DataStructures

Ideas about types of data structures

John Ryland

Monday 18th April, 2022

Contents

List of Figures

1 Data Structures

1.0.1 Ideas about types of data structures

2 Introduction

Data structures are a fundamental aspect of computer programming. They are as important as code. In fact the data structure chosen informs what the code must do, not the other way around.

In this regard data structures are fundamental to any kind of pragmatic program.

Pure functions are also fundermental. For example f(x), where f(x) = x * x. Realistically a pure function when implemented on a real machine will be interacting with memory such as a stack, and a stack is itself a data structure, so sometimes the data structures are there, but hidden.

Some modern techniques also try to defer the choice of data structure to as late as possible. Consider the use of C++ container types being used in conjunction with the auto keyword. This can allow swapping out the container type rather easily allowing a change in data structure with minimal disruption to the consuming code. This is certainly a good idea in many situations. However there are plenty of things to be mindful of so I do not think this is a complete divorce of code from data structures. It reinforces the idea that choosing the right data structure is important, so important that in case you get it wrong you will want to be able to change it easily.

2.1 Some basic data structures

Let's look at some fundamental data structures:

- An array
- A linked list *(A singly linked list)*
- A doubly linked list
- A binary tree
- A tree *(A M-ary tree)*
- A dictionary *(A hash table)*

Some have alternative or more specific names, but these are some of the most common data structures that are very widely used. It is assumed that the reader is already familiar with these data structures and the relative pros and cons to them.

3 Traversal

The data structures need to be used by code to access them and update them. This is how the pros and cons of data structures are measured, how easily the various operations can be performed.

Traversal of a data structure is one of the more fundametal operations. Some data structures offer different traversals. Lets look quickly at each.

3.1 Array

An array can be traversed from start to end or end to start by incrementing or decrementing an index in to the array. Very straight forward.

3.2 Linked lists

A singly linked list can only be traversed from start to end using the chain of `next` pointers.

A doubly linked list can be traversed in either direction as there are two sets of pointers. As an aside, an xor linked-list can achieve this with one set of pointers.

3.3 Trees

A binary tree has three types of traversal. An in-order traversal, a pre-order traversal and a post-order traversal.

Consider the binary tree shown in figure 1.

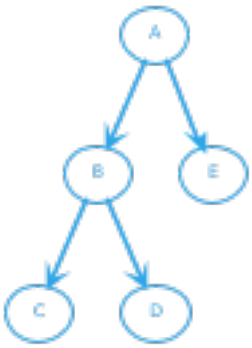

Figure 1: A binary tree

Depending on the traversal order used we will visit the nodes in the alphabetical order depicted, from A, B, C, D to E as shown in the below diagrams. In all cases we are still

dereferencing the same chain of links in the same way, but the difference between them is where the current node is visited, whether it is visited before descending the left side of the node (pre-order), between descending the left and right side (in-order), or after descending both the left and right sides (post-order).

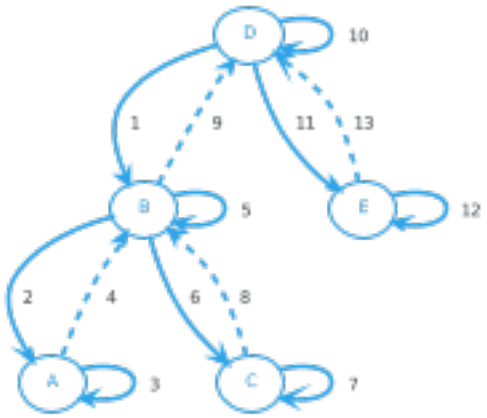

Figure 2: In-order traversal of a binary tree

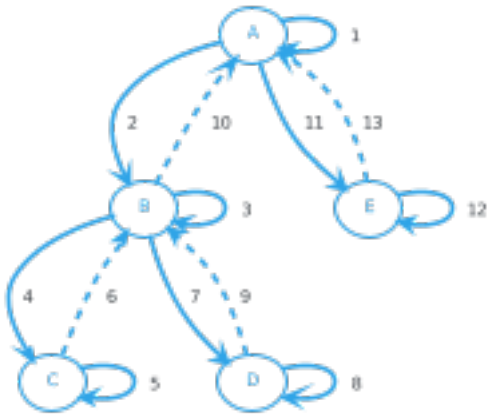

Figure 3: Pre-order traversal of a binary tree

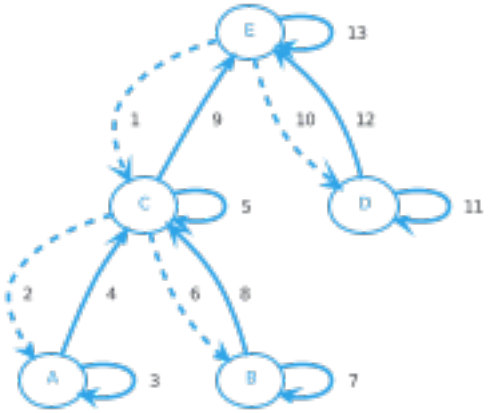

Figure 4: Post-order traversal of a binary tree

For non-binary trees where each node can have an arbitary number of children, the above pre-order and post-order traversal orders still make sense and are possible however the in-order traversal is less easy to define or useful for these trees.

3.4 Hash tables

Typical hash table implementations consist of a set of buckets with each bucket containing a singly linked list of nodes. A full traversal by iterating each bucket (in forward or backward order) and then for each bucket iterating the linked list of nodes is possible. If full reverse order was required, conceivably the hash table's buckets could contain doubly linked lists or xor linked lists.

Another form of hash table implementation is called open addressing and avoids using linked lists and attempts to only use the buckets to directly store all entries. If the number of buckets is not well chosen, either all the buckets fill up and extra time is spent probing and the table needs to be dynamically resized, or the other problem is the table is too large and the memory is sparsed populated with the data, reducing optimal cache use and wasting memory. On the positive side, it can avoid needing to make memory allocations for the nodes as they are added directly in to the buckets, and there is no dereferencing when doing a lookup, the data is directly in the bucket.

4 Searching

Another very important use for a data structure is searching for data within it. Without sorting, any data structure can trivially be searched if it supports traversal. We already saw how traversal can be supported by all the data structures mentioned so far.

Most of the data structures can be sorted to rearrange the items in to some kind of canonical order to help them be searched more quickly. For example an array can be

sorted using the C standard library function `qsort` to achieve this which then makes it possible to search efficiently using the `bsearch` library function which does so using a binary search strategy.

Sometimes just plain traversal on modern machines can be as efficient on small sets of data as more elaberate algorithms. This is due to a number of reasons. Firstly memory and it's caches are optimized for sequential usage patterns. When you access a single byte of memory, a full cache line of 64 bytes are fetched in the process. If you access only 1 byte in every 64 bytes scattered through the address space, you will only achieve 1/64th the potential of the memory's bandwidth. If you access all 64 bytes of the cache line before moving on to the next cache line you will be unlocking the full potential of that bandwidth. You might manually or the compiler or CPU might even be able to preload the next cache line in advance for you so that you don't even need to wait for memory to load in to the cache lines. This is part of the power of sequential access. Another aspect is the predictability of the code's branching. A sequential iteration through an array is a simple loop that the CPU's branch predictors have an easy time with.

However when the data gets really big then definately smarter algorithms win out. Unfortunately there is no easy rule as to what that size actually is and you might not control what size your input data is. If you know what your data will be, the best strategy is to profile. If you don't know but performance isn't going to matter when the data is small, use the smarter algorithm, otherwise perhaps do both and based on the size of the input switch implementations. There is no easy answer as to when to use which particular data structure and algorithm.

5 Insertions and deletions

The next aspect of various data structures is how well they support adding and removing new data. Sometimes this is required and sometimes not. The simplest data structure to support insertions and deletions is the linked list. I think this is still practically the first thing computer science students learn so I don't think I need to explain this. Arrays can be grown by reallocation which has it's own problems because you can't have pointers in to items in the array. A static array would be one where insertions and deletions aren't supported.

Trees and hash tables are almost as easy as lists for supporting insertions and deletions.

6 Actual implementation details

So far this discussion of data structures has been mostly theoretical. But what about some actual implementation detail considerations.

The first I want to talk about is intrusive vs non-intrusive implementations.

Figure 5: An intrusive list

An intrusive list is one where the items contained in the list themselves have the next pointers as shown in figure 5 where the `value` member and `next` member are both in the nodes.

Figure 6: A non-intrusive list

The non-intrusive list has the advantage that the item doesn't need to know it belongs to a list so can be used on arbitary data, however the downside is that additional small allocations will be needed for the nodes and additional pointer dereferences to access the data. Depending how the nodes and items are allocated, this may cause access patterns that are not well tuned with the way computers cache memory.

On the other hand, if the size of each item is large, the non-intrusive list may actually be more cache friendly for certain operations, like traversal because the size of the nodes are smaller.

6.1 Intrusive hash tables, binary trees

Once the concept of intrusive lists is understood, it is not a big jump to realize that this can be easily be applied to binary trees and hash tables. Given that a hash table is buckets of linked lists, changing those lists to intrusive lists is a no brainer. As well it should be fairly obvious how to make binary tree nodes be intrusive also.

6.2 Implementation details of m-ary trees

A binary tree is a fairly easy concept to understand once linked lists are understood. Instead of a next pointer in each node, the node contains a left pointer and right pointer. The more general case is a m-ary tree which has instead of just 2 children,

has m children. Each node might consist of a parent pointer and an array of children pointers such as in figure 7.

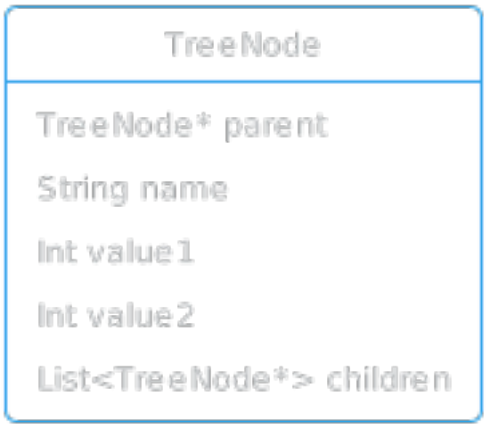

Figure 7: A tree node

This example is intrusive similar to previous examples, however contained in each node is another data structure, the list of children. This is unfortunate. The list is a list of pointers, so in effect it is a non-intrusive list.

But we can do better. It might not be immediately obvious but we can represent the same information in another way. Instead of thinking about each element as having a parent and multiple children, another way of framing this is that each element has a parent, an eldest child and a next youngest sibling. This suprisingly is the same amount of pointers a binary tree contains if including parent pointers (optional in both cases depending on your requirements). You can iterate all children by jumping to the eldest child, then from that element iterating the next youngest siblings across.

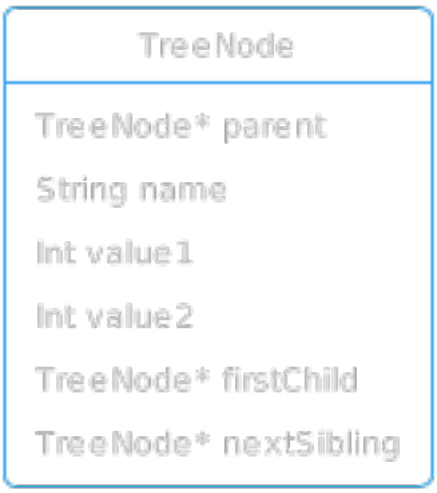

Figure 8: A better intrusive tree node

If we recall our earlier tree example in figure 1, when rearranged in this way it would appear as below in figure 9 using this alternative representation.

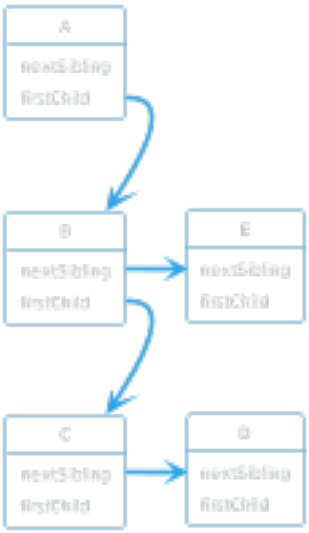

Figure 9: An intrusive tree

Space wise this is no different than a binary tree when used as a binary tree, but can be used to have nodes with arbitary numbers of children.

6.3 Cache friendly structures

As already mentioned, memory access patterns are important. Accessing memory by blocks at a time or sequentially is better than accessing small amounts of it randomly. Also if our data structures are smaller we can make better use of the available memory bandwidth.

As machines have moved from 32-bits to 64-bits, the corresponding size of a pointer has doubled from 4 bytes to 8 bytes. This makes using pointers more expensive. From this point of view one of the best data structures is the humble array and using indexes in to the array instead of pointers. Arrays are cache friendly and provided the array will not have more than 4 billion items, can be cheaper to reference items of the array using indexes.

With all of the data structures looked at so far, there is no reason that the items of these could not be allocated out of a pool instead. The concept is simple. A memory pool is simply a large pre-allocated chunk of memory. Then fixed size items can be allocated out of this pool. For example if we have a linked list, each list item could be allocated from a memory pool associated with this linked list. When it is time to delete the entire list, a single deallocation of the chunk of memory the pool pre-allocated can be made, effectively deleting all the items.

Now all the items of the list are in memory with addresses that are close together. Practically the list is almost an array. This isn't a great example as most of the time you would be better off just using an array and avoiding the complexities of needing a memory pool.

The way malloc is frequently implemented is first by pre-allocating a chunk of memory from the heap. Then a number of pools are created for different sized allocations. Then when a request for a certain sized piece of memory is made it will check the appropriate pool and walk what is called the free list to find an available piece of memory to return. Malloc also needs to be thread-safe so there will often be a mutex that needs to be locked. The free list is often just a linked list inside each of the pools linking together the free sections of memory using that free memory itself as the nodes. When an allocation is made, it adjusts the free list to jump past that allocated piece of memory. And when memory is freed it does the reverse, adding the freed memory back in to the free list.

With a memory pool impementation, it too could implement such a concept as a free list. Because typically memory pools are for fixed sized allocations, the free list only needs to take the first item it finds, it doesn't need to search for a better fitting free chunk of memory. This is wonderful as allocations can be done in constant time if the pool can be guarenteed to be created large enough for the peak number of items it ever needs to allocate. Deallocations can also be in constant time as it simply adds the item back in to the free list.

6.4 Table representations of trees

We combine this together with our better intrusive tree node, and we have a table of these nodes which represent a m-ary tree but are stored as an array. We should then replace using pointers with indexes for the space saving. So we end up with something like {#fig9}.

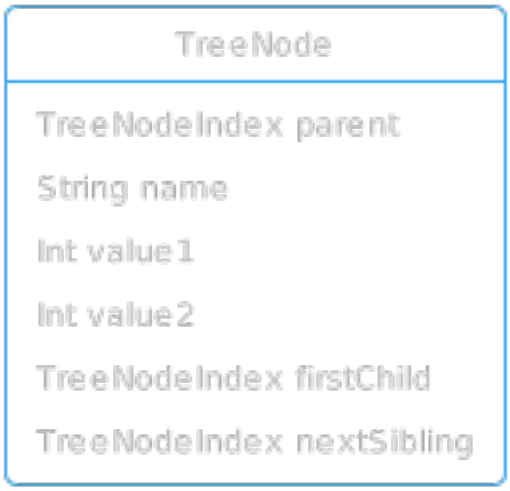

Figure 10: A pool tree node

So we can imagine these nodes inside of an array or table. Here is how to think about it:

Figure 11: Tree as a table

The references should be indexes in to the array. Lets change those.

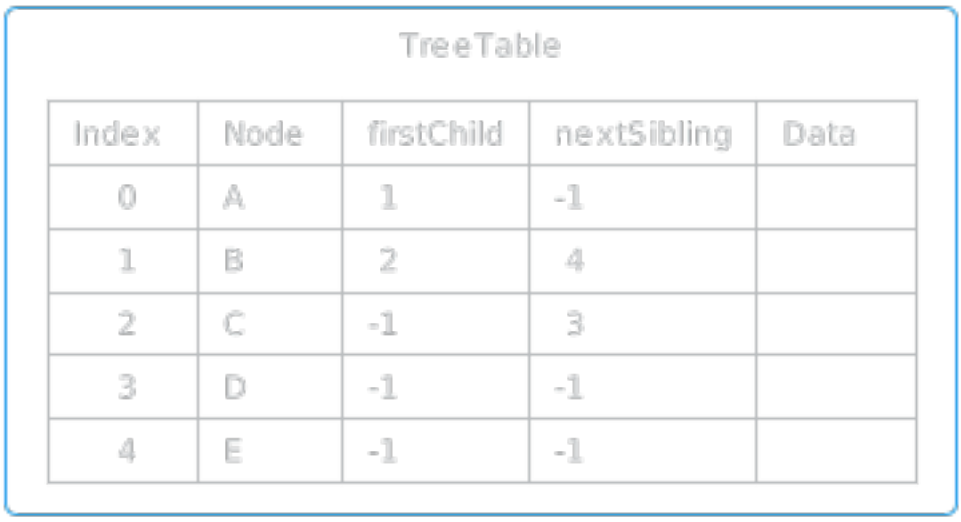

Figure 12: Tree table with illustrative indexes

The index column doesn't need to explicitly exist but can be inferred by the position in the array, but it should help follow where the firstChild and nextSibling point to.

It hopefully should be obvious that we can rearrange the rows of the table without changing the implied tree and tree order provided we correspondingly update the indexes based on any rearrangements we make. For example if we swap rows 2 and 3 and fix up the indexes like follows:

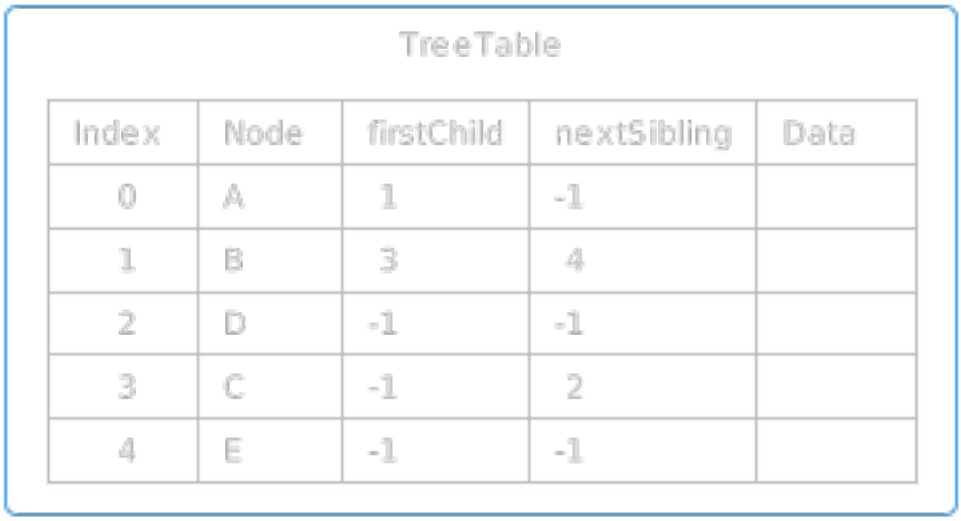

Figure 13: Tree table with swapped entries

This still represents the same tree. So if we removed a node from the tree, we could do this by simply blanking out the row of the table and adjusting the indexes that refered to it. For example if we were to delete node D from the above table / tree.

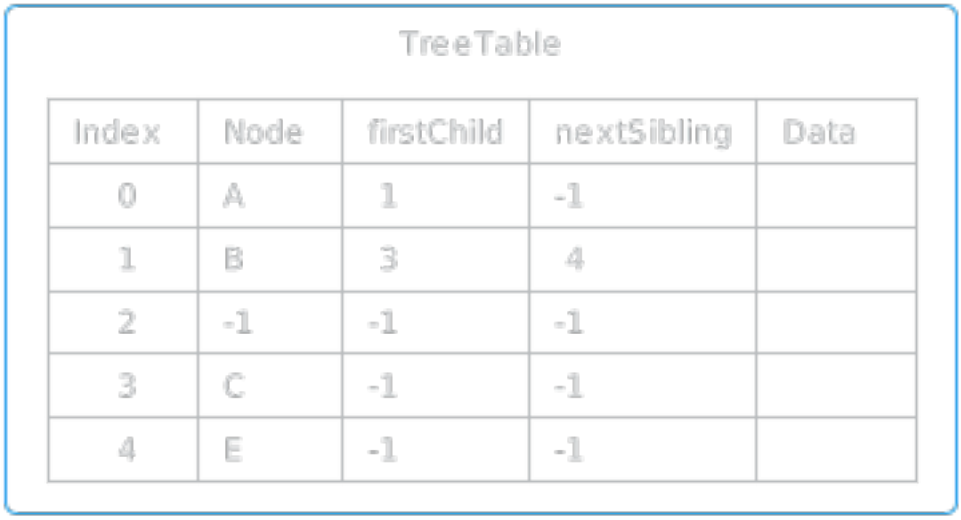

Figure 14: Tree table with removed entry

All the same tree like operations are possible, it's just a slightly different way of thinking about it. But what this does allow is when full traversal is needed it can be done in a more efficient manner.

We could add another column that marks if the node has been deleted and is in the free list and chain these blanked rows so they can be allocated from.

nextFree = 2

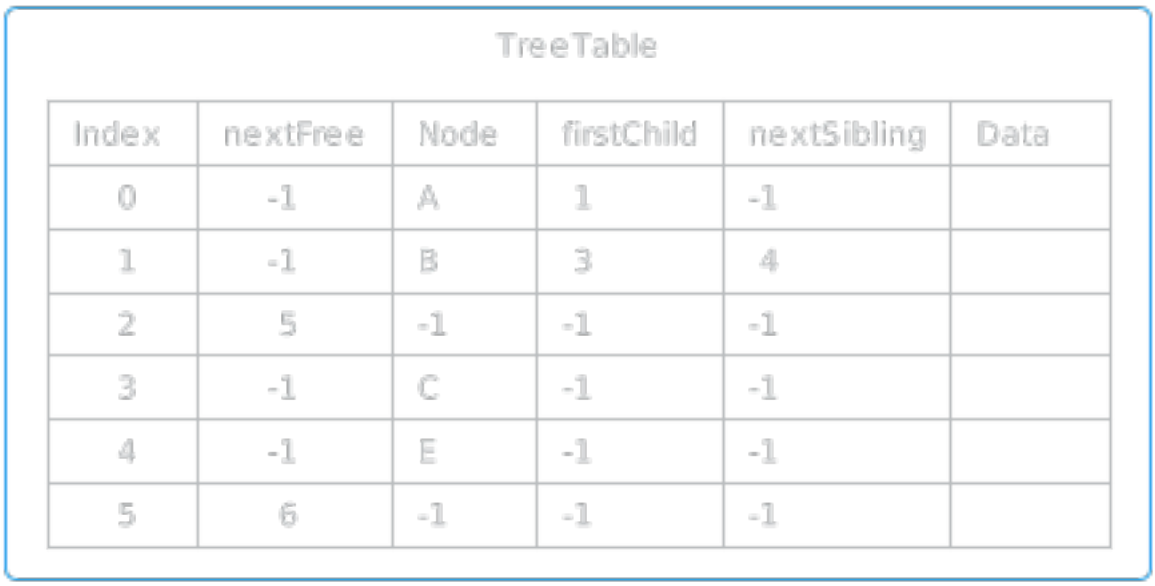

Figure 15: Tree table with free list

Alternatively, because we can swap entries in the array while preserving the integrity of the tree, we could swap the removed entry with the last entry in the array. This is the so called 'swap and pop' method of deletion.

6.4.1 Potential use cases

A conceptual tree is used widely to represent heirarchies. Heirarchies are useful for grouping and providing structure to information. One example you are using right now to view this page is HTML which is a heirarchy of elements that are arranged in a document object model, or DOM. It is the tree of elements that make up the page. The page is stored in HTML that is XML which is a way to nest elements, a textual representation of that tree.

Browsers keep this DOM in memory. Then javascript can execute which can do a number of things to it. It can search it, searching by id, tag, class etc. It can create new nodes in the DOM and manipulate the DOM in various ways. The DOM is also used to apply CSS. CSS uses selectors to determine which styles apply to which collections of elements. The selectors can be things like a given id, or something like a particular type of tag followed by another particular type of tag, or a tag that is a child of a particular other tag. For some of these, the DOM is required to determine which elements belong to the set of elements that the style applies to. Also the DOM is used for the visual representation of the page, following a box model where unless the element uses non-

default positioning, each element is visually nested in each other when they are a child of another node in the DOM.

In memory as a table structure as apposed to a traditional pointer tree, the data can be seen more as a database than as a tree. Databases can be indexed in various ways to provide fast lookups. This indexing can be similar to how we originally described sorting an array with qsort and searching it with bsearch. A side table is made with the search column and an original index column and then is sorted by the search column. Searching this table and finding a match allows reading the original index column to be able to refer back to the orignal table and find the element.

For large databases this is an efficient way to search and find one element from potentially millions of elements. However if searching from a collection of only perhaps 100 elements, and trying to find multiple matches with potentially multiple search criteria with frequently changing data, it might not be worth maintaining this kind of indexing of the data. For every field that could be searched the corresponding index table would need to be maintained when ever the tree is changed.

We haven't yet looked at hash tables in detail, but these can help us with searching our data in a way that is easier to update.

Lets just quickly look at a few other practical examples where tree structures are used. In games, often the objects in the game (commonly called GameObjects) are frequently represented conceptually in a tree. In user interfaces, the widgets are also commonly in a conceptual tree similar to how HTML's DOM of nested elements as a visual structure to a page or in the case of user interfaces of a window composed of widgets.

7 Hash tables and hashes

Hash tables are sometimes also called dictionaries, particularly if the key is a string. They can be thought of as a set of KVPs or key-value pairs. The general idea is pretty simple. If you want to look up the definition of a word, you search a dictionary for the word, and associated with the word is the definition right next to it. Imagine the dictionary was arranged in to chapters, and each chapter was for each letter and contained only words that started with that letter. These chapters would be our buckets in the hash table, and finding what the first letter of our word we are searching for to find the appropriate chapter is our rather simple hash algorithm.

We can use more complicated hashing algorithms and larger numbers of buckets to make the collection of words to search smaller to make it more efficient, but the idea is still the same. Usually a prime number of buckets is chosen for various reasons, particularly when using open addressing, this depends on the probing algorithm.

Whether using open addressing or using a linked list, the data stored is the same and it needs to be compared with what we are searching for to check if it matches or not or we need to continue probing or jump to the next item in the list.

If our key is a string, we can apply a hash to the string to condense it down to a single number. We can then use the modulus of this number with the number of buckets to

give us our first bucket to start from. Hashes can have collisions, particular after we've taken the modulus of it with another smaller number, such as N for when we have N buckets. There is a one in N chance of getting a collision. If N is 100 and we've already added 90 items to our hash table, the chance of a collision is very high. A collision represents two different items appearing to be similar or the same item. We therefore need to check. We have to do a more complete comparison to see if the items are infact identical or we have a hash collision.

In the case of strings, if our input strings have common prefixes, then when we do our complete comparison, we may have to iterate the strings and compare quite a number of characters to determine if they are the same or different. This has to be done as a last resort, but to avoid this, if a larger hash was stored with the item, like our hash value before we took the modulus of it, we could compare against it before needing to resort to a full string comparison. So our hash map item might look something like figure 16 below.

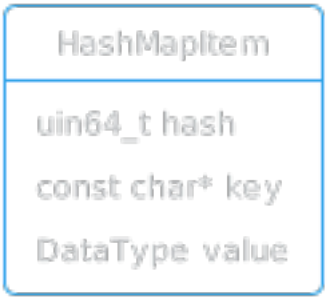

Figure 16: A hash map item

---

8 SoA vs AoS

As already mentioned, cache lines are 64 bytes and accessing just one byte always requires loading the entire cache line. If those neighbouring bytes are not accessed then only a small fraction of the memory bandwidth will be utilized. The way data is layed out is vitally important for ensuring that the executing code will not be throttled by cache misses and waiting for data to load. The way to lay out memory is to arrange it according to how it will be accessed. Place together data that will be access frequently together. In out hash table examples, open addressing hashing with linear probing and intrusive hash table items can suffer really bad performance if the size of the items is larger that the size of a cache line, however when the items are small, linear probing has few cache misses and performs better than the alternatives.

Often we think in terms of individual items as a structure, and this structure is a single piece of memory. This is most convinent when you have a single item or when you want

to pass this item around, such as between functions. However this is just one possible way to lay out this item in memory. When you have collections of such items, there is another possibility, instead of placing each field of the structure next to each other, one whole item followed by the next, to instead place one item's field next to the next item's field, and to then do this for each field. To access all the fields of a given item, one needs to lookup with the same index in to each of the arrays of each of the fields.

We call this a structure of arrays. This is in contrast to an array of structures which is the more regular approach. Figures 16 and 17 illustrate the layout of each.

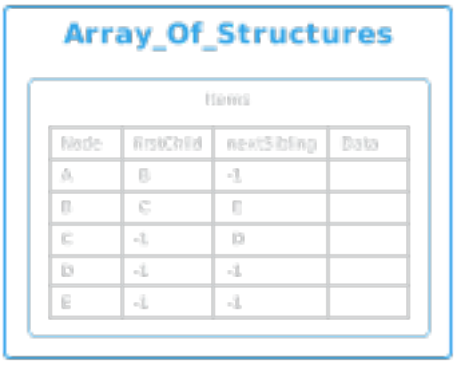

Figure 17: Array Of Structures

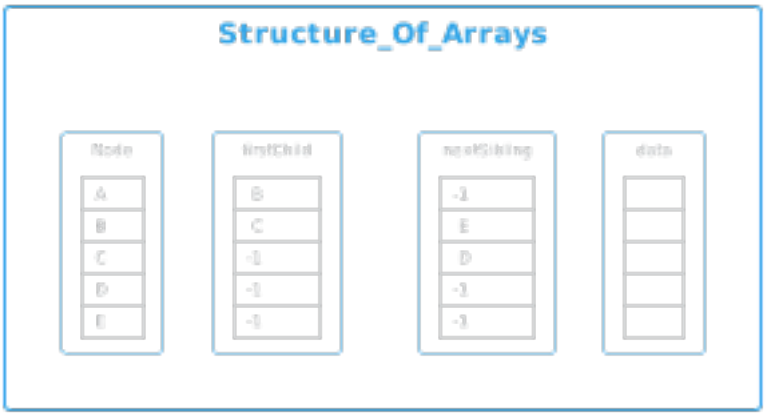

Figure 18: Structure Of Arrays

There is more complexity in dealing with a structure of arrays, however in terms of overall space occupied by the data, there should be no difference, or there is the possibility that if the original structure was poorly arranged there may have been wasted space with padding that will not exist in the structure of arrays layout, in fact making it more

efficient. We'll discuss in more depth structure memory ordering and padding a bit later.

It is not neccessary to split up and put each field in to it's own array. If some fields are frequently access together then it makes sense to keep them together. For example a 3d point made up of x, y and z components or colors made up of red, green and blue. If these are inside of a larger structure that contains several positions, a velocity, a color and so on, then it would be reasonable to keep some individual components together if they are accessed together, and move others to their own arrays.

9 Member ordering

Packing and padding. Packing is making a structure smaller. Padding is making it larger. There are reasons padding is added. The compiler does this automatically. It is required to do this because of the way CPUs operate. Most CPUs when accessing a 32-bit wide word of memory need to be accessing it with 32-bit alignment. That means the address to that 32-bit word needs to be a multiple of 4 bytes, the first it can access is at address 0, the next at address 4, then 8 and so on. For accessing a 64-bit word, the address needs to be a multiple of 8. For 16-bit wide words, the addresses need to be a multiple of 2. There are some CPUs where not doing this results in a hardware fault being raised, and your program will crash. Some CPUs can allow a program to attempt to access memory unaligned, but behind the scenes the CPU automatically does two accesses to the neighbouring pair of aligned addresses and swizzles the bytes to give the appearance it did the unaligned access. On some CPUs you incur a penalty for this, and most certainly such memory operations can never be atomic, in the worst case, the memory access straddles cache lines.

Some CPUs do the hardware fault thing, but the OS handles the fault and makes it transparently seem like the unaligned access is possible. Obviously the penalty for this quite bad, however the program won't crash.

There are some CPUs where it allows unaligned access and there is no penalty. But even when that is the case, the C language has defined rules about how structures are to be layed out. It requires members of structures to be aligned according to the size of their type, so a 32-bit member in a structure needs to be aligned to a 32-bit offset in the structure. If a structure contains a uint8_t followed by a uint32_t, the compiler is required to add 3 padding bytes between them to comply with this rule. This is padding.

Consider a structure like this:

```
typedef struct s_structA
{
    uint8_t     a;
    uint32_t    b;
    uint8_t     c;
    uint32_t    d;
} structA;
```

The compiler will place 3 padding bytes between a and b, and another 3 between c and d. That is 6 bytes of space not being used for anything. The total sizeof this structure will be 16 bytes. When we are filling cache lines with this structure, 32.5% of the space is unused padding, reducing the memory bandwidth we can use by 1/3. If however we rearrange the members by size, either from smallest to largest or largest to smallest (I don't think it actually matters), then the size of the structure becomes 12 bytes instead, only 2 bytes of padding between c and b in the below:

```
typedef struct s_structA
{
    uint8_t     a;
    uint8_t     c;
    uint32_t    b;
    uint32_t    d;
} structA;
```

Where padding is added, we could carefully add other members and try to make use of that space. When we have collections of these and using SoA (structure of arrays), then there is no wasted space at all.

Another neat thing about alignment is that pointers to a given sized type of item will be expected to have a certain alignment, so for example a pointer to a 32-bit value we can expect the pointer will be a multiple of 4. That means we expect the 2 lsb (lowest significant bits) of that pointer to be zeros. Consider this example:

```
typedef uint8_t bool8;

struct Blah
{
  bool8      isX;
  bool8      isY;
  uint32_t*  pointerToA32bitThing;
};
```

On a 64-bit machine the pointer will be 8 bytes in size and need to align to 8 bytes. In this example we are making the booleans 8-bits to illustrate a point. The two bools will take 2 bytes, then the compiler will add 6 padding bytes to align the pointer, and then there will be the pointer. The total size of this structure will be 16 btyes.

Now we know the last two bits of the pointer will always be 0. This is not something you would do regularly or for code clarity, but if this was performance critical, you could put the two bools in those lsb bits of the pointer, hence halving the size of memory used by the structure. Of course when ever the code needs to dereference the pointer, it must first mask out the bools, adding complexity to the code, but if being able to fit twice as many of these structures in to each cache line means twice the through put for a given amount of memory bandwidth, it could truely be worth it in some rare circumstances.

Now counter to what we have just learnt about how to avoid padding, there are times when padding is actually desirable. When we start looking at atomics, it will become apparent that sometimes we will want them to be on different cache lines so as to avoid something known as false sharing.