/2$.

Suppose that we have something that is 'almost' a heap, except that one key might be out of place, too light or too heavy. If too light, it is moved up to the correct position using **sift_up()**, if too heavy it is moved down to a correct position using **sift_down()**.

```
typedef struct
{
  int size;
  TYPE * entry; /* sortable type */
}
  HEAP;

... make_heap etcetera ...
```

```
sift_up ( HEAP * heap, int k )
{
  TYPE temp = heap->entry[k];
  int i;
  int finished;

  i = k;
  finished = 0;
  while ( ! finished )
  {
    if (i==0 || heap->entry[(i-1)/2] <= temp )
    {
      heap[i] = temp;
      finished = 1;
    }
    else
    {
      heap[i] = heap[(i-1)/2];
      i = (i-1)/2;
    }
  }
}
```

```
sift_down ( HEAP * heap, int k )
{
  TYPE temp = heap->entry[k];
  int i, j;

  i = k;
  while ( i < heap->size )
  {
    j = 2*i + 1;
    if ( j < n-1 && heap->entry[j+1] < heap->entry[j] )
      j = j+1;

    if ( j >= heap->size || heap->entry[j] >= temp )
    {
      heap->entry[i] = temp;
      i = heap->size;
    }
    else
    {
      heap->entry[i] = heap->entry[j];
      i = j;
    }
  }
}
```

To add a key to a heap:

```
add it to the end (adjusting size) and sift_up
```

To delete the minimum:

```
move the last element to the top and sift_down
```

To build a heap from an array, copy the array elements into heap entries, and perform:

```
  for ( k = (heap->size-1)/2; k>=0; --i )
    sift_down ( heap, k );
```

Cost of **sift_up():** i is reduced by one-half (at least) in every iteration, and it stops when i reaches 0:

$$O(\log k)$$

Cost of **sift_down()**: i doubles per step (at least) and it stops when i reaches n; initially k; so

$$O(\log(n/k))$$

Cost of **build_heap()**:

$$\sum_{k=0}^{n/2} \log_2(n/k) \leq \log \frac{n^n}{n!}$$

which is $O(n)$.

10 Other sorting methods

Sorting a list means putting it in sorted order. The collection to be sorted is usually an array or a file. We have already seen **heapsort**, which can sort n items in time $O(n \log n)$, by building a heap and repeatedly removing the minimum.

We shall review **mergesort, quicksort, and radix sort (bin sort, bucket sort).**

10.1 Mergesort

Mergesort is easiest described in relation to arrays of sortable items. It is based on the following **merge()** routine.

Required: `a[low_a..hi_a], b[low_b..hi_b]` are sorted (in nondecreasing order).

```
void merge ( SORTABLE TYPE a[], int low_a, int hi_a,
      SORTABLE TYPE b[], int low_b, int hi_b,
      SORTABLE TYPE c[], int low_c )
{
  int i, j, k;
  i = low_a; j = low_b; k = low_c;
  while ( i <= hi_a || j <= hi_b )
    if ( i < hi_a && ( j == hi_b || a[i] < b[j] )
    {
      c[k] = a[i]; ++i; ++k;
    }
    else
    {
      c[k] = b[j]; ++j; ++k;
    }
}
```

Efficiency. Let r = `hi_a - low_a + 1` and s = `hi_b - low_b + 1`; at each iteration, either i or j is incremented, so the runtime is $O(r + s)$. Obviously this can't be bettered — 'linear time.' Obviously it is easily applied to two sorted files of data, also.

Mergesort() involves using `merge()` repeatedly to sort a collection, an array, say. It is necessary in `merge()` that the third array `c` be disjoint from the other two. For this reason, *2-column* mergesort is suitable. This sorts by mergesort, leaving the sorted result in `target`. (Note: `source` is not preserved.)

```
void mergesort ( SORTABLE TYPE source[],
                 SORTABLE TYPE target[], int n )
{
  SORTABLE TYPE * a, * b, * swap;
  int step, i, j, mid, top;
  a = source; b = target;

  for ( step = 1; step < n; step = step * 2 )
  {
    for ( i=0; i < n; i = i + 2*step )
    {
      mid = min ( i + step, n-1 );
      top = min ( mid + step-1, n - 1 );
      merge ( a, i, mid-1, a, mid, top, b, i );
      swap = a; a = b; b = swap;
    }
    if ( b != target )
      merge ( b, 0, n-1, b, n, n-1, a, 0 );
  }
}
```

Runtime: `step` doubles each 'pass,' so there are about $\log_2 n$ 'passes.' Each pass takes total time $O(n)$; therefore, $O(n \log n)$.

10.2 Quicksort

Quicksort is best described recursively. To quicksort a collection `a` of items:

```
qsort ( COLLECTION a )
{
  COLLECTION lo, eq, hi;
  if ( a is nonempty )
  {
    choose some key k from a;
    lo = keys < k in a;
    eq =  keys == k in a;
    hi =  keys > k in a;
    recursively qsort lo and qsort hi;
    replace a by lo, eq, hi concatenated.
  }
}
```

This can be applied to arrays or files or other kinds of collection. Its *worst-case* runtime is bad. Suppose that the key chosen is always the smallest in a. Then there are recursive calls to qsort sets of size $n-1$, $n-2$, and so on: the overall cost is $\Omega(n^2)$.

On the other hand, if the average-case is sought, assuming that the rank of the key k in `a` is 1 to n with uniform probability, we get

$$A(n) = n + \frac{1}{n}\sum_{r=0}^{n-1}(A(r) + A(n-1-r)$$

The n in the recurrence is an estimate of the cost of choosing k and splitting the list into `lo, eq, hi`. This is essentially the same as in estimating the IPL of binary search trees, so the average runtime of quicksort is $O(n \log n)$.

Remark. One should pay some attention to the *way* that the splitting is carried out. The average-case analysis is valid only if we can be sure that the splitting operation preserves randomness, in an easily-defined sense.

10.3 Radix sort

The idea is simple. Suppose that the sort keys are 2-digit decimal numbers. First split the set into 'buckets,' according to the high-order digit. Then sort each 'bucket' on the low-order digit. More generally, let d be the number of digits in each key. Digits are indexed 0 to $d-1$.

```
void recursive_sort ( COLLECTION s, int i, int d )
{
  if ( i < d )
  distribute s over 10 buckets according to  its i-th digit;
  for each bucket b,
recursive_sort ( b, i+1, d );
  replace s by the contents of these sorted buckets,
   concatenated
}
```

```
31 41 59 26  53 58

buckets
  26
  31
  41
  59 53 58
   Recursive sort of 5-s bucket
    buckets
    53
    58
    59
  result 53 58 59
result 26 31 41 53 58 59
```

There is a more streamlined way of doing this, called *least significant digit* bucket sort. In this, we sort on the second digit first

```
31 41 59 26  53 58 -> 31 41 53 26 58 59
```

and then on the first digit

```
26 31 41 53 58 59
```

It is essential, though, that each 'sorting pass' be *stable.* For example, the numbers `53 58 59` appear in that relative order after the first pass, and the second pass cannot disturb the relative order.

(10.1) Definition *A sorting method is* stable *if, after sorting, items with the same key appear in the same relative order as before sorting.*

The UNIX `sort` utility is not stable. You must use `sort -s` to ensure a stable sort.

The cost of bucket sort is $O(dn)$ where d is the number of digits — the number of passes. It can be used for 'mixed-radix' keys: pounds and ounces, for example, or for sorting names by first name within surname. Notice that a stable sorting procedure makes this very easy with the 'LSD' (least significant digit) method: first sort by first name, then sort *stably* by surname.

10.4 Remarks.

Apart from stability, there are two other aspects of sorting.

- Pointer sorting. Instead of sorting an array `a`, to produce an array `index` so that `a` is unchanged but `a[index[i]] <= a[index[i+1]`, $0 \leq i < n - 1$.

  This is easy to achieve by re-interpeting '¡='.

- In-place sorting. Mergesort requires additional storage, separating target from source (unless some very convoluted tricks are used).

- Bucket sort doesn't, but it's not a general method.

- Quicksort can work 'in-place,' with some messing.

- Heapsort as described here uses separate storage for the heap, but the heap and the source array can share places to get an 'in-place' heapsort.

11 Lower bounds for sorting and searching

11.1 Lower bound for searching.

This lower bound applies to the 'comparison model' of searching (and sorting).

- In this model, keys are supposedly sought through queries of the form 'is $x < y$?' or perhaps 'is $x \leq y$?'

- We imagine a more general program, written in C, say; moreover, since lower bounds are sought, we can imagine different programs for different n (the number of keys stored).

- So we consider a search program which deals with a fixed number n of keys.

- Since n is fixed, while- and for-loops can be eliminated, 'unwinding' them to a series of if-statements.

- For a lower bound, it is enough to count the number of if-statements executed (for a specific key), and, moreover, that the search is successful.

- Every if-statement can be assumed to include an else-part, even if the else-part does nothing.

- If the program contains code like

```
A;
if ( B ) then
  C;
else
  D;
E;
```

where `A;` includes no if-statements, but `E` may contain if-statements, it can be replaced by

```
A;
if ( B ) then
{ C; E;}
else
{ D; E;}
```

- Although this duplicates code, it doesn't increase the number of if-statements executed in searching for any key.
- Thus the program may be arranged in a binary tree, where the if-statements correspond to nodes with two children, call them 'branching nodes.' Call this arrangement a 'decision tree program.' Searching (successfully) for a particular key corresponds to travelling down this tree to a place where the key is 'known.'
- We consider the program to be *comparison-based* if a (successful) search for a particular key depends solely on the *rank* of the key, its place in sorted order among the keys stored.

  Obviously binary search or binary tree search fits this description.
- Put more simply, we could suppose that the search procedure computes and returns the rank of the key.

So the decision tree contains n nodes, at the very least, where the rank of a particular stored key is returned. These nodes are obviously independent, in that none can be ancestor of any other.

(11.1) Lemma *Let T be a tree containing n independent nodes. Then these nodes have total depth $\Omega(n \log n)$.*

Proof. Suppose that T has a non-branching non-leaf node u; that is, u has one child v. Then v can be mapped to u, reducing the number of non-branching nodes. Repeating this process, nodes in T can be mapped surjectively to a tree T' in which no node has just one child. If nodes x and y are independent in T then they are mapped to a set X of n independent nodes in T'. The point is that the number of *branching-node* ancestors in T matches the number of *ancestors* in T', and the depth of a node in T is not greater than the depth of its image in T'.

The tree T' can be pruned so that only nodes in X and their ancestors in T' are retained. We get a tree T'' with n leaves.

This is similar to Quiz 2, question 4, and one can conclude that the total leaf depth in T'' is $\Omega(n \log n)$. Hence the total depth of these nodes in T is $\Omega(n \log n)$. **Q.E.D.**

(11.2) Corollary *Under the comparison model, the worst-case cost of a search is $\Omega(\log n)$, as is the average cost of a successful search.*

Proof. The average cost is $\Omega(\log n)$, and the worst-case cost is no better than average.[6] ∎

[6]Assuming all keys equally likely.

11.2 Lower bound for searching.

We assume a 'comparison model' for searching. There is no notion of 'successful sort,' but in its place we make the assumptions

- All input keys are distinct: this does no harm, since we are looking for lower bounds.
- We assume that the number n of keys to be sorted is fixed.
- The sorting procedure uses 'pointer sorting,' say, so it computes and 'returns' a permutation which will convert the input to sorted order.
- There are $n!$ permutations, so the decision tree contains at least $n!$ independent nodes.

Question. Of the four sorting procedures considered, which are comparison-based?

Under these assumptions, the same ideas as with searching apply, and we get lower bounds. But this time the decision tree contains at least $n!$ independent nodes.

(11.3) Corollary *Under the comparison model, the average and worst-case cost of sorting is* $\Omega(n \log n)$.

Proof. In a decision tree with $n!$ independent nodes, the total depth of these nodes is $\Omega(n! \log n!)$ and the worst-case depth is $\Omega(\log n!)$. Thus the average (assuming all permutations are equally likely) is $\Omega(\log n!)$, and so is the worst-case.

But

$$
\begin{aligned}
\frac{n^n}{n!} &\leq e^n \\
n \ln n &\leq \ln(e^n) + \ln n! \\
n \ln n &\leq n + \ln n! \\
\ln n! &\geq (n-1) \ln n
\end{aligned}
$$

Thus $\log n!$ is $\Omega(n \log n)$. ∎

12 Knuth-Morris-Pratt algorithm

(12.1) The string-matching problem can be stated as follows. Given two arrays, $pattern[1 \ldots m]$ and $text[1 \ldots n]$, if the pattern occurs somewhere in the text, to return the index where the first occurrence begins (default 0, say).

The natural way to do this is to compare the pattern from left to right with $text[1 \ldots m]$, then with $text[2 \ldots m+1]$, and so on, until a match is found or the end of the text is reached. This approach yields an $O(mn)$ algorithm. When the pattern is $a^{m-1}b$ and the text is a^n, there will be $m(n-m+1)$ comparisons, so the algorithm is $\Omega(mn)$ also.

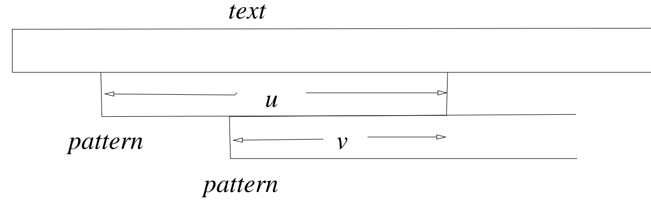


Figure 13: Overlapping argument underlying KMP algorithm.

(12.2) However, a partial match gives some information about the structure of the text, and this can be exploited to yield a linear time algorithm. Consider a partial match which matches a prefix u of the pattern beginning at text position i, but the next pattern character fails to match. Suppose that the pattern actually does occur at position $i + s$ in the text, where s is a fairly small shift. If $s < |u|$,[7] write r for $|u| - s$: the last r characters of u form a string v which is also a prefix of u (and of the pattern). So we need only consider shifts where the string u overlaps itself. See Figure 13. This leads to the following definitions.

(12.3) Definition *If x and y are strings then we write xy for the result of concatenating y to the end of x. This operation is associative.*

Let u and w be strings. If there exists a string v such that $w = uv$ (respectively, $w = vu$) then u is a prefix *of w (respectively,* suffix *of w).*

If u is both a prefix and suffix of w then it is a self-overlap *of w.*

(12.4) Lemma *Let u, v, w be strings.* (i) *If u is a prefix (respectively, suffix, self-overlap) of v and v a prefix (suffix, self-overlap) of w then u is a prefix (suffix, self-overlap) of w.*

(ii) *If u and v are both prefixes (suffixes, self-overlaps) of w then u is a prefix (suffix, self-overlap) of v or vice-versa, depending on the lengths of u and v. (Proof trivial.)* ▮

(12.5) Definition *For each nonempty prefix u of the pattern, let v be the longest proper self-overlap of u (possibly v is empty). Of course, u and v are defined uniquely by their lengths. Write $|v| = V(|u|)$. Then $V()$ is called the* overlap function *for the pattern.*

(12.6) Lemma *Let u be a prefix of the pattern. For all $i > 0$ for which $V^i(|u|)$ is defined, let u_i be the prefix of u of length $V^i(|u|)$. Then u_i is the i-th longest proper self-overlap of u.*

Proof. Suppose u_k is the shortest of these strings. Then u_k is the empty string.

By Lemma 12.4 (i), all the strings u_i are self-overlaps of u, proper since they are shorter than u.

Let v be any proper self-overlap of u. RTP $v = u_i$ for some i. If $v = \lambda$ then $v = u_k$. Otherwise choose the shortest string u_i such that $|u_i| \geq |v|$. It exists since u_1 is the longest proper self-overlap of u. Then v is a self-overlap of u_i by Lemma 12.4 (ii). By choice of i, $|u_{i+1}| < |v|$. Since u_{i+1} is the longest proper self-overlap of u_i, v is not a proper self-overlap of u_i, so $v = u_i$. **Q.E.D.**

[7] $|u|$ is the length of the string u.

(12.7) KMP algorithm. Granted that the overlap function V is available (stored in an array, of course), the following remarkably neat algorithm solves the pattern matching problem.

```
from           -- Eiffel version
  j := 0 k := 0
until
  j >= m or k >= n
loop
  if pattern.item(j+1) = text.item(k+1) then
    j := j+1
    k := k+1
  elseif j > 0 then
    j := V.item(j)
  else
    k := k+1
  end
end
```

(12.8) Correctness of this algorithm. A 'loop invariant condition' holds:

- `pattern[1..j]` matches `text[k-j+1..k]`
- `j` is maximal subject to the match extending to `j+1`. In other words, if `pattern[1..i+1]` matches `text[k-i+1..k+1]` then $i \leq j$.

Suppose that j and k are both incremented during an iteration. By the invariant condition, after incrementing, `pattern[1..j]` is the longest pattern prefix matching `text[1..k]`.

Otherwise, the pattern should be moved forward by the shortest shift which produces another match. If $j > 0$, this shift is given by `j:= V.item(j)`. If $j = 0$, then `pattern.item(1)` mismatches `text.item(k)`, and k should be incremented.

(12.9) Runtime. Next we consider the running time of this algorithm. The point is that in each iteration, either a partial match is extended or the pattern is moved forward, and whenever the pattern is moved forward the text position of the partial match never moves backwards (in a sense, mismatches are 'paid for' by previous partial matches). Putting this more precisely:

(12.10) Lemma *In a single iteration of the loop, consider the two quantities k and $k - j + 1$. Then these quantities never decrease, they are bounded by n, they begin at zero, and one or both quantities increases strictly. Hence the algorithm terminates within at most $2n$ iterations.*

(12.11) The above argument can be phrased more succinctly if obscurely by considering the combined quantity $2k - j + 1$. We turn to computing the overlap function $V[1 \dots m]$. Curiously, it can be computed in linear time by a strategy rather close to that of the pattern-matching algorithm proper.

```
V.put ( 0, 1 )    -- Eiffel version
from
  j := 1
  q := 0
until
  j = m
loop
  if pattern.item(j+1) = pattern.item(q+1)
  then
    j := j+1
    q := q+1
    V.put ( q, j ) ;
  elseif q > 0 then
    q := V.item(q)
  else
    j := j+1
    V.put ( 0, j )
  end
end
```

(12.12) correctness and runtime of calculating overlap. The runtime is $O(m)$ for the same reasons as in 12.9. Correctness follows from the observation that, given that u is the first j pattern characters, and ua the first $j + 1$, then q measures the longest self-overlap of u which might possibly extend to a self-overlap of ua. Therefore we conclude the following result.

(12.13) Proposition *The string-matching problem is solvable in time $O(m + n)$, where m is the length of the pattern and n that of the text.*

(12.14) Overlap function vs. failure function. We have used an 'overlap function' $V[j]$ here rather than the slightly different 'failure function' originally proposed by Knuth, Morris, and Pratt. This is because, although the failure function should be expected to produce better running-times, the given overlap function can be adapted to other uses, such as listing *all* occurrences (rather than the leftmost occurrence, which the 12.7 yields) of the pattern in the text.

C version.

```
int * make_overlap ( char * pattern )
{
  int m = strlen ( pattern );

  int * ovlp = ( int * ) calloc ( sizeof(int), m );
  int k, j;

  ovlp [0] = -1;
  k = 0;
  j = -1;

  while ( k < m - 1 )
    if (pattern[k+1] == pattern[j+1])
    {
      ++j; ++k; ovlp[k] = j;
    }
    else if ( j >= 0 )
      j = ovlp[j];
    else
    {
      ++k; ovlp[k] = -1;
    }

  return ovlp;
}
```

```
int kmp ( char * pattern, int * ovlp, char * text )
{
  int j,k;
  int m = strlen ( pattern )
  int n = strlen ( text );

  j = k = -1;

  while ( j < m-1 && k < n-1 )
    if ( pattern[j+1] == text[k+1] )
    {
      ++j; ++k;
    } else if (j >= 0)
      j = ovlp[j];
    else
      ++k;

  if ( j == m - 1 )
    return k-m+1;
  else
    return -1;
}
```

13 Aho-Coraskick structure

(13.1) Simultaneous pattern search with TRIEs. A *trie* (retrieval tree) is a tree whose links are labelled with characters. For every path from the root, the characters labelling the links define a character string.

The child links from any node are labelled with different characters, so every character string defines at most one path from the root. A set of strings can be represented in a trie as the set of paths leading to suitably marked nodes.

(13.2) Every node in the trie 'carries' a unique string. The root 'carries' λ. If p is a node, and x is the string it carries, and q is a child of p labelled a, the q carries xa. Generally the string x carrying p dictates the path from x to p. The set of strings stored in the trie is the set of strings carried by the marked nodes.

Given a string x, it is easy to find the node, if it exists, which carries x, and to check if that node is marked. Thus tries are handy for efficient storage of strings (rather wasteful of computer memory, though).

(13.3) The KMP algorithm can be extended to use a trie to search simultaneously for a set of strings

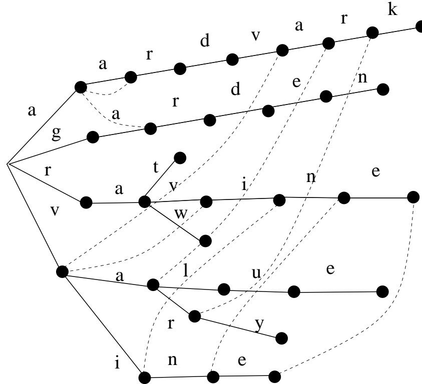


Figure 14: trie with back-links

in the text. The same method is used as before, only now the overlap function is a *mutual overlap* function.

(13.4) Definition *Let S be a set of nonempty strings.*

For any prefix u of any string in S, a proper mutual overlap *of u is any proper suffix of u which is also a prefix (not necessarily a proper prefix) of some string in S. Such a proper mutual overlap v may be a prefix of u, but need not be.*

(13.5) Definition *Let T be a trie, with* back-links *installed as follows: for any node p, not the root, let u be the string labelling tbe path from the root to p and let v be its longest mutual overlap. Then v defines a path leading to some node q of T, and q is the back-link from p.*

The back-link function in a trie corresponds directly to the overlap function, and the KMP algorithm can be adapted with very little change to search for all patterns stored in the trie simultaneously.

(13.6) Construction of backlinks. It is necessary to enter all the strings in the trie before installing the backlinks.

(13.7) Definition *For any node p, except the root, its* backlink *is the deepest node q such that the string carried by q is a proper suffix of the string carried by p.*

One can compute the backlinks efficiently so long as one processes the nodes in *nondecreasing* order of depth, i.e., in *breadth-first* order.

- Let p be a next node in breadth-first order (initially, the root).
- For each child q of p, let a be the letter labelling the link from p to q. Let the backlink from q be the root (default).

  Let $r = p$. Keep replacing r by the backlink from r (which is already computed, since p is deeper than r) until r is the root or r has a child s labelled a. If so, s is the backlink from q.
- Continue with another node p in breadth-first order...

Cost. For any node p, except the root, let

$$\text{gap}(p) = \text{depth}(p) - \text{depth}(\text{backlink})(p)$$

The amount of work done in computing the backlink at q is proportional (at worst) to $1 + \text{gap}(q) - \text{gap}(\text{parent}(q))$. Summing over all leaves of the tree, we get $O(n)$, where n is the total length of all strings stored in the trie.

14 The union-find problem

(14.1) The UNION-FIND problem is to maintain an evolving partition of a fixed set of n items — $\{1 \ldots n\}$ for simplicity — under two operations:

FIND(x): to identify the set in the partition currently containing x, and

UNION($s1, s2$): given two different sets in the partition to combine them into one.

The partition is initially into n singleton sets, and it coarsens over a mixed sequence of UNIONs and FINDs — there can be at most $n - 1$ UNIONs, which would bring all the items into the same set.

It is natural to represent the partition as a forest of trees, defined by an array `parent`, where `parent[x]` is the parent of x in the forest — -1 if x is currently the root of a tree of the forest. The array is initially -1.

With this arrangement, to implement `find(x)` it is enough to return the root of the tree currently containing x, and to implement `union(x1, x2)`, where x_1 and x_2 are currently roots, just make x_1 the parent of x_2 or vice-versa.

(14.2) Potential inefficiency, and size array. With no further control over the structures it is possible that the trees in the forest become deep, so a `find` operation could visit $\Omega(n)$ nodes, which is hardly efficient. The first improvement is to control the `union` operation to keep the trees shallow. This is achieved using another array `size`, where

> As long as `x` is a root, `size[x]` is the number of descendants `x` has in the forest.

(14.3) Size balancing. The array `size` is initialised to 1 for each entry. The UNION operation ensures that the node with fewer descendants is not made parent. This heuristic is called *size balancing*.

```
void uf_union ( int i, int j )
{
  if ( i == j )
    return;

  if ( size[i] >= size[j] )
  {
    size[i] += size[j];
    parent[j] = i;
  }
  else
  {
    size[j] += size[i];
    parent[i] = j;
  }
}
```

Clearly `size[x]` is maintained as the number of descendants of x, whether or not x is a root: however this will not hold after we revise the `find` operation.

(14.4) Lemma *A node of height k has size at least 2^k.*

PROOF. The size-balancing heuristic ensures that the size of the parent of a node is at least twice the size of the node. There exists a path of length k beginning at a leaf and ending at the node; hence the size is at least 2^k. ∎

(14.5) Path compression. The revised `find` operation includes a 'path-compression' heuristic as follows:

```
int uf_find ( int u )
{
  int v,w,x;
  v = u;
  while ( parent[v] >= 0 )
    v = parent[v];
  w = u;
  while ( w != v )
  {
    x = parent[w];
    parent[w] = v;
    w = x;
  }
  return v;
}
```

The idea is that nodes are brought closer to the root of the tree containing them: the tree retains the same root y, so the heuristic doesn't invalidate the algorithm. However, it doesn't update the `size` array, so `size[x]` is only guaranteed to count x's descendants if x is a root. As a result of the size-balancing strategy, no tree's height exceeds $\log_2(n)$, so the worst-case runtime of `find(x)` is $O(\log(n))$. The amortised runtime is much less.

(14.6) The 'reference forest.' We are going to estimate the overall runtime of a mixed sequence, call it S, of UNIONs and FINDs. For the rest of the discussion we shall suppose S to be fixed. The sequence S constructs a forest of trees. It helps to imagine another forest defined in terms of S, formed by executing the UNIONs but not collapsing the trees with path compressions. We call this imaginary forest the *reference forest*.

(14.7) Rank. We begin by defining the *transient rank* of a node x to be

$$\lfloor \log_2(\texttt{size[x]}) \rfloor.$$

The transient rank of x is subject to change as long as x remains a root; however, once x acquires a parent, `size[x]` is frozen; we take the *final* value of the transient rank, and define this to be the *rank* of x. Equivalently,

(14.8) Definition *The rank of x is $\lfloor \log_2(s) \rfloor$, where s is the number of descendants x has in the reference forest.*

(14.9) Lemma (i) *The parent of x in the real forest is always an ancestor of x in the reference forest;* (ii) `rank(parent[x]) > rank[x]` *always, once x gets a parent.*

Proof. (i) is true when x first acquires a parent y, since y is its parent in the reference forest. Thereafter whenever the parent u of x is replaced by another node v, by induction we can assume

that v is an ancestor of u in the reference forest, and u and ancestor of x, so v is an ancestor of x in the reference forest.

(ii) by the same arguments as in 14.4, all proper ancestors of x in the reference forest have rank greater than `rank(x)`, so (ii) follows from (i). ∎

(14.10) Our analysis will involve a rapidly increasing sequence $F_0, F_1, F_2 \ldots$

$F_0 = 0$, but we shall not specify the sequence yet. Given a node x, let its *rank group* be the least r such that $\mathtt{rank}(x) \leq F_r$.

(14.11) Lemma *There are at most* $n/2^r$ *nodes of rank* r.

Proof. From Lemma 14.9, all nodes of the same rank are independent nodes in the reference forest (that is, neither is ancestor of another, and they have disjoint sets of descendants). By Lemma 14.4, each such node has at least 2^r descendants. Thus, if $x_1, \ldots, x_k$ are all the nodes of rank r and $X_1, \ldots, X_k$ their descendants, then

$$\sum_{j=1}^{k} |X_j| = |\bigcup X_j| \leq n,$$

and $|X_j| \geq 2^r$ for each j, so $\sum 2^r \leq n$, i.e., $k2^r \leq n$ or $k \leq n/2^r$. Q.E.D.

(14.12) Corollary *If* $r > 0$*, there are fewer than*

$$\frac{n}{2^{F_{r-1}}}$$

nodes in rank group r.

Proof. Nodes with rank group $r > 0$ have ranks between $F_{r-1} + 1$ and F_r. Hence, applying the above lemma, there are fewer than

$$\sum_{s=F_{r-1}+1}^{\infty} \frac{n}{2^s} = \frac{n}{2^{F_{r-1}}}.$$

Q.E.D.

(14.13) The sequence F_j defined. We now specialise the sequence F_j by requiring $F_{j+1} = 2^{F_j}$; a very rapidly increasing sequence, and the above corollary implies

(14.14) Lemma *If* $j > 0$ *then there are fewer than* n/F_j *nodes in the* j*-th rank group.* ∎

(14.15) Definition $\log^* n$ *is the smallest* s *such that* $F_s \geq n$.

Let L be the index of the highest rank group, i.e., the smallest L such that F_L bounds all node ranks. Since the ranks are bounded by $\lfloor \log_2 n \rfloor$,

$$L \leq \log^* n.$$

(14.16) Amortised cost of a find operation. Let us consider a FIND operation; indeed, let $x_0, x_1 \ldots x_k$ be the sequence of nodes visited tracing from $x_0 = x$ to the current root $y = x_k$.

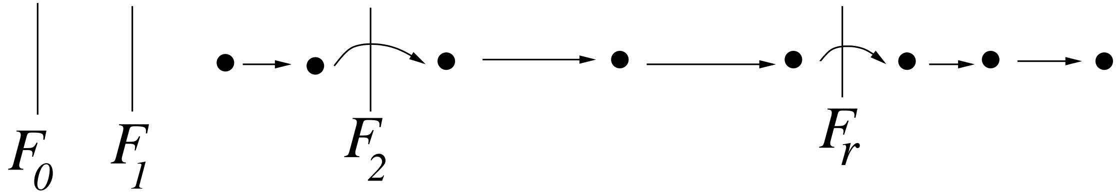


Figure 15: amortised cost of a FIND.

Estimate the real cost of the FIND by the length of the path, i.e., k. We imagine that the cost is measured by the number of nodes from x_0 to x_{k-1}. See figure 15

Rather than using a potential function (which we could), it is more natural to amortise the cost in the following way. When x_j and its parent x_{j+1} are in the same rank group, we 'charge' one unit to x_j. This accounts for all of the cost except where x_j and x_{j+1} are in different rank groups. The 'amortised' cost then is bounded by the number of 'inter-group jumps' which is bounded by L or $\log^* n$.

If x is in the r-th rank group, so its rank is $> F_{r-1}$ and $\leq F_r$ (suitably interpreted at $r = 0$), every time it gets charged in this way its parent's rank increases; so after rather less than F_r FINDs its parent must be in a higher rank group.

Thus every node in the r-th rank group acquires at most F_r charges in this way; since there are at most n/F_r elements in this group (if $r > 0$), that accounts for at most n charges for the entire group. Cost of the unions is, of course, $O(n)$, so

(14.17) Theorem *A mixture of m FINDs with any number of UNIONs costs $O((m+n)\log^* n)$.* ∎

15 Hash tables

Another way of accessing keys is through an array based on *hash addresses.* Let $h()$ be a hash function, some function which produces an index within the array's range.

We shall not worry about computing the hash address. Sometimes $x \mod M$ is good enough, where x is a bitstring regarded as an integer and M is the array size. The point is that the hash function should be fairly random, i.e., the values of $h(x)$ are uniformly distributed in the array's range.

If all the keys produce different hash addresses then we have 'perfect hashing.' This is rarely possible.

15.1 Chaining

So we must expect to store groups of keys with the same hash-value. The simplest way to do this is with *chaining*, where the array contains pointers to linked lists, and when a key x with hash-value i is stored, it is stored in the i-th list in the array (Figure 16).

Put another way, the keys are stored in 'buckets.'

Also the general practice is to insert a key, which is not stored, at the place where the search ends. For chaining that means at the end of the list searched.

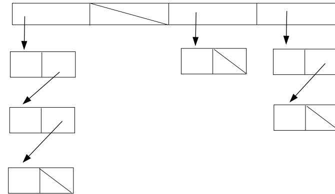


Figure 16: chaining

```
/* LIST_ENTRY as described earlier ... */

LIST_ENTRY * lookup ( KEY x, TABLE * table )
{
  LIST_ENTRY * p, * q;
  int i;

  i = h(x);
  p = NULL;
  q = table -> header [i];

  while ( q != NULL )
    if ( q -> key == x )
      return q;
    else
    {
      p = q;
      q = q -> next;
    }

  /* at this point, q is null and
   * the key has not been found
   */

  q = make_list_entry ();
  q -> next = NULL;
  q -> key = x;

  if ( p == NULL )
    table -> header = q;
  else
    p -> next = q;

  return q;
}
```

(15.1) Analysis of chaining. Suppose $x_1, \ldots, x_n$ are the keys inserted. When the r-th key is inserted, there are $r-1$ stored; and with suitable assumptions of randomness, the length of the 'bucket' searched for the r-th key is roughly $(r-1)/m$. Summing from $r=1$ to n, and dividing by n to get the average search, we get

$$\frac{1}{n}\frac{n(n-1)}{2}\frac{1}{m} \approx \frac{n}{2m}.$$

This is the average cost of unsucessful search, which is roughly the same as the cost of successful searching as well.

15.2 Open addressing

Under open addressing, the keys are stored in the hash table. Convention: n is the number of keys stored and m is the size of the hash table. This time $n \leq m$ or else the table is full; as with chaining, the ratio n/m is significant.

Now, when searching for a key x with hash-value i, one may see table entries containing keys with *other* hash-values. In this case we have a so-called (secondary) *collision*.

The most interesting one to analyse is that of *linear rehashing*. Under this scheme, in searching for a key x, one tries the positions

$$h(x), \;\; h(x) \oplus c, \;\; h(x) \oplus 2c, \ldots$$

We use $a \oplus b$ as an abbreviation for $(a+b) \mod m$. As usual, m is the hash table size and c is any increment, so long as c is prime to m; otherwise the search will cycle through a fraction of the table. We may as well fix $c = 1$.

We investigate the effect of inserting a sequence

$$x_1, x_2, \ldots, x_n$$

of keys. With the key sequence left implicit, we define, for $0 \leq j < m$,

$$B_j = \text{number of keys with hash-value } j$$
$$C_j = \text{number of keys whose search included } j \text{ and } j \oplus 1$$

(C_j measures 'carry past j').

(15.2) Lemma

$$C_{j\oplus 1} = \begin{cases} 0 & \text{if } C_j + B_{j\oplus 1} \leq 1 \\ C_j + B_{j\oplus 1} - 1 & \textit{otherwise.} \end{cases} \quad \blacksquare$$

Let p_k be the probability distribution for B_j and q_k that of C_j. Under randomness assumptions, the distribution is the same for all j.

The above lemma, and common sense, yield the following

(15.3) Lemma (i) *The distribution p_k is binomial*

$$\binom{n}{k}\frac{1}{m^k}\left(1-\frac{1}{m}\right)^{n-k}$$

(ii)

$$\begin{aligned} q_0 &= p_0q_0 + p_1q_0 + p_0q_1 \\ q_k &= \sum_{i+j=k+1} p_iq_j \quad \blacksquare \end{aligned}$$

Generating functions. The generating function for a probability distribution p_j is defined as

$$\sum_j p_j z^j$$

For the given (binomial distribution) let us call it $B(z)$:

$$\begin{aligned} B(z) &= \sum_k \binom{n}{k}\frac{z^k}{m^k}\left(1-\frac{1}{m}\right)^{n-k} \\ &= \left(1+\frac{z-1}{m}\right)^n \end{aligned}$$

Let $C(z)$ be the generating function for the q_j.

$$\begin{aligned} q_0z &= p_0q_0z + (p_1q_0 + p_0q_1)z \\ q_kz^{k+1} &= \sum_{i+j=k+1} p_iq_jz^{k+1} \quad (k > 0) \\ zC(z) &= p_0q_0z + B(z)C(z) - p_0q_0 \end{aligned}$$

Express $B(z)$ (which is known) as follows:

$$B(z) = 1 + (z-1)D(z)$$

thus introducing a related (and known) function $D(z)$.

$$\begin{aligned} zC(z) &= p_0q_0(z-1) + C(z) + (z-1)C(z)D(z) \\ (z-1)C(z) &= p_0q_0(z-1) + (z-1)C(z)D(z) \\ C(z)(1-D(z)) &= p_0q_0 \\ C(z) &= \frac{p_0q_0}{1-D(z)} \\ C(z) &= \frac{1-D(1)}{1-D(z)} \end{aligned}$$

the last because $C(1) = 1$.

$$C(z) = \frac{1 - D(1)}{1 - D(z)}$$
$$C'(z) = \frac{(1 - D(1))(D'(z))}{(1 - D(z))^2}$$
$$C'(1) = \frac{D'(1)}{1 - D(1)}$$

$$B(z) = 1 + (z-1)D(z)$$
$$B'(z) = D(z) + (z-1)D'(z)$$
$$B'(1) = D(1)$$
$$B''(z) = 2D'(z) + (z-1)D''(z)$$
$$B''(1) = 2D'(1)$$

$$B'(z) = \frac{n}{m}\left(1 + \frac{z-1}{m}\right)^{n-1}$$
$$B'(1) = n/m$$
$$B''(z) = \frac{n(n-1)}{m^2}\left(1 + \frac{z-1}{m}\right)^{n-2}$$
$$B''(1) = n(n-1)/m^2$$

Therefore

$$C'(1) = \frac{B''(1)}{2(1 - B'(1))} = \frac{n(n-1)/m^2}{2(1 - n/m)}$$
$$= \frac{n(n-1)}{2m(m-n)}$$

(15.4) Counting carries. Again we consider inserting keys $x_1, \ldots, x_n$ in that order. The key x_j will be inserted once it has been determined not to be already stored; the path followed in that initial unsuccessful search will be followed in any subsequent search for x_j.

Suppose that the path has length $r + 1$, so r locations probed are already occupied. For each of these locations there is another carry-past, namely, x_j; so the search for x_j contributes to r of the carry-pasts.

The total number of probes is the total carry-past $+n$.

The *average* carry past $\sum_j jq_j$ at a fixed location is $C'(1)$. The total is $mC'(1)$, and therefore the average total number of probes is

$$mC'(1) + n. \tag{15.5}$$

(15.6) Definition *S_n and U_n are the average lengths of successful and unsuccesful searches with n items stored under linear rehashing.*

Divide Equation 15.5 by n to get S_n.

$$S_n = 1 + \frac{n-1}{2(m-n)}.$$

Splitting the constant,

$$S_n = \frac{1}{2} + \frac{n-1+m-n}{2(m-n)} = \frac{1}{2} + \frac{m-1}{2(m-n)}$$

Put $\alpha = n/m$, the hashing density.

(15.7) Lemma *The average length S_n of a successful search under linear rehashing is*

$$\frac{1}{2} + \frac{m-1}{2(m-n)}$$

Ignoring the -1 term, we get

$$S_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right). \quad \blacksquare$$

To get the average length of an unsuccessful search, we observe that every key is added following an unsuccessful search in a table with fewer keys (clearly $U_0 = 1$), so

$$S_n = \frac{1}{n}\sum_{j=0}^{n-1} U_j.$$

so

$$\begin{aligned} U_n &= (n+1)S_{n+1} - nS_n \\ &= \frac{1}{2} + (n+1)\left(\frac{m-1}{2(m-n-1)}\right) - n\left(\frac{m-1}{2(m-n)}\right) \\ &= \frac{1}{2} + \left(\frac{m-1}{2}\right)\left(\frac{mn - n^2 + m - n - mn + n^2 + n}{(m-n-1)(m-n)}\right) \\ &= \frac{1}{2} + \frac{m(m-1)}{2(m-n-1)(m-n)}. \end{aligned}$$

By ignoring the -1 terms in the above expression, we get

(15.8) Lemma *The average length U_n of an unsuccessful search in a table with n items stored, linear rehashing, is approximately*

$$\frac{1}{2} + \frac{1}{2}\frac{1}{(1-\alpha)^2}. \quad \blacksquare$$

15.3 Uniform and double hashing

Uniform hashing is an ideal: double hashing approximates it. In double hashing, for every key x there are two hash functions, $h(x)$ and $k(x)$, and the search sequence is: a linear search beginning at $h(x)$ with steplength $k(x)$.

It is necessary that

$$\gcd(k(x), m) = 1$$

otherwise the search will cycle back before all locations have been probed.

(15.9) Proposition *With random hash functions h and k, double hashing behaves like uniform hashing (see below).* ▮

(15.10) Definition *A hashing scheme has the* uniform hashing property *if the hash sequences are independent, so that search lengths follow a geometric distribution.*

Properly speaking, since there is a finite sample space, a hypergeometric distribution, but the difference is negligible.

With n keys stored, the probability of a slot being occupied is n/m. As previously write $\alpha = n/m$ so the distribution of unsuccessful search lengths is

$$p_k = \left(\frac{n}{m}\right)^{k-1}\left(1 - \frac{n}{m}\right) p_k = \alpha^{k-1}(1-\alpha)$$

for $k = 1, 2, \ldots$.

Therefore (with U_n and S_n possessing their old meaning in this new context)

$$U_n = \sum_j (1-\alpha) j \alpha_{j-1} = (1-\alpha)\frac{d}{d\alpha}\frac{1}{1-\alpha} = \frac{1}{1-\alpha}.$$

To get S_n:

$$S_n = \frac{1}{n}\sum_{k=0}^{n-1}\frac{1}{1-k/m} = \frac{m}{n}\sum_k \frac{1}{1-\alpha_k}\frac{1}{m}$$

The sum can be approximated by the integral

$$\int_0^{\alpha}\frac{1}{1-x}dx = -\ln(1-\alpha)$$

Thus

$$S_n \approx \frac{1}{\alpha}\ln\frac{1}{1-\alpha} \quad ▮$$

15.4 Bibliographic notes

The analysis of linear rehashing is presented in 'Algorithms: their complexity and efficiency,' by Lydia Kronsjö.[8] Knuth (AOCP volume 3) gives a much more difficult analysis, and points out the slight inaccuracy in the other one (the 'carry past' probabilities are on an infinite probability space). That double hashing is close to uniform is apparently due to Guibas and Szemerédi, 1976.

16 Bloom filters

A Bloom filter is a cheap way of searching, generally for words in a document, where accuracy is traded for economy. One uses k independent hash functions into a bitstring of length m, where k and m are design choices.

To add a key x to the filter, calculate the set of $\leq k$ hash values

$$\{h_1(x), \ldots, h_k(x)\}$$

and set the filter bits to 1 at these places.

To query a key x, calculate the set of $\leq k$ hash-values, and check the bits at these positions; if all these bits are on, accept; if at least one is off, reject.

Thus, there are *no false negatives*: no string registered in the filter can be rejected. There can be false positives, i.e., x can pass the filter test for accidental reasons.

They are famously used in spell checkers. Some spelling mistakes may be missed, but this doesn't seem to do much harm.

Also in pattern matching, Bloom filters are used to reject words which are certainly not in the document, and KMP or Boyer-Moore can be used to check those not rejected by the filters.

17 Directed graphs

(17.1) We begin the study of algorithms operating on graphs and directed graphs. It will be seen that the *runtime* of these algorithms is usually obvious, but their *correctness* is not.

It was remarked at the start of this course that runtime analysis is rather 'soft' — growth rates are established using $O()$ and Ω — in contrast with, say, numerical algorithms for which accuracy is critical. But *correctness* is critical. In graph algorithms we shall try to be careful about their correctness, although this may degenerate into 'sketch proofs.'

(17.2) Definition *A* directed graph *or* digraph *is a pair* (V, E) *of* vertices *and* edges *where* V *is finite and*

$$E \subseteq \{(u, v) \in V \times V : \quad u \neq v\}.$$

The absence of edges (u, u) reads: there are no self-loops.

[8] communicated by a student, Margaret Gallery

(17.3) Definition *A* path *in the graph is a sequence $v_0, \ldots, v_k$ of vertices such that for $0 \leq j \leq k-1$, $(v_j, v_{j+1}) \in E$.*

It is trivial *if $k = 0$. It is* simple *if $0 \leq i < j \leq k \implies v_i \neq v_j$.*

A directed cycle *is a path $v_0, \ldots, v_k$ such that $v_k = v_0$. It is* simple *if the path $v_1, \ldots, v_k$ is simple.*

Directed cycles may be cyclically permuted, i.e., if $v_0, \ldots, v_k$ is a directed cycle and $1 \leq i \leq k-1$, then $v_i, \ldots, v_k, v_1, \ldots, v_i$ is a cycle containing the same edges and vertices.

A digraph is acyclic *if it contains no nontrivial cycles.*

There are two ways to represent digraphs. One is by an *adjacency matrix* A, supposing that the vertices are $1 \ldots n$: $A = [a_{ij}]_{n \times n}$, where

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E. \end{cases}$$

The other is by *adjacency lists,* where for each vertex u there is attached the list of all vertices *adjacent to* u, i.e., a list representing the set

$$\{v \in V : \ (u, v) \in E\}.$$

The latter representation is usually more economical, because graphs arising in practice are usually sparse, with $o(n^2)$ edges.

The *indegree* of a vertex v is the number of vertices u such that $(u, v) \in E$: similarly outdegree.

If $G = (V, E)$ is a digraph and $w \in V$, then

$$G \backslash w = (V \backslash \{w\}, \{(u, v) \in E : \ u, v \neq w\})$$

A *topological order* on a digraph is a sequencing of its vertices

$$v_1, \ldots, v_n$$

with the property that if $(v_i, v_j) \in E$ then $i < j$.

(17.4) Lemma *If a digraph has a topolgical order, then it is acyclic.*

Proof. Equivalently, if it has a nontrivial cycle then it cannot be topologically ordered. For suppose $v_1, \ldots, v_n$ is any ordering and $v_{i_0}, \ldots, v_{i_k}$ is a nontrivial cycle. The cycle may be cyclically permuted if necessary so that we can assume $i_k = i_0$ is the maximum index of all vertices on the cycle. Then (v_{i_0}, v_{i_1}) is an edge and $i_1 < i_0$, so the order is not a topological order.

The converse involves an inductive argument. Namely, G is acyclic iff G is empty, or it has a vertex u of indegree zero, and $G \backslash u$ is in its turn acyclic.

In this discussion, a node whose indegree is zero (in G) is a *source* (in G).

Certainly, if G is nonempty and has no vertex of indegree zero, then by tracing backwards from any starting vertex, one must visit the same vertex twice, between which we get a cycle.

So if G has a vertex u of indegree zero, and G contains a cycle, then that cycle cannot contain u, so $G \backslash u$ also contains a cycle.

So here is a construction of an ordering which will turn out to be a topological ordering.

- $G_0 = G$.
- For $i = 0, \ldots, n-1$, while G_i has a vertex of indegree zero: choose some such vertex.
- Let v_{i+1} be this vertex, and let $G_{i+1} = G_i \backslash v_{i+1}$.

If G is acyclic, then this produces a sequencing $v_1, \ldots, v_n$. If G is not acyclic, then for some $i < n$ G_i will have no vertices of indegree zero.

Suppose G is acyclic and (v_i, v_j) is an edge of G. G_j contains only the vertices $v_j, \ldots, v_n$, so v_i was deleted beforehand and $i < j$. Thus the sequence is a topological ordering. **Q.E.D.**

(17.5) An algorithm for topological sorting. We assume the graph is represented by adjacency lists, and that one can label to each vertex v of G with a number, indegree(v), which is initialised to the indegree of v in G. (This is easily done by traversing the vertices and their adjacency lists.)

A set `source` is initialised to the set of sources of G. It can be implemented in any way which allows constant-time insertion and deletion.

- Initialise i to 0.
- While source is nonempty:
  - delete a vertex u from source
  - increment i and output u as v_i
  - traverse the adjacency list for u and for all edges (u, v), decrement indegree(v); if indegree(v) becomes zero, then add v to source.

(17.6) Convention. $n = |V|$ and $m = |E|$ always.

(17.7) Runtime. The amount of work done processing a vertex u is proportional to $1 + k$ where $k =$ outdegree(u). The overall runtime is $O(m + n)$.

18 Depth-first search, acyclicity, and strong connectivity

Depth-first search is a curious operation on graphs and digraphs G, which is defined recursively and is surprisingly useful.

It is best understood as yielding a sequencing $v_1, \ldots, v_n$ of the vertices and a forest of 'spanning trees' such that for any vertex v_i, the descendants of v_i in the forest are precisely the vertices reachable from v_i in

$$G \backslash \{v_1, \ldots, v_{i-1}\}$$

(This notation is an obvious extension of the notation $G \backslash v$.)

We always work with an adjacency list representation.

This calls for a *parent* link to be attached to each vertex, together with a *preorder* rank, and sometimes *postorder* rank as well. Preorder and postorder make good sense with any tree, not just binary trees (inorder is different). A full DFS (depth-first search) of the digraph is executed as follows:

- Preorder ranks are initialised to zero for all vertices. Also, global variables pre_count and post_count are initialised to zero.
- Traverse the vertices of G in any order. When considering a vertex u, if its preorder rank is still zero then it hasn't yet seen in DFS: set parent(u) to NULL and dfs(u).

The dfs(u) procedure is recursive.

- Increment pre_count and assign it to preorder_rank (u)
- For each vertex v adjacent to u, if its preorder_rank is zero, then assign u to parent(v) and dfs(v)
- Increment post_count and assign it to postorder_rank (u)

(18.1) Lemma *If G is acyclic, and DFS is executed, then reverse postorder rank is a topological order.*

Proof. In other words, if (u, v) is an edge and u precedes v in postorder, then G is not acyclic.

There are two cases. First case: u precedes v in preorder. That means that when dfs(u) was executed, v had not yet been visited, and it is reachable directly from u, so it must become a descendant of u in dfs(u), and postorder(v) $<$ postorder(u), contrary to hypothesis.

In the second case u follows v in preorder and u precedes v in postorder. This means that u is a descendant of v in the depth-first forest, so there is a path from v to u; also (u, v) is an edge, so there is a cycle containing u and v. **Q.E.D.**

(18.2) The strongly connected components of a digraph G. Two vertices u, v are *strongly connected* if there exist paths from u to v and from v to u (i.e., there exists a cycle containing u and v). This is an equivalence relation whose equivalence classes partition V. The subgraphs spanned by these classes are the *strongly connected components* of G.

(18.3) Lemma *Suppose a full DFS is executed. If u and v are strongly connected, they belong to the same DFS tree.*

Proof. Let C be a cycle containing u and v, and let w be the vertex of minimum preorder rank in C. Then all vertices on C, including u and v, become descendants of w during dfs(w). **Q.E.D.**

(18.4) Lemma *Suppose that DFS is executed on G and the vertices are output in* reverse postorder *to a set S. Then suppose that a copy G' is made of G, with the same vertices but its edges* reversed. *If DFS is executed on G', restarting the search according to the order given in S, then the trees in the second DFF span the sccs of G.*

Proof. Vertices u, v are strongly connected in G iff they are strongly connected in G', so the SCCs of G and G' correspond.

Every tree in the second DFF spans a union of sccs of G', hence of G. We need to show that no tree spans more than one BCC.

Suppose otherwise. Choose a second DFF tree T whose root belongs to a BCC K but which contains vertices in other BCCs. Choose a vertex w in the tree such that $w \notin K$ but all proper ancestors of w in T are in K.

There is a path from the root of T, call it u, to w in T, so there is a path from w to the root in G, but no path from any vertex in K to w in G.

Case (1) w is visited before v in Phase 1. Then v, and all vertices in K, are descendants of w, so w follows u in postorder, a contradiction.

Case (2) v is visited before w in Phase 1. Let x be the phase 1 high point of K, so v is a descendant of x. Since there is no path from x to w in G, w is not a descendant of x.

Since v is visited before w, so is x, being an ancestor of v; and since w is not a descendant of x, x, and all its descendants, including u, precede w in postorder, a contradiction. ∎

19 A better SCC algorithm

The double DFS method of computing SCCs is as simple as possible, but impractical. Here is more practical method using a single DFS.[9]

We say that one node is *earlier* than another if it is visited earlier during the DFS — so it ends up with lower preorder rank.

We begin with an observation.

(19.1) Lemma *Let K be a SCC, and u the earliest vertex in K. As before, all vertices in K are descendants of u. Then the vertices in K span a* prefix subtree tree *at u, meaning that if $v \in K$ then all ancestors of v, which are descendants of u, are in K.*

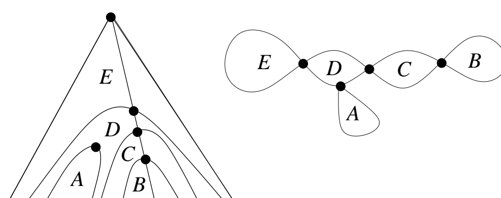


Figure 17: the nodes in each SCC form a prefix subtree.

Proof. There is a branch of the DFS tree from u to v, and there is a path in G from v to u; together they make a directed cycle, and all vertices in the cycle are in the same SCC, namely, K. **Q.E.D.**

Relative to a DFF, the edges of G can be classified as follows.

- Tree edges (u, v): u is the parent of v in the DFF.
- Forward edges (u, v) where v is a DFF descendant of u but not a child of u.
- Back edges (u, v) where v is an ancestor of u.
- Cross edges (u, v) where v is not an ancestor of u but v precedes u in preorder.

The algorithm needs the following

[9] See (Aho, Hopcroft, Ullman). Originally: R.E. Tarjan (1972). Depth-first search and linear graph algorithms. *SIAM J. Computing* **1:2**, 146–160.

(19.2) Definition *The* high point *of a SCC is the vertex visited earliest. It is ancestor to all other vertices in the SCC.*

Given some efficient method of detecting high points, the following algorithm computes SCCs: modified `dfs(u)`. It uses a *pushdown stack.*

- Push u onto the stack.
- For each edge (u, v) out of u,
    - If v has not been searched, `dfs(v)`.
    - Adjust a quantity `link_rank(u)`
- All edges (u, v) have been scanned. If at this point, it is determined that u is a high point, remove all vertices on stack down to and including u, emitting them as a SCC.

Correctness of this procedure. Briefly put, in `dfs(u)`, all children of u in the same SCC as u remain on the stack together with their descendants which are in this SCC; all children of u in other SCCs are high points and the vertices in their SCCs are removed from the stack before `dfs(u)` finishes. ∎

(19.3) Definition *Given a vertex v, suppose K is the SCC containing it. The* link *of v is the earliest vertex x such that for some descendant w of v, (w, x) is a back edge or a cross edge, and the root of the SCC containing x is an ancestor of v.*

Clearly v is earliest in its SCC iff it equals its link.

(19.4) Modified dfs. This uses a pushdown stack, and needs to attach a boolean variable to every vertex: `cleared`, as well as `preorder_rank` and `link_rank` fields.
`preorder_rank` is used to test whether a vertex has been visited already.
Initially, `cleared` is set to 1 (true), and `preorder_rank` to 0, for all vertices v.
Dfs(u) is called repeatedly for unvisited nodes u until all the digraph has been processed.
Dfs(u) does the following.

- Set `preorder_rank` in the usual way. Also set `link_rank` to the same value.
- Push u onto the stack, setting `cleared(u) = 0` (false).
- For each child v of u,
    - If v has `preorder_rank` zero (unvisited), recursively dfs(v).
    - Next, If v is not `cleared`, and its `link_rank` is less than the current `link_rank` of u, then reset u's link_rank to v's.
- It can be shown that, at this point, u is a high point iff `link_rank (u) == preorder_rank(u)`. If this holds, output an SCC as described previously. (Remember that when vertices are popped, their `cleared` value should be set to 1 (true).)

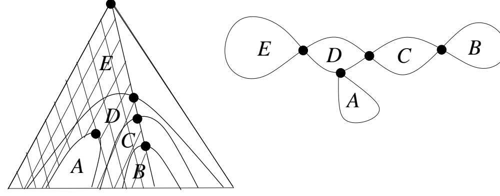


Figure 18: the nodes in each SCC contiguous on stack.

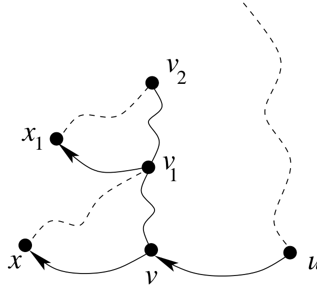


Figure 19: chain of links leading back to an ancestor of u.

Let us write $\texttt{link}(u)$ for the link of u, i.e., that vertex v such that, after `dfs(`u`)` ends, has preorder rank `pre_rank(`u`)`. Also write K_u for the SCC containing u. And write $\texttt{high_pt}(u)$ for the high point in K_u.

(19.5) Lemma *All proper ancestors of u remain on the stack when* `dfs(`u`)` *ends.*

Proof. Otherwise there exists an ancestor v of u which is cleared from the stack before `dfs(`u`)` ends. Say v is cleared at the end of `dfs(`x`)`. Then it is at or below x on the stack and therefore must have been added to the stack during `dfs(`x`)`, so it is a descendant of x. Therefore u is a descendant of x and `dfs(`u`)` ends before `dfs(`x`)` ends, contradicting the assumption that `dfs(`x`)` ends before `dfs(`u`)` ends. ∎

(19.6) Lemma *For all u, at all times,* $\texttt{link}(u) \in K_u$*, and after* dfs(u) *ends,* $\texttt{link}(u) = u$ *iff* $\texttt{high_pt}(u) = u$*, and in this case the vertices cleared from the stack are exactly those in K_u.*

Proof. This is proved by induction on the postorder rank of u, i.e., the order in which dfs(u) *finishes.*

Base case: u is the first vertex in postorder, so it is a leaf in the DF forest and at the top of the stack.

(a) If u has no out-edges, then $K_u = \{u\}$, $\texttt{high_pt}(u) = u$, $\texttt{link}(u) = u$, and K_u is output correctly.

(b) If u has out-edges, then since u is the earliest leaf, there is a back edge, $\texttt{link}(u) \neq u$, $\texttt{high_pt}(u) \neq u$, and nothing is output.

Induction. We may assume the following:

For all vertices v preceding u in postorder, $\texttt{link}(v) = v$ iff $\texttt{high_pt}(v) = v$, and v is cleared iff $\texttt{high_pt}(v)$ precedes u in postorder.

(c) Say $\texttt{high_pt}(u) \neq u$. RTP $\texttt{link}(u) \neq u$ when `dfs(`u`)` ends.

There is a path in G from u to $\texttt{high_pt}(u)$. Let v be the first vertex on the path which is not a descendant of u. Write the path as

$$u = u_0, u_1, \ldots, u_k, v, \ldots$$

Every vertex on the path is in K_u.

Since u_k is a descendant of u, we can assume that the edges from u_0 to u_k are all tree edges. Obviously we may assume that u occurs only once on the path.

Since v is not a descendant of u it is not a descendant of u_k, and (u_k, v) is either a back edge or a cross edge.

If it is a back edge then v is an ancestor of u, a proper ancestor by assumption, and it is uncleared until after dfs(u) ends (lemma 19.5). Therefore $\texttt{link}(u_k) \leq v < u_k$ in preorder. If $k > 0$ then u_k is uncleared after dfs(u_k) finishes, so $\texttt{link}(u_{k-1}) \leq v < u_{k-1}$ in preorder, and so on for u_{k-2}, $u_{k-3}, \ldots$, so $\texttt{link}(u) < u$ in preorder.

If it is a cross edge then v precedes u_k in postorder as well as in preorder. Since $v \in K_u$, $\texttt{high_pt}(v)$ does not precede u (nor u_k) in postorder, so v is uncleared and $\texttt{high_pt}(v) \leq v < u_k$ in preorder, so $\texttt{link}(u_k) < u_k$ in preorder after dfs(u_k) ends, and u_k is uncleared.

If $k > 0$, it follows that $\texttt{link}(u_{k-1}) \leq v$ after dfs(u_{k-1}) ends; and so on, up the tree, until we reach u: $\texttt{link}(u) < u$ in preorder when dfs(u) ends.

(d) Suppose $\texttt{link}(u) < u$ (in preorder) after dfs(u) ends. It is fairly obvious, and easy to show, that there is a path from u to $\texttt{link}(u)$

$$u = u_0, u_1, \ldots, u_k = v_1, x_1, \ldots \texttt{link}(u)$$

where for each pair x, y of successive vertices on the path, except possibly the last, $\texttt{link}(x)$ is copied from $\texttt{link}(y)$ during dfs(x). Since $\texttt{link}(u) < u$ in preorder, not all vertices are descendants of u.

Here $k \geq 0$ and u_k is the last descendant of u on the path.

We need to show that $\texttt{high_pt}(u) \neq u$. It is enough to show that there exists a proper ancestor of u reachable from u.

If (v_1, x_1) is a back edge, then x_1 is an ancestor of a descendant of u, and is not a descendant of u, so it is proper ancestor of u reachable from u, as required.

If (v_1, x_1) is a cross edge, then dfs(x_1) has terminated before dfs(v_1) begins, and x_1 is not cleared, so $x_1 \neq \texttt{link}(x_1)$ and by induction $x_1 \neq \texttt{high_pt}(x_1)$, and there exists a proper ancestor v_2 of x_1 reachable from x_1. Suppose v_2 is not a proper ancestor of u.

20 Bipartite matching

A *bipartite* graph is an undirected graph whose vertices are partitioned into two 'sides' X and Y, and whose edges only connect vertices on different sides. Equivalent to 2-colourable.

A bipartite graph resembles a multivalued partial function from X to Y or vice-versa.

A *matching* is a function 'covered' by the graph. Equivalently, it is a set of pairwise vertex-disjoint edges.

The *maximal matching problem* is: given a bipartite graph, produce a matching M of maximum cardinality.

The standard approach to this problem is to use *augmenting paths* to extend the matching.

(20.1) Definition *Let M be a matching in a (bipartite) graph G.*

M is perfect *if $|X| = |Y|$ and $|M| = |X|$.*

Call an edge e exposed *if $e \notin M$. Call a vertex* exposed *if it is not incident to an edge in M.*

A path P is alternating *if the edges along the path are alternately in M and exposed. It is* augmenting *if it is simple and nontrivial, and the first and last edges are exposed.*

Given an augmenting path P, represented as a set of edges, the set of edges

$$(M \backslash (P \cap M)) \cup (P \backslash M)$$

is a matching of cardinality $|M| + 1$. So we need to search for augmenting paths.

Every vertex in X is incident to at most one edge in M, so $|M| \leq |X|$ and $|M| \leq |Y|$ and if there are no exposed vertices, so M is perfect, then M is maximal.

Start with an exposed vertex x. Build a tree in breadth-first order (this is not essential). We describe the construction in layers, though properly a queue should be used.

- Layer 0 is just x.
- Layer 1 contains all children of x. For all vertices u in this layer, if u is matched to a vertex v, then v goes into layer 2.

  If there is an exposed vertex in this layer, stop: there is an augmenting path.
- Otherwise, develop layer 2. For all edges $\{u, v\}$, where u is in layer 2, and $\{u, v\}$ is *exposed,* and v is not already seen, add v to layer 3.
- Develop layer 3. For all vertices u in layer 3, if u is exposed, stop. Otherwise, let $\{u, v\}$ be the matching edge and add v to layer 4,
- Etcetera.

Implicitly, tree links are installed — we omit the details.

Starting with the empty matching, this procedure can be used repeatedly to locate augmenting paths and enlarge the matching. It is reasonably efficient. Hopcroft and Karp improved the algorithm and/or the analysis.

It is when the procedure *fails* to reveal an augmenting path that is interesting.

Write $L_0, L_1, \ldots$ for the layers. Without loss of generality, $x \in X$. Write

$$A = L_0 \cup L_2 \cup \ldots \cup L_k B = L_1 \cup L_3 \cup \ldots \cup L_{k-1}$$

This implies k is even; this is so, because whenever a vertex u is a leaf in the tree, if it is at odd level then the tree branch to u is an augmenting path.

More generally, if $\{u, v\}$ is an edge, and $u \in A$, then either it is a matching edge matching v to u, so $v \in B$, or it is exposed, and v would have been added to the tree if not already there; so again $v \in B$. In other words,

$$\Gamma(A) = \text{(def)} \quad \{v : (\exists u \in A)\{u, v\} \in E\} = B.$$

Note that $|L_1| = |L_2|$, $|L_3| = |L_4|$, and so on. Therefore

$$|A| = |B| + 1$$

(20.2) Lemma *A necessary, and obviously sufficient, obstruction to a perfect matching is that there exists a subset A of X or Y such that*

$$|A| > |\Gamma(A)|.$$

21 2014 exam syllabus, 3467 module

- Needless to say, runtime analysis is expected in most of the following items.
- Average depth of binary search trees, average depth of binary trees. (You should know the latter estimate, but its proof will not be asked.)
- Binary search trees: insertion and deletion. The $O(n \log n)$ average case does not apply with deletion. Michaela Heyer gave an alternative strategy for deletion.
- Red-black trees: definition, height bounds, insertion, deletion, join, and split, runtime analysis; $O(\log^2 n)$ is an acceptable estimate for splitting, but the 'sharp' estimate is actually $O(\log n)$. Know how `fix_double_red()`, `fix_rank_deficit()` are used; you won't need to remember the details of these routines.
- Splay trees: state and prove the amortised cost of 'splay to root.' How to split, join, insert, delete, with amortised costs.
- Heap operations, including sift up and sift down, and building a heap in linear time. Heapsort (the direct method, not the 'in-place' method). Mergesort, quicksort, radix sort. Lower bound for sorting (not searching, as the notes said, misleadingly).
- Knuth-Morris-Pratt algorithm: using the overlap table, and constructing it. (Be able to produce C code or a close description.)

  Describe how to generalise it to search simultaneously for all patterns in a retrieval tree (trie), assuming that 'back-links' have been installed in the trie. (Aho-Corasick structure.)

  Cost of building and of using the Aho-Corasick structure.
- Union-Find strategy: forests, size balancing, path compression. $O((m+n)\log^* n)$ overall cost. Alternative strategies: rank balancing, path splitting (as in the quiz).
- Hash tables: linear rehashing, uniform and double hashing — analysis.
- Directed graphs: topological sorting (first method). Depth-first search and reverse postorder (topological sorting again). Sharir's strong components algorithm. Tarjan's strong components algorithm.
- Bipartite matching. Method of alternating augmenting paths. How to search for augmenting paths. Prove the $|\Gamma(Z)| \geq |Z|$ criterion for existence of a perfect matching.