The A-tree - a Simpler, More Efficient B-tree

Alistair Crooks, atree@alistaircrooks.com

29th September 2004

Abstract

Traditionally, searching for text is accomplished by using a data structure fitted to the task, and database access methods have tended to standardise on B-tree multi-way trees, especially where proximity searching may be desired.

B-trees were discovered in 1970. Recent advances in processor speeds and memory speeds have resulted in a very high speed for memory to memory copying, and for a much larger set of information to be held in memory at any one time. Some of the restrictions and constraints which were in place when B-trees were discovered are no longer in place.

This paper describes a new method of data organisation for searching, the A-tree, and explains the background and reasons for its design and implementation. Performance characteristics of different multi-way trees are examined and discussed.

Introduction

Advances in Memory and Processor Speed

1970

In 1970, the multi-way B-tree was discovered by [Bayer1972], and independently at about the same time by M. Kaufman [unpublished]. In 1971, the Intel 4004 was released by [Intel1971], with a clock speed of 108 KHz, able to address 1 KB of program memory and up to 4 KB of data memory. [IBM1971]shipped its first 370 in 1970, and it typically shipped with 1 MB of core memory, and a roomful of disk drives, probably totalling 200 MB. In 1973, the big mainframe [IBM1970a]disk drive was model 3330-11: 400 MB for $111,600 or $279/MB.

2004

In 2004, only 35 years later, commodity microprocessors are omni-present in datacentres, on desktops and in embedded work. Disk sizes of 160 GB and greater are common and cheap, and we are seeing disk sizes of 400 GB appear as commodity items. Processor speeds, admittedly taking advantage of clock multiplication, are at 3.4 GHz, with faster CPUs expected, and memory sizes of more than 1 GB are normal.

[Economist2004]summed up the differences between the PDP-7 which Ken Thompson used to develop the early versions of Unix as:

> ...it is necessary, though difficult, to recall just how comparatively primitive the state of computing was 30 years ago. The first version of Unix was written by Dr Thompson for the PDP-7, a computer made by the Digital Equipment Corporation, which cost a mere $72,000, and came with eight kilobytes of memory, and a hard disk a bit smaller than a megabyte. By contrast, a desktop computer today typically costs a hundred times less, has roughly 64,000 times as much memory and a hard disk 40,000 times as big.

Implications of Progress

Between 1970 and 2004, processor speed, on-chip caches and main memory have all been speeded up in a manner which would have been inconceivable when Knuth wrote his seminal work on sorting and searching in 1973 [Knuth1973]. A memory to memory copy on a relatively slow processor such as the ARM is now regularly achieving speeds of well over 1 Gigabyte per second for both aligned and unaligned copies under the NetBSD operating system [NetBSD2004].

It's also necessary to put the era in perspective: in 1970, man had just landed on the moon, Unix was undergoing an internal rewrite within Bell Labs to add pipes, and to use a high-level language like C, neither Microsoft nor Apple Corporations had yet been founded, and the teletype was the usual interface to a multi-user computer. Unix was five years away from making its way into the educational community. Kurt Cobain was 3 years old, it would be 17 years until Joss Stone would be born, and Bill Gates was 15 - Windows 3 was still 20 years away. The troubles in Northern Ireland were only just starting, and the USA was still fighting a war in Vietnam. Usually, mainframe computer systems were large, multi-user systems, maybe with magnetic drum memory, and certainly in their own area in a data center.

To jump forward just 34 years, CPUs and memory are strikingly inexpensive, and disk space has grown to the state where we are no longer being asked by friendly system administrators to clear up unwanted files, and desktop and laptop computers (undreamt of in 1970) are now much faster than the largest supercomputer in 1970. LCD screens are usually the user's means of interaction with a computer, and mice and windowing environments are the standard. Virtual memory is an integral part of every chip but the smallest embedded device, and even most new handheld telephones have an embedded chip inside which has an integral MMU. Peripheral speed has advanced, DMA is the standard means of transferring data from a peripheral to memory, and we have moved to 64-bit technology on a number of operating systems and platforms.

1 Access Methods

When an access method is deployed in computer systems today, there are generally two possibilities

- the B-tree
- Hashing

This paper will now add a third method

- the A-tree

and then compare and contrast their uses, benefits and drawbacks.

2 The B-tree

2.1 Description

Since Bayer & McCreight's discovery of the B-tree, a number of people have improved and extended the original description. Most of these are documented in [Comer1979]'s work. To recap, a B-tree is an example of a multi-way tree, where:

1. Every page contains at most 2n items (where n is the order of the B-tree)
2. Every page, except the root page, contains at least n items
3. Every page is either a leaf page i.e. has no descendants; or it has m + 1 descendants, where m is the number of entries in the page
4. All leaf pages appear at the same level

(see [Wirth1976])

If we take the Berkeley DB implementation of B-trees as the reference implementation for just now (version 1 of the Sleepycat db implementation is standard on the *BSD distributions, mainly due to licensing issues with later versions), a B-tree is made up of leaf and internal pages. Each page holds a number of (key, data) pairs. A traversal of the leaf elements from first to last will produce an ordered linear list of the elements of a tree. A B-tree has an order property, where the order is the number of elements which may exist in a leaf page (a B-tree of order 2 will have at least 2 entries in each leaf page, and at most 4 entries). The sparse property of B-trees leads to a number of benefits - insertion into a tree is usually simple, since, by law of averages, there will be space to insert a (key, data) pair into a B-tree page. On the rare occasions that B-tree insertion overflows a leaf page, then the insertion will overflow into neighbouring pages, and internal pages will receive new indices themselves, denoting the leading entry in the page underneath the internal page. In a similar fashion, deletion from a B-tree may mean underflow in rare cases, and so the tree will shrink in an ordered manner, again perhaps involving re-calculation of indices in internal pages higher up the B-tree.

When updating a B-tree by adding or deleting entries, in an environment where multiple access by different processes is allowed, may require the whole internal page to be locked, as well as parent pages, in case of overflow and underflow.

2.2 Overview

In all, a B-tree provides a convenient structure for ordered searches (where predecessor and successor entries can be easily found, and the whole tree itself can be traversed in order). In normal use, a B-tree will organise itself into a flat structure, given a large enough order, and searching within pages of a B-tree is done by binary searching, which proves to be extremely efficient in practice.

2.3 Extensions

In order to avoid duplication of data, B-trees are usually implemented with entries in internal pages pointing directly to their associated values - there is no need to duplicate this entry in the correct position in the leaf page, since the searching process will hit the entry in the internal page first. This is the usual method of implementation, and is also referred to in literature as a B+-tree. One side-effect of this is that the data at the leaf pages, when read off from first to last, is still ordered, but incomplete, as the internal pages now contain some of the data.

A B*-tree is a standard B-tree with the following characteristics:

1. Every page except the root page has at most m children (where m is the order of the B*-tree)
2. Every page, except for the root and the leaves, has at least (2m - 1)/ 3 children (this means that we use at least two thirds of the space available in each page)
3. The root has at least 2 and at most (2 * floor((2m - 2)/3)) + 1 children
4. All leaves appear on the same level
5. A non-leaf page with k children contains (k - 1) keys

from [Knuth1973] pages 477-478.

The B-tree can be extended to provide multi-dimensional key-searching (see Guttmann's R-trees for an example of this), by keeping a relatively small order, and storing maxima and minima values for the extra index fields in each internal page. Internal pages of R-trees hold the maxima and minima values of key fields from subsequent pages. The query optimiser can then be used to make searching more efficient for multi-dimensional searches.

B-trees were originally conceived to hold their data internally to the page - this improves locality of reference. Most modern implementations usually hold pointers to (string) keys in separate pages which are themselves cached.

2.4 Usage

B-trees are typically the access method of choice for database administrators. Searching by B-tree is almost as quick as by using hashing methods. Typically costly overflow and underflow operations happen rarely, and proximity searches are possible. Scanning whole trees in order is possible.

2.5 Drawbacks

In light of these speeds quoted above, the whole design of a B-tree seems outmoded.

- a B-tree duplicates the virtual memory hardware in user-level software, with a resulting slowdown in performance
- typical operations on B-trees, even B-trees with large orders, result in a number of calls to move small areas of memory around
- hand-crafting iterative statements in a high-level language such as C seems to be missing the point - given the right addresses to copy from and to, the CPU is much more efficient when using hand-tuned assembler routines to copy memory from one location to another, rather than trying to optimise the amount of memory to be copied, and perhaps performing that copy in a loop.
- the original B-tree, as described by Knuth, expects keys to either be integers with keys which are solely integers - strings are much more common. Knuth also briefly mentions the B*-tree, which keeps variable length strings in the leaf pages, presumably for locality of reference. This has been overtaken by time, and now most implementations of B-trees (such as the Sleepycat db implementation) use a separate cache for variable-length keys. Implementing caches on top of the underlying hardwares own caches may provide suboptimal performance in real-world applications.
- if these strings are held in the B-tree page, to provide locality of reference, then when an entry moves from one page to another upon insertion or deletion, overflow or underflow, the data must be transferred to the new page, which is time-consuming and inefficient.
  Consider, for example, the example of a file system which uses a B-tree to order a directory. If the directory entry information is retained within the B-tree page, then locality of reference is assured, and directory traversal can be achieved by just using the information in the page itself. Conversely, when adding or deleting directory entries, directory entry information may need to be copied between B-tree pages, which may be time-consuming and inefficient. If the directory information is held in a separate string table, it is unclear how that table would be organised to allow efficient searching, insertion and deletion, even allowing for efficiencies provided by reference counting - obviously a segmented, ordered approach would be best, but a recursive B-tree is, unfortunately, out of the question.
- data can be added to a B-tree in such a way as to minimise performance and maximise tree constructions times, by causing pages to overflow in a pathological way.

2.6 Illustration

Figure **??**, taken from [Wirth1976], illustrates the way a B-tree of order 2 will grow when data is added to the tree in the following order:

20
(a)

40 10 30 15
(b)

35 7 26 18 22
(c)

5
(d)

42 13 46 27 8 32
(e)

38 24 45 25
(f)

The data are specially chosen to exercise all aspects of the B-tree code, including overflow of leaf pages and internal pages.

3 Hashing

3.1 Description

By using a transformation function or "hashing function" on the key, a value is obtained - the "hash value" - and is used as a subscript to index into an array which holds pointers to the data. Hashing is a very simple access method, and only needs a hashing function and an array to be implemented. When the hash, or transformation function, of two distinct values results in the same value, we say that a collision has occurred. There are a number of methods of resolving collisions

- one way is to have the array value point to a linear list of values, external to the array. If there are a lot of insertions and deletions, this method can be costly, as linear lists have to be traversed in order
- another way is to step on by a number of slots in the array, but this approach can prove problematic if the original value is deleted (since the fact that the subscript is "busy" is used as a trigger to step on). If the array itself is full or near full, this approach can be costly

3.2 Overview

In real-world usage, the performance of hashing is proportional to the values produced by the hashing function - if these are a uniform spread across the array, then the chances are that the access method will perform well. If the array is itself too small, then performance problems will ensue. For good performance, an intelligent choice of hash function must be made.

In general use, there are a number of hashing algorithms in general use:

- the hashing algorithm used in Perl has been found to produce a good spread of values
- the simple hashing function from [Kernighan1976] is not the best, but is simple to understand
- the[FNV]hashing algorithm as found in various places - the FreeBSD 4.3 NFS code, Linux's NFSv4 code, etc, although the NetBSD project, after extensive benchmarking, concluded that there were better hashing algorithms than FNV
- the sdbm hashing code by Ozan Yigit performs well on various sample sets
- [Yigit2001]gives an excellent comparison of various hashing methods as applied to real-world uses, by using all the C identifiers in X11R4 sources, hashed onto an 8192-entry hash table and an 8209-entry hash table. More information is available at [Yigit2004] and in [Jenkins2004]

The holy grail of hashing algorithms is called the perfect hashing algorithm - one where, for every distinct input key, there is a distinct and unique hash value. Usually, these take a long time to compute, and can only be deterministic in the case of a known and pre-computed set of inputs, but there is software available which will compute the best hash function, given a known set of inputs - see [gperf2004].

3.3 Usage

For some reason, hashing is usually the access method of choice for programmers, and often is used in systems programming. FreeBSD have added hashed entries in directories (over a certain threshold) to speed up directory traversal.

4 The A-tree

4.1 Description

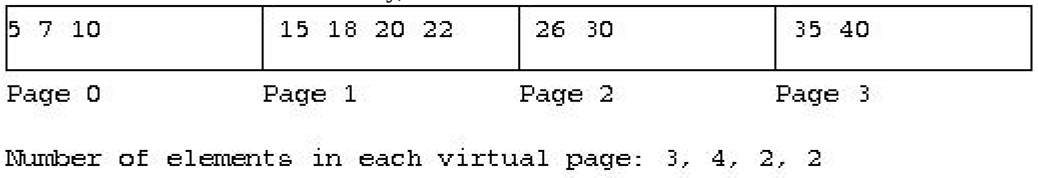
The A-tree can be thought of as an array of pointers to storage elements, grouped together logically into virtual pages. This array is sparse. The size (maximum number of storage elements) of a virtual page is called the *order* of the A-tree. An A-tree has the following characteristics:

- The array is sparse, so that there are at most *order* elements in a virtual page, and every virtual page except the first one must have at least one element. The empty A-tree has 0 elements in its first virtual page.
- If there is a positive number of elements in the A-tree, there will always be an element in the first element of a virtual page.
- Searching within an A-tree uses (conceptually) two binary searches; the first binary search is done using the first element in each virtual page, so that the correct virtual page is found, and the second binary search takes place within the virtual page, to test for the presence or absence of an element.

Our A-tree implementation uses two arrays:

- one array for the elements, which contains total tree size elements; each element is a pointer to the (*key, value*) tuple. The key and the value are both sized strings.
- one array of integers, which contains (*total tree size* / *order*) elements; each element of this array contains the size of the virtual page

Conceptually, if a B-tree had all its elements in leaf pages (i.e. a standard B-tree as described by Knuth), and its nodes were laid end-to-end in order, the resulting array would be similar to an A-tree. Pictorially, an A-tree can be envisioned like this:



With 11 elements and four virtual pages in the A-tree, a search for an element would start by finding the correct virtual page in which to search.

This is done by a binary search, using the first element of the virtual page as the key, and using two temporary bounds to find the virtual page.

For example, assuming C/C++ array subscript notation, let us search the A-tree illustrated above for the search key “10”.

The binary search would start with the higher bound at (virtual page) 3, and the lower bound at (virtual page) 0, and so the mid-point is at (3 + 0) / 2 == 1. The first element of virtual page 1 is “15”. If the search element is less than 15, then the higher bound will be set to (1 - 1) == 0, and the lower bound will stay at 0. When the two bounds are equal, or the lower bound is higher than the upper bound, the virtual page has been found.

The binary search then continues within the virtual page - the upper bound is set to (3 - 1) == 2, the lower bound to 0. Quickly the search finishes by finding the element “10” in subscript 2 of the virtual page.

4.2 Characteristics

Using the insights from the increased memory and CPU speeds as a base, and observing some of the characteristics of both B-trees and hashing, it was possible to come up with the basic characteristics for A-trees:

- a large array of pointers to key and value tuples is kept
- this array is sparse
- the array is conceptually split up into virtual pages - in reality, a separate array of “number of items in a virtual page” is kept
- each virtual page must have at least one element in the virtual page
- binary searching in two stages:

   1. finding the correct virtual page by using the first element in virtual pages, and then
   2. another binary search within the virtual page, used to locate elements

- fast memory to memory copying is used when inserting elements into the A-tree. If an element can be inserted into a virtual page, any elements which need to be moved up the virtual page will be moved. If there is no room in the page, an oscillating search is used to find the nearest virtual page which has room, and elements are moved up or down the A-tree accordingly, to make space for the element to be inserted.
- when deleting elements, the same fast memory-to-memory copying is used to move any elements in the virtual page down. If underflow occurs, the virtual pages are all moved down 1, again using the fast memory-to-memory copy. It does not matter if unused elements are copied, since this overhead is vastly outweighed by the speed in using memmove(3).

The A-tree itself is implemented by using a resizable sparse array of elements, and a smaller array of virtual page sizes is kept - the size of this array is the total size of the array of elements divided by the order of the A-tree.

4.3 Searching

When searching an A-tree, the following pseudo-code is used:

1. calculate low and high virtual page subscripts, and from these, the mid-point virtual page subscript
2. compare the first element of the mid-point virtual page to the search element, and modify low and high virtual page subscripts accordingly. Repeat steps 1 and 2 until the search converges on a single virtual page (in which the search element either resides, or would reside, if it was present in the tree).
3. within the virtual page, calculate the low and high indices for elements
4. perform a binary search within the page to find the desired element

Sample C code for searching for the virtual page and the position within it is shown below:

```
/* do a binary search in the virtual page */
static int
binsearch(atree_t *ap, const DBT *key,
          int *pg,int *lo, int *hi)
{
    *lo = 0;
    *pg = -1;
    if ((*hi = ap->virtpgsizec - 1) < *lo) {
        return 0;
    }
    for (;;) {
        do {
            int mid = (*lo + *hi) / 2;
            int cmp;
            cmp = (*ap->info.compare)
                   (key, &KEY(ap,
                          (PAGE_FOUND(*pg)) ?
                          mid :
                          SUB0(ap, mid)));
            if (cmp <= 0) {
                *hi = mid - 1;
            }
            if (cmp >= 0) {
                *lo = mid + 1;
            }
        } while (*lo <= *hi);
        if (PAGE_FOUND(*pg)) {
            return (*lo - *hi == 2);
        }
        /* normalise the page number */
        if ((*pg = (*lo - *hi == 2) ?
             *hi + 1 : *hi) < 0) {
            *pg = 0;
        }
        *lo = SUB0(ap, *pg);
        if ((*hi = *lo + ap->virtpgsizev[*pg]
             - 1) < *lo) {
            return 0;
        }
    }
}
```

4.4 Insertion

When adding an element to the A-tree, the following steps take place:

1. if the tree is full, then resize (by doubling) the main array, and the array of virtual pages. After resizing, make the array more sparse, by moving half the elements out of each virtual page into the next virtual page

2. perform a binary search to find the correct page and position within the page to insert the new element. If the element is already in the tree, do not attempt to insert it.

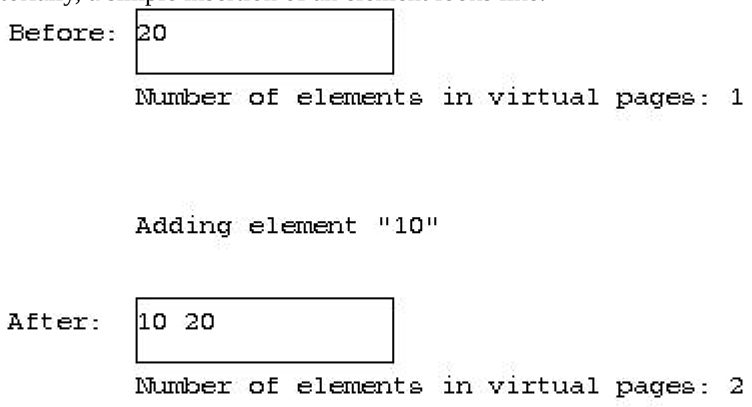
3. mark the A-tree as “updated”

4. if there is no room in the virtual page, then perform an oscillating search to find the nearest page which has room. Assuming the current page is *i*, the order of the page search will be (*i* + 1), (*i* - 1), (*i* + 2), (*i* - 2), etc until room is found. When room has been found (and there must be room to insert an element, because of Insertion(1)), the elements are moved inter-page using memmove(3). No attempt is made to optimise the number of elements to move - every element in the desired range is copied.

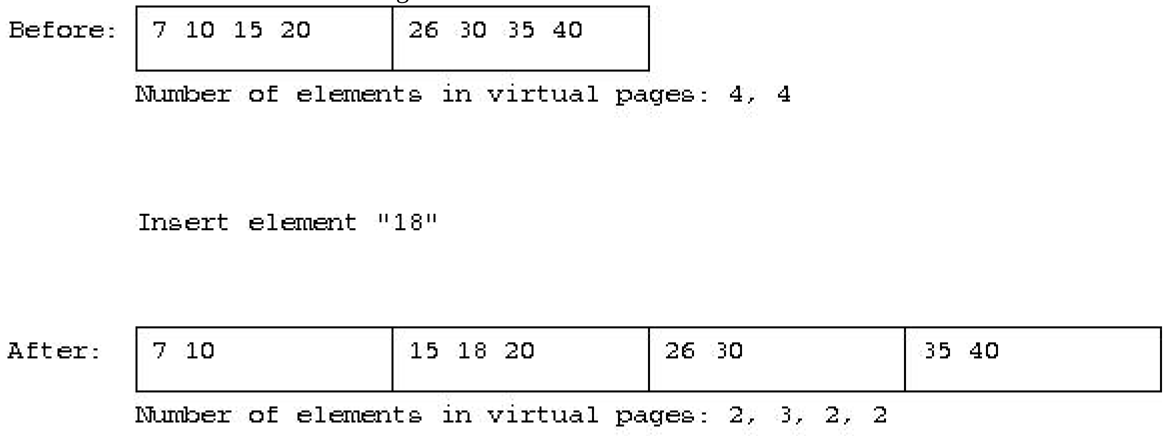
5. the necessary elements in the page are moved up one to make room for the element to be inserted

6. the element is inserted in the A-tree

Pictorially, a simple insertion of an element looks like:


Before:
20
Number of elements in virtual pages: 1
Adding element "10"
After:
10 20
Number of elements in virtual pages: 2

and an insertion which results in the growth of the atree looks like:


Before:
7 10 15 20
26 30 35 40
Number of elements in virtual pages: 4, 4
Insert element "18"
After:
7 10
15 18 20
26 30
35 40
Number of elements in virtual pages: 2, 3, 2, 2

4.5 Deletion

When deleting an element from the A-tree, the following steps take place:

1. the element to be deleted is located using a binary search

2. the storage allocated to the element is freed
3. if there is only one element in the virtual page, each virtual page is moved down one, and each element in the main array is moved down by “order” elements
4. if there is more than one elements in the virtual page, then move the elements in the virtual page down one

4.6 Illustration

Using the same data as provided in the B-tree illustration, and with an A-tree of order 4 to simulate the same growth characteristics as the A-tree, we obtain the following picture:

It is interesting to note some of the aspects of the growth of the A-tree when it is used in conjunction with the same data as used in the B-tree illustration.

- The number of virtual pages in the A-tree grows in powers of 2
- The data in the array, although sparse, is always in increasing order
- The array containing the number of elements in the virtual pages has not been shown, but is insignificant compared to the data stored - for an A-tree with 8 virtual pages, an array of 8 integers is needed
- The virtual page size can be tuned to the underlying page size of the virtual memory subsystem, to improve performance
- The oscillating search will find the nearest page with any room. This is a much cleaner way of finding space than the “merging leafs” method of B-tree space management.

5 Comparisons

5.1 A-tree and B-tree Comparison

If we contrast an A-tree with a B-tree, the following observations can be made:

- the B-tree provides a segmented, ordered list of pages in the same way that an A-tree does
- a B-tree (not a B+-tree or a B*-tree) will have a list of leaf elements along the “bottom” of the B-tree, and is similar to an A-tree
- binary searching is performed in much the same way that a B-tree accomplishes its searches

- A-tree code is much simpler than B-tree code - especially during B-tree deletion, when underflow of pages occurs, there are many corner cases to consider, and this usually results in large object codes. In comparison, the heart of A-tree insertion and deletion is a memmove(3) call, which will be hand-optimised assembly code on most modern machines and operating systems.
- with an A-tree, an ordered walk of the tree is possible, from head to tail, or from tail to head. Proximity searches can be performed with ease.
- all kinds of B-trees segment their dataspace into pages - this is done at the user-level, rather than at the hardware level, and the abstraction wastes time and space with modern computers
- A B-tree will typically store strings as data and keys. These strings need to be stored somewhere - for locality of reference, the strings are often stored in the page of the B-tree, which reduces the amount of space which can be allocated to elements, and which can make the number of elements in a B-tree page unpredictable. This is not a problem in B-trees, since the higher-order pages provide the indices through which the correct sub-page can be located, but it can be a problem if there are many insertions into a tree, and the strings need to be moved between pages.
- an A-tree has none of the possible “pathological insertion” problems which may occur with B-trees. An A-tree’s size is governed by the number of elements in that tree.

5.2 A-tree and Hashing Comparison

When compared with a hashing, the following observations can be made:

- an A-tree will take longer to search for an element, since two binary searches are taking place using an A-tree, whilst hashing will involve calculating the hash value and then testing for its presence in the hash table, and any overflow chains, or subsequent location searches it may have to perform
- an A-tree will be as quick as hashing to insert elements into a table. In addition, hashing the key will lose information, and so no proximity searching can be performed using hashing
- both A-trees and hashing use sparse arrays to contain the elements

Future Work

There are a number of aspects of A-trees which would be good to investigate:

1. Some research must take place into the best method of growth for an A-tree - when an A-tree contains a small number of elements, doubling the size of the

A-tree is easy and efficient. When the number of elements in the A-tree grows large, the addition of a number of slots for elements in the A-tree is the best approach.

2. Extension to multi-dimensional searching, by way of an analogy of Guttman's R-trees would be a beneficial area of research. Guttman found that typical pages in an R-tree would only contain a small number of entries, and it would be interesting to contrast that with A-trees, where the information would also be held in the A-tree entry, but where fast memory copies and comparison functions could be used to good effect.
   To adapt R-tree methods of searching subsidiary keys to A-trees would involve solving the problem of "summary" information being held in higher-order pages further up the R-tree towards the root page.

Performance

The speed of searching in an A-tree is slower than when using a hashing scheme, but preserves order, which is useful in certain situations. The speed of searching an A-tree is roughly equivalent to that of searching a B-tree. Insertion and deletion from an A-tree are much simpler than the corresponding insertion and deletion from a B-tree.

Conclusions

The size of memory on today's machines, and the speed of the CPUs, busses and peripherals mean that we have come to rely on older data organisations which were conceived at the time that memory was scarce and CPUs were slow, and no virtual memory was available.

In 2004 and beyond, both disk and memory space are plentiful and inexpensive, and the constraints which existed when B-trees were first discovered are no longer in place.

In practice, the same characteristics of a B-tree apply to an A-tree - the segmented nature of its elements, which allow relatively easy insertion and deletion - are more efficiently encoded in an array, rather than as separate pages in a B-tree. Ordered searching and proximity searching are possible when using B-trees and A-trees, although not the B+-tree, which is the standard mechanism for B-tree implementation.

An A-tree will grow in size by doubling when inserting an element and the A-tree is full. The A-tree does not shrink - rather, its elements migrate towards the smaller end of the array.

A-trees and B-trees use the same two-step binary search mechanism.

If the 0-th element in a virtual page is unaccessed, it will be quite possible for the virtual memory subsystem within the operating system to reuse the memory allocated to the page, and very little overhead is associated with this.

In all, the A-tree is a much simpler alternative to a B-tree, has the same benefits, and is more performant.

References

[Bayer1972] [Acta Informatica (1972), 345 - 349]

[Intel1971] http://www.intel4004.com/

[IBM1971] http://www-1.ibm.com/ibm/history/history/year_1970.html

[IBM1970a] http://www.beagle-ears.com/lars/engineer/comphist/ibm360.htm

[Economist2004] http://economist.com/science/tq/displayStory.cfm?story_id=2724348

[Knuth1973] Sorting and Searching, Addison-Wesley, 1973

[NetBSD2004] http://www.netbsd.org/

[Comer1979] "The Ubiquitous Btree", ACM Computing Surveys, Vol 11, pp 121-137, June 1979.

[Wirth1976] Algorithms + Data Structures = Programs, Prentice-Hall, 1976

[Kernighan1976] The C Programming Language, Prentice-Hall

[FNV] http://www.isthe.com/chongo/tech/comp/fnv/

[Yigit2001] http://mail-index.netbsd.org/tech-kern/2001/11/28/0034.html

[Yigit2004] http://www.cs.yorku.ca/~oz/hash.html

[Jenkins2004] http://burtleburtle.net/bob/hash/

[gperf2004] http://www.gnu.org/software/gperf/gperf.html

[Guttman1984] "R-trees: a Dynamic Index Structure for Spacial Searching" in Proceedings of ACM SIGMOD, 1984

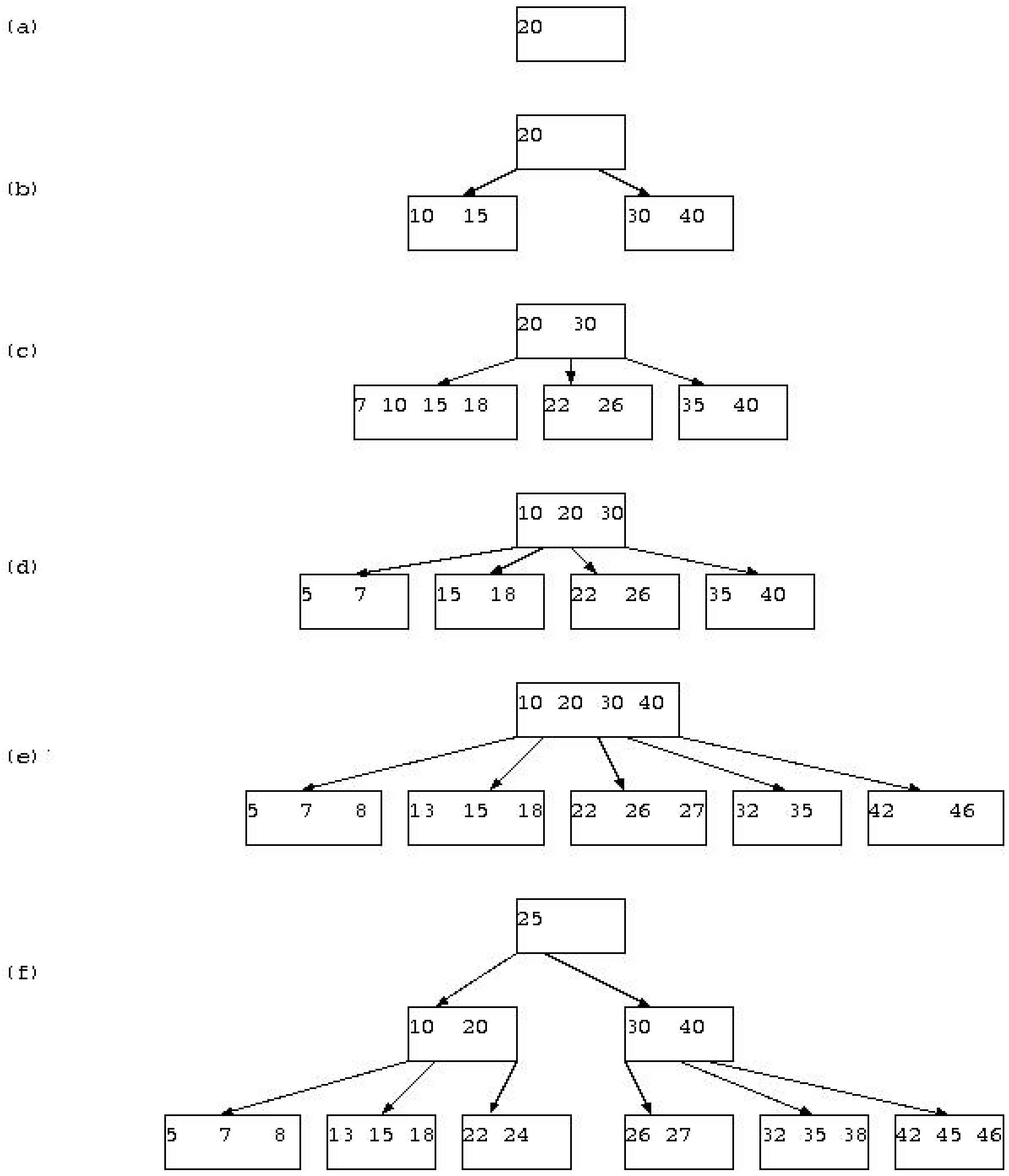


Figure 1: Addition to B-tree

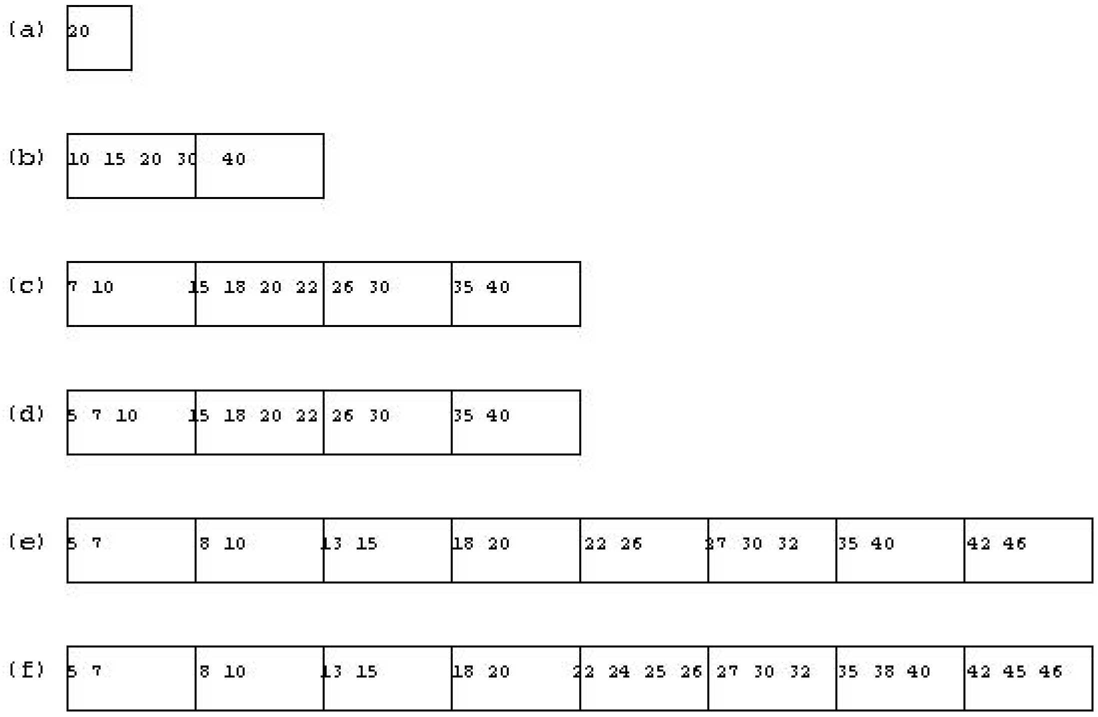


Figure 2: addition to A-tree