Sorting and Searching using Controlled Density Arrays

by

Robert Melville

David Gries

TR 78-362

Computer Science Department

Cornell University

Ithaca, New York

This research supported in part by National Science Foundation Grant MCS 76-22360

Abstract

Algorithms like insertion sort run slowly because of costly shifting of array elements when a value is inserted or deleted. The amount of shifting, however, can be reduced by leaving *gaps* -- unused array locations into which new values can be inserted -- at regular intervals in the array. The proper arrangement of gaps is maintained by periodic adjustment.

We demonstrate this technique with a stable comparison sort algorithm with expected time $O(N\log N)$, worst case time $O(N\sqrt{N})$, and space requirements 2N. We conjecture that, by using an interpolation search, the expected time can be reduced to $O(N\log\log N)$. By comparison, Quicksort takes expected time $O(N\log N)$, worst case time $O(N^2)$ and space $N+\log N$.

Second, we show that for any fixed $d\geq 2$ a table management algorithm can be constructed that can process a sequence of N instructions chosen from among INSERT, DELETE, SEARCH, and, MIN in worst case time $O(N^{1+1/d})$. Experiments with a version of the algorithm using $d=2$ show a marked improvement over balanced tree schemes for N as large as several thousand.

1. Introduction

We begin by presenting our simple idea behind sorting, which may be all the educated reader will need. Following this, we outline the rest of the paper.

We want to perform a variation of insertion sort [1,80-81]. At each step, there is a sorted portion $a[1:n]$ of array a and an untouched portion $a[n+1:N]$. Each step consists of finding the correct position $a[j]$, $1 \leq j \leq n+1$, into which $a[n+1]$ is to be placed and then shifting the elements $a[j:n]$ up one position to make room for it. This shifting, however, can take time $O(n)$, leading to an $O(N^2)$ algorithm. Shifting, and not searching, is the expensive operation.

In order to reduce the amount of shifting, we introduce *gaps* into the array: empty elements into which new values can be inserted. For example, suppose at one point the already sorted values $V_1, V_2, \ldots, V_n$ appear in $a[1:m]$, where $m=2n$, with exactly one gap before each value:

$$\text{gap},\ V_1,\ \text{gap},\ V_2,\ \ldots,\ \text{gap},\ V_{n-1},\ \text{gap},\ V_n \qquad (1)$$

and suppose for the moment that the values $V_{n+1}, \ldots, V_N$ are stored elsewhere. Then with a suitable encoding of gaps the place $a[j]$ to put V_{n+1} can still be determined in time $O(\log n)$ using binary search. But insertion of V_{n+1} requires no shifting since it can fill a gap. Likewise, insertion of $V_{n+2}, V_{n+3}, \ldots$ will be cheap. One readily recognizes the similarity to hashing with linear probing [1,518-521]. Provided the table does not get too full (provided not too many gaps are filled) and given a uniform distribution of values, the expected number of shifts required when inserting a new value is O(1).

At some point, however, table a[1:m] will become too full and shifting will become costly. It is then time to <u>configure</u>, to arrange the sorted values into form (1) with a gap between each value. Thus, the basic algorithm is:

```
initialize a[1:m] to contain $V_1$ and have form (1);
do n≠N ->
        do table a[1:m] is not too full ->
                n:= n+1; insert $V_n$ into a[1:m]
        od;
        if n≠N -> configure [] n=N -> skip fi
od
```

The problem is to determine what "not too full" means. The right balance between inserting and configuring must be found so that neither, in total, overrides the other. We analyze this problem in Section 2. In section 3 we give the algorithm in detail and in Section 4 we discuss a version that allows deletion as well as insertion -- i.e., that operates in the setting of "table management algorithms", where balanced trees have previously been used.

Other configurations of gaps are possible. For example, the original idea developed by the first author was:

$$\underbrace{\sqrt{n}\text{ values, }\sqrt{n}\text{ gaps, }\ldots,\ \sqrt{n}\text{ values, }\sqrt{n}\text{ gaps}}_{\sqrt{n}\text{ times}}$$

This form leads nicely to a recursive construction that for any given $d \geq 2$ supports a sequence of N operations in worst case time $O(N^{1+1/d})$. This is presented in section 5.

We close in section 6 with some empirical results and general remarks.

2. Worst case time and expected time analysis

The algorithm outlined in Section 1 contains three important parts: searching a[1:m] to determine the element a[j] where a value x should be placed; inserting x, including shifting values a[j],a[j+1],... up one position to make room for x; and configuring a[1:m] when it becomes too dense.

Assuming a suitable encoding of gaps, binary search can be used to search a[1:m] and, since the search occurs once for each element to be inserted, total search time will be O(NlogN). A mixture of binary search and interpolation search [2] can be used and we conjecture that this achieves worst case time O(NlogN) and expected time O(NloglogN) for searching. This will save time only if N is extremely large and we do not discuss it further.

Let us consider inserting and configuring together since their costs are related. We present in table 1 worst case asymptotic bounds for the algorithm using three different strategies for scheduling configurations. The bounds for the first two methods are proved in Theorems 2.1 and 2.2. The third method, used in the algorithm of section 3, is a combination of the first two. Theorem 2.3 proves the bound $O(N\sqrt{N})$ for the third strategy.

	configure after $\sqrt{n'}$ insertions	configure after $n'/2$ insertions	configure after $n'/2$ insertions or when more than $2\sqrt{n'}$ shifts needed for an insertion, whichever comes first
configure	$N\sqrt{N}$	N	$N\sqrt{N}$
insert	$N\sqrt{N}$	N^2	$N\sqrt{N}$
search	$N\log N$	$N\log N$	$N\log N$
total	$N\sqrt{N}$	N^2	$N\sqrt{N}$

n': number of elements in sorted portion at last configuration

N: total number of elements to be sorted

Table 1. Worst case bounds for three strategies.

Theorem 2.1 Let n_i be the number of values in the table just after the i-th configuration, with $n_0=1$. Suppose that $c\sqrt{n_i}$ values are inserted between the i-th and i+1-st configuration for some fixed $c>0$:

$$n_0 = 1,\ n_{i+1} = n_i + c\sqrt{n_i} \text{ for } 0\leq i$$

where by $\sqrt{n_i}$ we mean the square root truncated to an integer. Then the total worst case times for configuring and inserting are proportional to $N\sqrt{N}$.

Proof Let there be e configurations, so that $n_e < N \leq n_e + c\sqrt{n_e}$.

Each n_i satisfies $n_i<N$. The number of values inserted is N, where

$$N > 1+\sum_{i=1}^{e} n_i-n_{i-1} = 1+\sum_{i=1}^{e} c\sqrt{n_{i-1}} > c\sum_{i=1}^{e}\sqrt{n_{i-1}}$$

The total number of values rearranged during all configurations is

$$\sum_{i=1}^{e} n_i = \sum_{i=1}^{e} n_{i-1} + \sum_{i=1}^{e} c\sqrt{n_{i-1}} < N + \sum_{i=1}^{e} n_{i-1}$$

$$< N + \sum_{i=1}^{e} \sqrt{N}\sqrt{n_{i-1}} = N + \sqrt{N}\sum_{i=1}^{e}\sqrt{n_{i-1}} < N + N\sqrt{N}/c$$

Hence the worst case time for all configurations is no worse than $O(N\sqrt{N})$.

The worst case time for inserting occurs when the maximum number of shifts is made at each step. Between the i-th and i+1-st configuration this is:

$$0+2+\ldots+(c\sqrt{n_i}-2) < (c\sqrt{n_i})(c\sqrt{n_i}+1) = c^2n_i+c\sqrt{n_i}$$

shifts. The total number of shifts over all insertions is thus bounded by

$$\sum_{i=0}^{e}(c^2n_i+c\sqrt{n_i})$$

$$< c^2\sum_{i=0}^{e} n_i + c\sum_{i=0}^{e}\sqrt{n_i} < c^2 + c^2N + cN\sqrt{N} + c + N$$

Hence the worst case for insertions is $O(N\sqrt{N})$. Q.E.D.

Theorem 2.2 Let n_i be the number of values in the table just after the i-th configuration, with $n_0=1$. Suppose for some fixed α, $0 < \alpha < 1$, that αn_i values are added between the i-th and i+1-st configurations:

$$n_0 = 1, \quad n_{i+1} = n_i+\alpha n_i = (1+\alpha)^i \quad \text{for } 0\leq i$$

Then the total worst case times for configuring and inserting are proportional to N and N^2, respectively.

Proof Let there be e configurations so that $n_e < N \leq n_e + \alpha n_e$. Thus $(1+\alpha)^e < N \leq (1+\alpha)^{e+1}$. The total number of values rearranged during all configurations is:

$$\sum_{i=1}^{e} (1+\alpha)^i = \frac{(1+\alpha)^e - 1}{(1+\alpha)-1} - 1 < \frac{N}{\alpha}$$

Hence all configuring takes linear time.

The worst case for inserting occurs when the maximum number of shifts is made at each step. This happens, for example, when the values are in reverse order. Given n_i values, with a gap between each pair, to insert n_i new values takes at most

$$0 + 2 + 4 + \ldots + (2\alpha n_i - 2) < \alpha^2 n_i^2 - \alpha n_i$$

shifts. The total number of shifts over all insertions is bounded by

$$\sum_{i=0}^{e} (\alpha^2 n_i^2 - \alpha n_i) = \alpha^2 \sum_{i=0}^{e} (1+\alpha)^{2i} - \alpha \sum_{i=0}^{e} (1+\alpha)^i$$

$$= \alpha^2 - \alpha + \alpha^2 \frac{(1+\alpha)^{2e} - 1}{(1+\alpha)^2 - 1} - \alpha \frac{(1+\alpha)^e - 1}{(1+\alpha) - 1}$$

$$< \alpha^2 - \alpha + \alpha \frac{(1+\alpha)^e}{\alpha+2} (1+\alpha)^e - \frac{\alpha+2}{2} + \frac{2}{\alpha+2}$$

$$< \alpha^2 - \alpha + \frac{\alpha}{\alpha+2} N^2 - N + \frac{2}{\alpha+2}$$

Hence the insertions in the worst case take time $O(N^2)$. Q.E.D.

Theorem 2.3 Let n_i be the number of values in the table just after the i-th configuration, with $n_0 = 1$. Suppose a configuration occurs whenever an insertion causes a shift of $2\sqrt{n_i}$ or more values. Then the total worst case times for configuring and inserting are $O(N\sqrt{N})$.

Proof The time for configuring follows from Theorem 2.1. The bound on inserting is determined as follows. Suppose e configurations are performed, each occurring after $i_1, i_2, \ldots, i_e$ insertions, respectively. Note that $i_1+i_2+\ldots+i_e \leq N$. After i_1-1 (say) insertions the longest possible sequence of values in the array has length $2i_1-1$, so that the insertion that causes the configuration can cost no more than $2i_1$ shifts. Hence the insertions that cause the configurations can require no more than

$$2i_1+2i_2+\ldots+2i_e \leq 2N$$

shifts. Each of the N-e insertions that does not cause a configuration needs less than $2\sqrt{N}$ shifts, so that in total they need less than $2N\sqrt{N}$ shifts. Q.E.D.

Next we give an analysis of the average case performance of the algorithm when scheduling configurations by criterion 5. Our goal is to show that the average total costs of inserting and configuring are both O(N), which is perhaps a bit surprising. We admit that the analysis is approximate and suggest how it could be improved.

We relate the algorithm to hashing with linear probing, which is analyzed in Knuth [1,518-521]. We first assume that, after configuring, the values are uniformly distributed in the even positions in the array. We next assume that each of the values to be inserted has the same chance of being placed at any odd position of the array -- the odd positions initially being gaps. We realize this assumption is suspect since the positions of values are changed during an insertion. It does, however, yield a model that is simple enough to work with. It means that we may deal with only the initial gaps, omitting the values in the even array positions completely. If a sequence of r gaps are filled, then 2r+1 values occur without a gap in the original problem. Finally, we consider that a value shifted past the high end of the array is inserted at the beginning, causing a cyclic shift. This makes the model less efficient than the real problem; it encourages denseness at the low end, while in the real problem a shift past the high end reduces density by introducing a new gap (see Section 3).

Thus we model the analysis of inserting N (say) uniformly distributed values into the sorted array by the analysis of hashing N uniformly distributed values, with linear probing used to handle collisions. Knuth [1,527-530] shows that the average number of shifts necessary to insert a new value is :

$$\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right) \qquad \alpha=g/n$$

where α is the load factor or density, g is the number of gaps filled, and n is the initial number of gaps. Translating this back to the sorting problem, and letting $\alpha=(n+g)/2n$ be the density of the array, to insert a new value will cause on the average:

$$1+\left(\frac{1}{1-\alpha}\right)^2 \text{shifts.}$$

Thus, if the table is 3/4 full -- i.e., half of the gaps are filled -- the average number of shifts is 5.

Asuming the model we propose, an algorithm that configures before the table is α full, for $1/2 < \alpha < 1$, will spend only $O(N)$ total time shifting to sort N values.

Now let us consider the expected time for configuring. Knuth[1] gives a formula $P_{k,c}$ for the probability that exactly $k+1$ shifts will be needed on the c+1-st insertion in a hash table of size M:

$$P_{k,c} = M^{-c}(g(M,c,k)+g(M,c,k+1)+\ldots+g(M,c,c))$$

where

$$g(M,c,k) = \binom{c}{k} (k+1)^{k-1}(M-c-1)(M-k-1)^{c-k-1}$$

Using this, for a given s, one can readily determine a formula for:

> the expected number of insertions i into an empty hash table of size M until a shift of s or more values is needed for an insertion.

We would like to show that if $s=\sqrt{M}$ then $i=\alpha M$ for some $0<\alpha<1$, because application of Theorem 2.2 would then yield the desired expected time $O(N)$ for configuring. Unfortunately, this analysis has eluded us and instead we have settled for the following. We will show that:

> if the hash table of size M is half full, then the probability of needing more than $\sqrt{M}$ shifts in the next insertion is small -- of order $\lesssim 2^{-\sqrt{M}/4}$.

The probability of needing more than $\sqrt{M}$ shifts on the $(M/2+1)$-st insertion is:

$$\sum_{k=\sqrt{M}}^{M/2} P_{k,M/2}$$

$$= M^{-M/2} \sum_{k=\sqrt{M}}^{M/2} (k-\sqrt{M}+1)\ g(M,M/2,k)$$

$$= M^{-M/2} \sum_{k=\sqrt{M}}^{M/2} \frac{(k-\sqrt{M}+1)(M/2)!(k+1)^{k}(M/2-1)(M-k-1)^{M/2-k-1}}{(k+1)!\ (M/2-k)!}$$

Applying Stirling's formula:

$$n! \sim \sqrt{2\pi}\ \frac{n^{n+1/2}}{e^{n}}$$

and simplifying yields:

$$\frac{e(M/2-1)\sqrt{M/2}}{\sqrt{2\pi}} \sum_{k=\sqrt{M}}^{M/2} \frac{(k-\sqrt{M+1})}{(k+1)^{3/2}(M/2-k)^{3/2}}\ 2^{-M/2}\left(\frac{M-k-1}{M/2-k}\right)^{M/2-k-1}$$

The term $2^{-M/2}$ we write as

$$2^{-(k+1)/4}\ 2^{-3(k+1)/4}\ 2^{-M/2-k-1}$$

This rewriting, along with Lemma 2.4 given below, allows us to bound the desired sum from above by:

$$\frac{e(M/2-1)\sqrt{M/2}}{\sqrt{2\pi}} \sum_{k=\sqrt{M}}^{M/2} \frac{(k-\sqrt{M-1})}{(k+1)^{3/2}(M/2-k)^{3/2}}\ 2^{-(k+1)/4}$$

which simplifies to:

$$\frac{e^{2}(M/2-1)\sqrt{M/2}}{\sqrt{2\pi}}\ 2^{-\sqrt{M}/4} \sum_{k=\sqrt{M}}^{M/2} \frac{k-\sqrt{M}-1}{(k+1)^{3/2}(M/2-k)^{3/2}}\ 2^{-(k-\sqrt{M}+1)/4}$$

Each of the terms in the sum is bounded from above by $2/M$ so that the sum itself is bounded by 1. Hence the probability that more than $\sqrt{M}$ shifts will be needed is bounded by:

$$\frac{e^2\,(M/2-1)\sqrt{M/2}}{\sqrt{2\pi}}\,2^{-\sqrt{M}/4} < \frac{e^2 M^{3/2}}{\sqrt{2\pi}}\,2^{-3/2-\sqrt{M}/4}$$

$$= \frac{e^2}{\sqrt{2\pi}}\cdot 2^{-3/2-\sqrt{M}/4+(3/2)\log M}$$

$$= O(2^{-\sqrt{M}/4}).$$

Lemma 2.4 Given $M>1$ and any k such that $0\le k\le M/2$:

$$2^{-M/2-k-1}\,2^{-3(k+1)/4}\left(\frac{M-k-1}{M/2-k}\right)^{M/2-k-1} < e$$

Proof Direct inspection with $k = M/2$ and $k = M/2-1$ yields the desired result. Consider $0\le k<M/2-1$. Substituting $k = M/2-t-1$, so that t satisfies $0<t\le M/2-1$, and rearranging yields, for the left hand side of the inequality,

$$\left(\left(\frac{1}{2}\right)^{\frac{3M/2+t}{4t}}\frac{t+M/2}{t+1}\right)^{t}$$

which has the form $(\sigma\,(t+M/2)(t+1))^{t}$. The result follows if we can show that, for $0<t\le M/2-1$,

$$\sigma\frac{(t+M/2)}{(t+1)} \le 1+\frac{1}{t}$$

since $(1+\frac{1}{t})^{t} < e$ for all integers t. Thus we must show that:

$$\sigma\frac{(t+M/2)}{(t+1)} - \frac{(t+1)}{t} \le 0$$

or

$$\frac{t(t+M/2)}{(t+1)^2} - \frac{1}{\sigma} \leq 0.$$

Since

$$\frac{t(t+M/2)}{(t+1)^2} < \frac{t(t+M/2)}{t^2}$$

it suffices to prove:

$$\frac{t(t+M/2)}{t^2} - \frac{1}{\sigma} = \frac{t+M/2}{t} - 2^{(3M/2+t)/4t} \leq 0.$$

For $t=M/2$ the expression

$$\frac{t+M/2}{t} - 2^{(3M/2+t)/4t} \qquad (4)$$

has the value 0. We will show that (4) is an increasing function in the range $0<t\leq M/2$. Taking the derivative of (4) with respect to t yields:

$$-\frac{M}{2t^2} + \frac{3M \ln 2}{8t^2}\, 2^{(3M/2+t)/4t}$$

which in the range $0<t\leq M/2$ has the same sign as:

$$-\frac{1}{3 \ln 2}\, 2^{7/4} + 2^{3M/8t}.$$

This clearly is a decreasing function of t. In the range $0<t\leq M/2$ it is positive, which means that (4) is increasing in the range $0<t\leq M/2$ and the lemma is proved.

3.A <u>stable sort algorithm</u>

Given an array a[0:2*N+1] where $N \geq 1$ and a[N+1:2*N] contains a set of values $V_1,\ldots,V_N$ the algorithm given here sorts a[N+1:2*N]. Upon termination:

> a[1:m] contains a sorted permutation of $\{V_1,\ldots V_N\}$ together with up to N "gaps" and $m \leq 2*N$

The sorted sequence can be recovered easily in linear time.

To facilitate searching, a gap contains the value of the array element to its right. Thus for a gap a[i], a[i] = a[i+1]. The algorithm uses a boolean function isgap(i) with the meaning "a[i] is a gap" for $0 \leq i \leq m+1$.

<u>Remark</u> If the values $V_1, V_2, \ldots V_N$ are positive, the sign bit in each element can be used to represent the function isgap; negative numbers represent gaps. If the V_i are all distinct then no extra bit is necessary and isgap(i) will always have the value
i > m <u>or</u> (i $\leq$ m <u>and</u> a[i] = a[i+1]). See below. <u>end of remark</u>

The variables used are:

n, $1 \leq n \leq N$:	the number of V_i already sorted. Those not yet processed are in a[N+n+1:2N].
m, $n \leq m \leq 2N$:	the sorted values $V_1, \ldots, V_n$ are in a[1:m] with at most one gap between each adjacent pair.
r, $r^2 \leq n$:	just after a configuration, r is an approximation to the square root of n.
shifts :	the length of the last run of values detected during insertion.
maxn :	min(N,1+floor(αm)) for predetermined α, $1/2<\alpha<1$.

To make more precise the contents of array a we give the following invariant P (the bounds on variables are given above). Note that there is a delimiter at each end of a[1:m], which eliminates special tests of boundary conditions. Note also that gaps can only occur in odd positions a[1], a[3], ...

P:

$\{V_1, \ldots, V_n\} \subset$ a[1:m] <u>and</u> m-n values in a[1:m] are gaps;

$\{V_{n+1}, \ldots, V_N\}$ = a[N+n+1:2*N];

isgap(m+1) <u>and</u> <u>not</u> isgap(0);

(Ai: $1 \le i \le m$: even(i) => <u>not</u> isgap(i));

(Ai: $1 \le i \le m$: isgap(i) => a[i] = a[i+1]);

(Ai: $1 \le i \le m$: a[i] $\le$ a[i+1]);

The invariant of the main loop is P1 below. The third conjunct indicates that if k≠N there is exactly one gap between each pair; the array has just been configured:

P1: P <u>and</u> $r^2 \le n < (r+1)^2$ and (n = N <u>or</u> 2*n = m).

The program:

```
a[0],a[1],a[2]:= a[N+1],a[N+1],a[N+1];
isgap(0),isgap(1),isgap(2),isgap(3):= false, true, false, true;
n,m,r,shifts,maxn:= 1,2,1,0,min(N,2);
{P1}
do n ≠ N ->
   {P1}{P}
   do n < maxn and shifts ≤ 2*r ->
         n:= n+1; x:= a[n+N];
         insert x into a[1:m] to reestablish P
   od;
   {P}
   if n < N -> configure [] n ≥ N -> skip fi
   {P1}
od
{P1 and n ≥ N}
```

Algorithm configure is given below. The purpose of the first loop is to spread out the n values in a[1:m] into a[1:2*n] so that one gap precedes each value. In the loop, a[1:i] contains the portion still to be configured while a[j+1:2*n] contains the already configured portion with each odd position a gap. Let g be the number of gaps in a[1:i]. Then i-g is the number of V_k in a[1:i]. Part of the invariant is:

$$0 \le 2*(i-g)=j \text{ and } i-g \ge g \text{ and not } isgap(i).$$

Hence i≤j. When i=j we have i=2*g, which means that there are just as many gaps as values in a[1:i] and a[1:i] is configured.

configure:

```
{P}
i,j:= m,2*n;
do i ≠ j ->
    a[j-1],a[j]:= a[i],a[i];
    isgap(j-1),isgap(j):= true,false;
    i,j:= i-1,j-2;
    do isgap(i) -> i:= i-1 od
od;
m,shifts,maxn:= 2*n,0,min(N,1+floor(α*2*n));
isgap(m+1):= true;
do (r+1)^2 ≤ n -> r:= r + 1 od
{P1}
```

The algorithm to insert x into a[1:m] has three main steps. First the position j where x belongs is determined. The invariant for the binary search loop is:

$$1 \le h < j \le m \ \underline{\text{and}}\ a[h] \le x < a[j]$$

and the loop terminates with a[j-1]≤x<a[j]. This means that upon termination a[j-1] is not a gap and that x belongs in position a[j].

The second step shifts the sequence a[j...] upwards, changing x as it progresses, until a gap is found. At the same time the number of shifts required is counted. The final step stores x in its place. Note that if x belongs after a[m], m is increased by 2 to include both the new value and a new gap.

```
Insert x into a[1:m] to reestablish P:
  Find the position a[j] to place x:
        if a[m]≤x -> j:= m+1
        [] a[1]>x -> j:= 1
        [] a[1]≤x<a[m] ->
                 Find j such that a[j-1]≤x<a[j] by binary search:
                    h,j:= 1,m;
                    do h+1≠j ->
                           e:= (h+j) div 2;
                           if a[e]≤x -> h:= e
                           [] a[e]>x -> j:= e
                           fi
                    od
       fi;
  shifts:= j;
  do not isgap(j) -> a[j],x := x,a[j]; j := j+1 od;
  shifts:= j-shifts;
  Insert x at a[j]. Note that j is odd and j≠m:
     if j<m -> a[j]:= x
     [] j>m -> m:= m+2; maxn:= min(N,1+floor(α*m));
              a[m],a[m-1]:= x,x;
              isgap(m-1),isgap(m),isgap(m+1):= true, false, true
     fi
```

4. Practical table management using controlled density

The algorithm of Section 3 can be modified to delete as well as insert values. This yields an algorithm that supports the operations INSERT, DELETE, SEARCH, and MIN as defined in [3].

Care is required to maintain gaps, because there may be more than one gap in a row after a deletion. Let us represent a gap by the closest real value to its right, so that each gap sequence looks like:

? , y , y , y , ..., y , y

not isgap, isgap, isgap, isgap, ..., isgap, *not* isgap

Insertion, as shown earlier, causes no problems because it changes only a leftmost gap into a value. To maintain gaps during a deletion, however, all the gap values to the left of the value being deleted must be changed. This takes time proportional to the number of gaps to be changed. Hence configuring should take place whenever a gap sequence of length $2\sqrt{n_i}$ or a value sequence of length $2\sqrt{n_i}$ is detected. Note that configuring is more complicated because it could cause the portion a[1:2n] to shrink.

5. A class of table management algorithms

The technique of interspersing gaps with data values can be extended to a recursive construction with some fixed number d (say) of levels.

The resulting data structure supports INSERT, DELETE, MIN, and SEARCH in total time $\leq O(N^{1+1/d})$ for N operations. In this section we describe a workable implementation of this idea and give an informal worst case running time analysis. The practical utility of these algorithms is questionable for $d>2$.

The data structure consists of a one dimensional array A containing data and gaps, together with d-1 arrays of indices into A. The array A is partitioned into segments, segments into subsegments, etc., with the array itself regarded as a level 0 segment. For n values and $0 \leq k < d$ a level k segment satisfies:

(1) the level k segment has $n^{1/d}$ data areas alternating with $n^{1/d}$ gap areas;

(2) a gap area has length $\leq n^{(d-k)/d}$;

(3) a data area is of length $\geq (d-k)n^{(d-k)/d}$ and contains $n^{1/d}$ level k+1 segments;

(4) the beginning of the i-th data area is indexed by $da^{(k)}[i]$ and the beginning of the i-th gap area is indexed by $ga^{(k)}[i]$;

(5) a gap area is never empty.

The smallest segments are on level d and are contiguous sequences of data values of length $n^{1/d}$. Note that the sizes of segments are balanced so that the number of subsegments in a segment is always $n^{1/d}$. This choice of constant ratio across all levels is optimal for this arrangement of segments.

As in the earlier algorithm the data area is sorted and a gap is a marked copy of the value following it. A SEARCH operation, then, can be done in time $O(\log dn) = O(\log n)$ for n items stored in the table. An INSERT operation determines by SEARCH the proper location for the new item, then searches forward for the closest gap -- by the arrangement of the data structure this takes time at most $O(n^{1/d})$. If insertion of the item does not cause a level d-1 gap area to be filled, the insertion is finished; if a level d-1 gap area is filled, the level d-1 segment containing the new item is configured using $n^{1/d}$ space from the gap area of the containing level d-2 segment. If taking the space from the level d-2 gap area uses up this gap area then the level d-2 segment is configured using space from the containing levle d-3 segment , etc. All this manipulation references and modifies the pointer arrays da and ga.

A configuration at level k costs $O(N^{(d-k)/d})$. In a sequence of N operations a configuration at level k occurs no more often than once every $O(N^{(d-k-1)/d})$ steps, so can happen at most $O(\frac{N}{N^{(d-k-1)/d}})$ times. The total cost for N operations chosen from MIN, SEARCH, and INSERT is then

$$\leq O(\ N(\log(dN)\ +\ N^{1/d})\ )\ \ +$$

$$\sum_{k=0}^{d-1} O(\ N^{(d-k)/d}\ \ \frac{N}{N^{(d-k-1)/d}}\ )$$

$$\leq O(N^{1/d}) + O(dN^{1+1/d}) = O(N^{1+1/d}).$$

As discussed for the earlier algorithm a deletion can be programmed in several different ways but any reasonable choice should not disturb the asymptotic bound.

6. Empirical results

Versions of our algorithm were compared with Heapsort , Quicksort and a balanced tree scheme taken from [4]. All programming was done in the C programming language running under UNIX on a PDP-11. The times reported here were obtained with the UNIX "time" command and should be accurate to a resolution of about 0.1 seconds. Test data described as random was obtained from the UNIX library "rand" function. Each number reported for random input represents an average over ten trials.

The Quicksort progam taken from [5] was programmed recursively -- an iterative version might run still faster. The balanced tree data structure was implemented with an array -- again a pointer implementation might run faster.

As described above, the controlled density algorithms schedule configurations by two conditions:

(1) density of data in array $\geq \alpha$;

(2) shift costs more than square root of size of array at last configuration.

so α is a parameter of the algorithm. The storage required by the algorithm is always $\leq$ 2N but is typically much less -- the maximum result of storage used is called M and is reported along with the running time.

Empirical Results

Sorting Algorithms

Random Data N	Heapsort time*	Quicksort time	Controlled Density alpha=2/3 time	alpha=2/3 M	Controlled Density alpha=3/4 time	alpha=3/4 M
5 000	4.8	0.9	1.7	9 234	1.7	8 264
7 000	7.8	1.3	2.4	11 813	2.3	12 789
Sorted Data N						
5 000	5.2	11.5**	10.2	9 902	10.3	9 898
7 000	8.6	22.8	16.9	13 808	17.1	13 804

Balanced Tree Comparison

	Balanced Tree time	Controlled Density time	Controlled Density M
Random Data N=4 000	8.8	2.0	6 592
Sorted Data (decreasing) N=4 000	13.0	9.0	7 830
Sorted Data (increasing) N=4 000	13.1	1.5	6 732

* All times in seconds.

** Probably due to unsophisticated choice of partition element.

7. Comments

(1) This class of algorithms was suggested by our work with some fast Turing Machine constructions [5], [6].

(2) These algorithms would seem to be a particularly attractive alternative to balanced trees in programming languages that do not allow easy manipulation of linked structures and pointers. If a link and an array entry take the same amount of storage then the controlled density array has less than half the space overhead of a balanced binary tree.

(3) Balanced trees are very flexible data structures and other operations from [3] such as SPLIT and MERGE are hopelessly awkward with arrays.

(4) The periodic configurations are expensive. This would discourage use of the controlled density algorithms when some bounded "response time" is required.

References

[1] Knuth, D.E., The Art of Computer Programming, Vol 3 (Sorting and Searching), Addison-Wesley, Menlo Park, California, 1973

[2] Perl, Y., A. Itai, and H. Avni. Interpolation search - a loglogn search. CACM 21 (July 1978), 550-553.

[3] Aho, A., J. Hopcroft, J. Ullman. The Design and Analysis of Computer Algorithms, Addison-Wesley, Menlo Park, California, 1974.

[4] Guibas, L. and R. Sedgewick. A Dichromatic Framework for Balanced Trees. 19th Annual Symposium on Foundations of Computer Science, October 16-18, 1978, 8-21.

[5] Sedgewick, R. The Analysis of Quicksort Programs. Acta Informatica, Vol. 7, Fasc. 4, 1977, 327-360.

[6] Melville, R. An improved simulation result for INK bounded Turing machines. Technical Report 78-348, Department of Computer Science, Cornell University, Ithaca, New York.

[7] Hennie, F. and Stearns, R. Two tape simulation of Multi-tape Turing Machines. JACM Vol. 13 No. 4 (Oct. 1966) 533-546.