BLACKWOOD DESIGNS

Beautiful Structures

AUTHOR RANDALL MAAS

OVERVIEW Structuring data and processing in a project

BENEFITS Smaller memory footprint, faster execution, often easier to understand than other methods

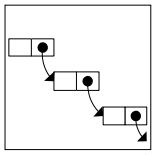
HOW IT WORKS Matrix methods:
- Bitmaps
- Directed Acyclic Graphs
- LALR(1) parser state machines

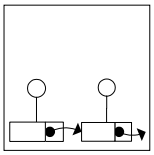
USES
- Binary relations
- Decision Trees
- Keyword searches
- Resource management
- Semantic feature sets
- Personalization
- Solving practical math problems, including those with units & unit conversion

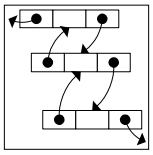
STRUCTURES
- Arrays
- Bitmaps – packed, hierarchical
- Bloom filters
- Buffers – including character-lists ('clists')
- Graphs – floor maps, scene graphs
- Lists – linked, XOR doubly-linked lists
- Name-value pairs (hash tables)
- Sets
- String matching
- Trees – linked-lists, depth first, and decision trees

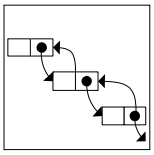
FILE: G:\My Documents\BD4 SW Design Documents\Beautiful Structures;3.doc

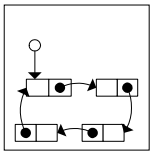
lists


p3

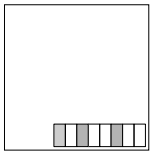

p3

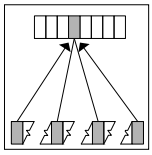

p4

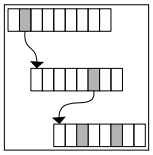

p4


p4

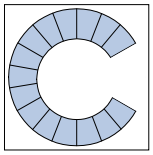
bitmap

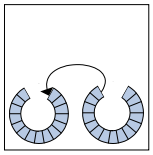

p5

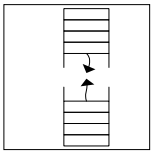

p5

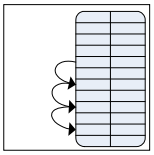

p5

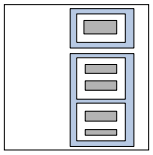
buffers and tables


p6,8

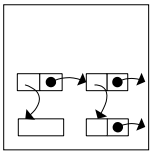

p7

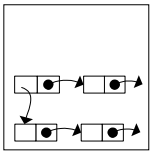

p6

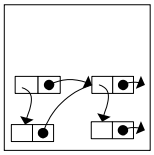

p8

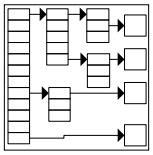

p10

trees


p10


p10

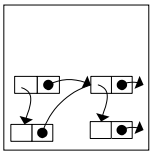

p12

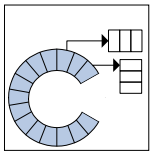

p11

rtree

timers

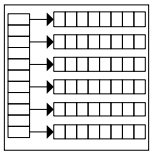
Simple

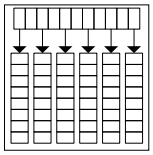

p


p8

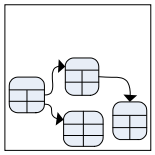
matrix

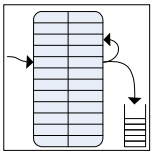
Flat

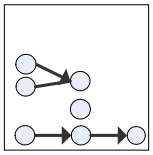

p16


p16

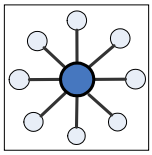
finite automatons

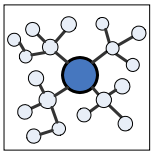

p14

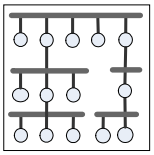

p14

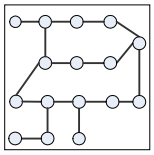

p9

networking

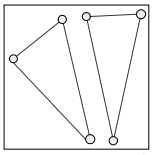


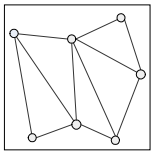


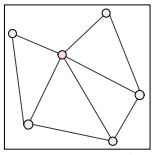




3d


p18


p18


p18

Beautiful Structures

This note describes wonderful structures. A structure characterizes how an entity – or a set of entities – is represented and manipulated. Structures have a key role, guiding the decomposition of a task or problem. The definitions of the structures are concrete – specifying their key nodes and defining fields. Of course, extra fields may be added to hold contents without changing the structure. Often a field is limited in what it may reference, or the values it may take.

e.g. whether a field may reference the current node, whether cycles are allowed, whether a node can have multiple parents.

There are analytical ways of classifying how a structure uses space, time, and work. For instance, complexity theory tells us if a structure becomes ugly with use. Yet these only tell us when a structure is not beautiful.

Beauty lies in the structure's compactness, cleverness, its choice of supported features, or generality. Typically such a structure has few fields, is easy to manipulate, and does interesting things, while remaining analytically simple. Elegance may be judged by the description's complexity, and how well we diagram examples of the structure.

1 Linked Lists

Linked lists are defined by their use of the *next* field. Each record links to the next record in the list. The simplest lists terminate, with a nil reference at the end.

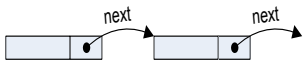


Figure 1: *Linked list*

HIERARCHIES are sets partitioned into $A \subseteq B$, $B \subseteq C$ and so forth. These can be implemented using linked lists with something similar to:

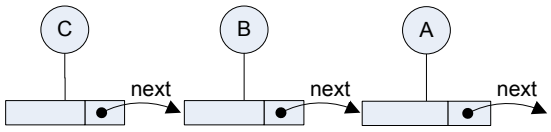


Figure 2: *Hierarchical containment using linked lists*

STACKS (e.g. resource pools) can be implemented as linked lists, with easy support for multi-processing environments. This requires a fixed *head* node pointing to the first item, with an associated *count*. Items are pushed onto the stack by:

1. Creating a new node that holds the contents and whose next is equal to the contents of `Head`.
2. Atomically updating (e.g. `cmpxchg`) the `Head` pointer to point to the new node. If that failes, repeat from step 1.

Removing an item from the list is not symmetrical – it requires an extra step:

1. Create a copy of the Head structure (the First pointer and Count), called Tmp1
2. Create a new Head structure, called Tmp2, where First = Tmp1.First->Next and Count = Tmp1.Count+1
3. Atomically update (e.g. cmpxchg against Tmp1 and Tmp2) the whole Head structure to new Tmp2 structure. If it failed, repeat from step 1.

The Count field is needed to prevent a problem such as another thread removing two items, then pushing the first item back again. That is, Tmp1's First field is correct, but its Next field (and Tmp2's First) is not.

1.1 Circular lists

Circular lists make it easy to append and prepend, as well as dequeue from the head or tail. The head points to the *last* item of the list. The first item is the "next" after the last – circular lists aren't nil terminated. Prepending is a matter of inserting an item after the "last" item. Appending to the list is the same as prepending, except that the head node must also be updated.

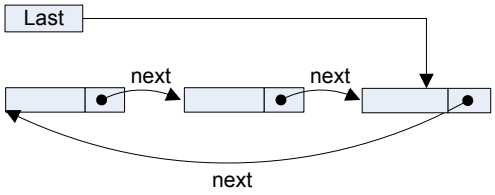


Figure 3: *Circular lists*

1.2 Doubly Linked List

Doubly linked lists can be traversed in reverse order, and a node can be inserted before or after any arbitrary node (e.g. to support ordered lists or containment hierarchies) without scanning the list.

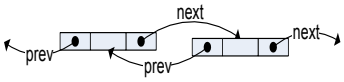


Figure 4: *Doubly linked list*

This allows us to make an LRU cache without a per-element timestamp (or a clock). The list reflects the access order – when an item is accessed, it is placed at either the front or tail of the list. When a cache slot is needed, it takes an item from the other side of the list to reuse. A hash-table is often used for fast look up of specific items.

1.3 XOR Linked List

An unusual form of a doubly-linked list is the XOR linked list. This combines references to the next and previous nodes. The downside is that operations require tracking two nodes, and performing bitwise operations on pointers or some other reference. To iterate the list:

Courtesy Honeywell.

```
for (I=List; I ; I ^= I->ForwBack)
...
```

To append a Node, one tracks a reference to the tail of the list:

```
Tail->ForwBack ^= Node;
Tail = Node;
```

To insert a Node between A and B:

```
A->ForwBack     ^= B^Node;
Node->ForwBack  = A^B;
B->ForwBack     ^= A^Node;
```

To remove a Node, you will need to know the node before the one to be removed:

```
Next      = Node->ForwBack ^Prev;
Prev ->ForwBack ^= Node^Next;
Next->ForwBack  ^= Node^Prev;
```

2 Bitmaps

Bitmaps are sets of bits accessed by index. The bits represent items (which tasks to run, resources in use, etc.), or expressions (e.g. binary relations). In special cases, when items are identified by cardinal numbers, bitmaps can sort a set faster than other methods.

Counting the number of items represented in the bitmap can be done with a *population count* operation on each word of the bitmap. It is even possible to see if any bit greater-than bit *X* is set, or any bit less-than bit *Y* is set.

Henry S. Warren, Hackers Delight*, 2002, Addison Wesley*

2.1 Bloom Filters

Bloom filters are fast screening tools – quickly reporting if an item can't possibly be in the set. If the probability of an item being in the set (e.g. a set bit) is more than 50%, you should invert the bits before or'ing into the bloom filter. A Bloom filter uses one or more filter functions – a number called *m*. To insert a record into a Bloom set:

```
for (I = 0; I < m; I++)
{
  BIdx = hash[I](key)%k;
  Bitmap[BIdx/32] |= 1u<<(BIdx%32);
}
```

To see if any item is a member of the Bloom set:

```
for (I = 0; I < m; I++)
{
  BIdx = hash[I](key)%k;
  if (!(Bitmap[BIdx/32] & (1u<<(BIdx%32)))
    return 0;
}
return 1;
```

2.2 Hierarchical Bitmaps

By squeezing out the zero bytes (or words) bitmaps can be compressed enough to represent large sets. This is useful for mapping keywords to a large number of files.

To do so, the conventional bitmap is placed at the lowest level. Each bit (except in the lowest level) refers to word in a lower level; it is set if the corresponding lower level word is nonzero. In this example there are only 8 bytes in the bitmap, so the next layer up is a single byte bitmap. With two or more bytes in a level, the process repeats with a higher level. This tree looks like:

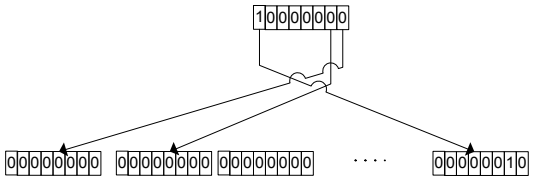
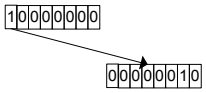


***Figure 5:** Two-level hierarchical bitmap*

Next the tree is traversed, depth first. Each word is written to the output string. Then each child is visited *only* if its bit in the current mask is set. Our example becomes, compressed:

```
0x80 0x02
```

***Figure 6:** Tree traversal*



The interpretation of the bitmap is similar to the above. The drawback is that the code is complex, compared to accessing a flat bitmap.

Note: Directly using the population count on the bytes will produce a count that is greater than or equal to the actual count of members. This is useful as a fast approximation.

2.3 Semantic Features as Bitmaps

A bitmap can hold the semantics of a concept or object. Each bit corresponds to a property (or property-value pair) akin to a 20-questions game – *male, female, bigger-than-a-breadbox*:

man	– human, adult, male
woman	– human, adult, female
boy	– human, male
girl	– human, female

When a property may take on different values – such as color or mass – each property and possible-value pair is assigned a bit. The same is true for comparative values such as *heavier-than-a-toaster,* or *bigger-than-a-breadbox.* This method works surprisingly well despite its simplicity.

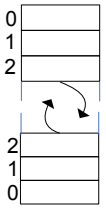
3 Arrays, Buffers and Tables

Arrays, tables and buffers are defined by their use of an *index* to access a node and its fields. The index identifies the *slot* where a node is stored; trees and networks reference a node directly.

3.1 Stacks

Often two areas (e.g. stack and local variables, or multiple stacks) are used. They are spaced so as to not overlap, but a grow-to-the-middle approach reduces the impact of those cases where one *does* overflow. Otherwise a stack overflow would corrupt the used parts other stacks after it. Since both stacks are seldom at their max at the same time, this approach reduces the likelihood of damage. On machines without enough memory to support guard spacing, this serves as a *best effort* technique.

***Figure 7:** Grow to the middle*

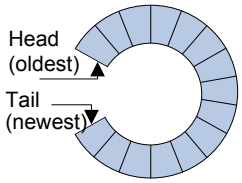


3.2 Circular Buffers

A circular buffer is a dynamic stack (built on an array). Usually holding data to process, this buffer can be used as a task run-list. They are used in multiprocessing system since it is easy to add or read blocks with little or no locking. Three thresholds may be used:

- When the number of entries exceeds a threshold, the source is disabled, or lowered in priority.

***Figure 8:** Circular Buffer*



- When the number of entries falls below a threshold, the source is enabled, or raised in priority
- When the number of entries exceeds a threshold (e.g. zero) the sink process is enabled.

Number of entries: `(number of slots+tail – head) % number of slots`

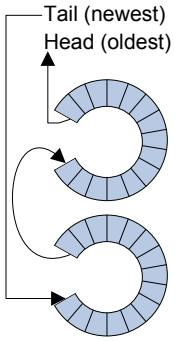
Two tails can be used for multi-process enqueuing

3.3 Chains of circular buffers

BSD has a similar technique called c-lists

A chain of buffers has the advantage of flexible use. Space constrained systems can buffer memory by dividing it into a *n* circular buffers (of various sizes), but not with a fixed allocation to any single task. This allows tasks to allocate, use and discard buffers on demand, although there is not enough memory to dedicate to each task. (It is still possible to run out of buffer space in *overload* conditions).

***Figure 9:** Circular buffer chain*



It can also be employed as a string or IO buffer with fast prepend, append, and character access operations. (Traditional strings need to be copied for modification)

3.4 Control Bands

Control bands define 'normal' variable ranges, and actions triggered when the variables leave that range:

***Table 1:** Control band upper-range example*

Var#	*When >*	*Action*
2	8	disconnect power
2	9	reduce PWM speed
4	60	Beep
4	64	Blink light

A similar table identifies actions to take when the variable is below a value. The lowest (or highest) crossed threshold can be found using a binary search – if sorted on the variable and threshold value.

3.5 Timers

Timers can be implemented as a variation on control bands – a variable identifies a clock, and the trigger value serves as the time stamp. If an identifier is tracked, a timer can be cancelled. Expired timers are removed from the table and periodic timers need to reschedule themselves.

Timers commonly segregated into separate tables, based on clock source, and count down to zero. On zero, the action the action is triggered.

***Table 2:** Count-down timers example*

Timer#	*Ticks Left*	*Action*
2	8	disconnect power
4	64	Start beep

Courtesy Jim

Like the earlier timer table, this table can be updated less frequently. There is an elegant way to track the tick count since the last update, without locks, working even when the tick counter rolls over. Using variables, of *word*-size that allow accessing the counter atomically:

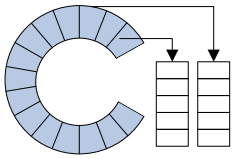
```
Tmp = grab tick counter atomically
```

```
Number of ticks since last update = Tmp – Mark
Mark = Tmp
Update the count-down table
```

Courtesy FreeBSD

A circular queue speeds timer access. Each slot links to a list of actions. For each timer tick the head pointer is moved, and the list of actions is performed. An empty queue – when the head and tail meet –may be repopulated from a *spill* table, in the form discussed earlier. The timers are offset appropriately; lists are made and inserted into the slots. Appending a new timer appends to an existing slot, to the end of the queue (if room), or to the spill table.

Figure 10: *Timer Buffer*



3.6 Hash Tables

Hash tables are a fast associative storage. The table size should be a prime number. Given a key, two hashes values and two working variables are computed from the key:

> The *index* is initially set to the first hash value (mod the table size).
>
> The *probe* value is initialized by second hash value (mod the table size). It must be at least 3.

Table 3: *Hash table example*

Key	*Hash*	*Value*
foo		1
bar		2
eek		3

To fetch an item, examine the entry at the index. A matching key settles the matter. An empty slot means the item isn't a member. Otherwise, add the probe value to the index, wrapping around the table as needed, and repeat the process. With large keys, this can be sped by storing the key's hash value in the table and comparing it first.

Storing an item is the same process – except that the entry is added (if it doesn't exist already) in the first empty slot encountered. When 90-95% full, the table will have to be expanded and rebuilt.

The hash table can store multiple items with the same primary key. This is useful for sorting items by a secondary key (e.g. priority). While inserting, if an item is found with the same key, the secondary keys are compared. If the current entry is less than or equal to the item being inserted, everything continues as before. Otherwise, the values are swapped – the current entry is preserved, and the item is inserted into slot. Then the process continues, attempting to insert the saved item.

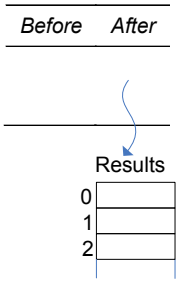
A hash table can also sort random input. To do this the probe value is fixed at 1, and the index value does not wrap around. If the value exceeds the table size, the table must be rebuilt with a larger size. When a miss occurs while inserting into the table, the keys are compared. If the current entry is less than the item be inserted, everything continues as before. If the entry is *greater*, the values are swapped – the current entry is preserved, the slot erased, and the new item inserted. Then the process continues, but attempting to insert the saved item.

3.7 Topological sorts

Sometimes we would like to specify rules of ordering items – *A* must happen sometime before *B*, or *B* must occur after *A*, etc. – and produce an ordered list from these rules. *Topological sorting* is one way to do this. The rules are represented as a dependency table, which is then scanned.

1. All of those items in the *run list* that have finished are placed onto the *completed list*.
2. All of the *waiting* items whose dependencies have been completed (or do not dependent on anything) are placed (in any order relative to each other) onto the *run-ready* list.
3. If the run-list is empty, but the waiting-list isn't, there is a cycle. Topological sorting is well-defined only with a directed graph – if the graph has a cycle an arbitrary *cut* must be made to provide an approximate ordering. To do this, a waiting item is randomly selected and placed onto the run list.

Figure 11: *Topological sorting*



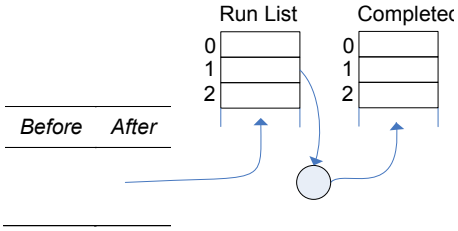


Figure 12: *Topological sorting*

Efficiency improvements are possible, such as using a hash-table discipline or sorting the table. Another is to track the number of items a task depends on, decremented it as the dependencies complete. Zero count items are then moved (in any order) to run-ready table. Linkers add a *lazy* population step – items are considered only when *needed* by some other required or needed item.

3.8 Annotating Objects

Items – *objects* – can be annotated with name-value pairs. The following address book entries illustrate the concept. Links to other entries are made *symbolically*, requiring matching the values of items with the **name** attribute.

Table 4: *Address Book example*

objId	*Name*	*Type*	*Value*
1	**name**	string	Adam Jones
1	**email**	string	adam@jones.org
1	**age**	int	33
1	**spouse**	string	Bethany Jones
1	**child**	string	Chuck Jones
1	**father**	string	Bartleby Jones
1	**mother**	string	Diane Jones
2	**name**	string	Bethany Jones
2	**spouse**	string	Adam Jones
2	**child**	string	Chuck Jones
2	**father**	string	Ethan Kirk
2	**mother**	string	Frieda Kirk

To find all email addresses in a business card, one would:

1. Scan the array entries in this node or that have this node as an ancestor.
2. Keep those entries whose type is a string, and whose Name is 'email.'

3.9 Extent (Span) Tables, Interval Sets

Span tables have entries with a start and stop location. Span tables work well with one dimension variable and when entries can't overlap (see *Range Trees* for another technique that works with multiple dimensions.)

ANNOTATING TEXT. A file or text may be annotated using such a table. This may be used to mark up portions to indicate their font typeface or attributes such as bold or italic. The annotations can also cross-link to other portions of the text, provide comments, and so forth. Below is an example annotation array:

Start	*Len*	*Name*	*Type*	*Annotation*
0	8	fontWeight	string	bold
0	8	fontFamily	string	Helvetica
28	64	link	URL	Link to earlier section
28	64	comment	string	Comment

Table 5: *Annotation example*

DEPTH-FIRST TREES AS AN ARRAY. Trees can be flattened out (depth-first) and stored as a span able – one of the few ways of storing trees in SQL databases. *B* is a child of *A*, if *B*'s start is greater than or equal to *A*'s, and its end is less than or equal to *A*'s. If the tree is sorted, there are no more children when an entry is found with a start greater than the given parent's end. This sort by the start value (in ascending order) and the end value (in descending order).

Joe Celko popularized this technique in his articles and "SQL for Smarties" books

To implement such a tree, you may need an arbitrary numbering scheme. For instance, start with a (possibly imaginary) root node given the full integer range. Each child has a span equal to the parent's divided by a max number of children per node (possibly per node at that level). Their start and end values are computed as appropriate, and the process continues.

4 Tree Structures

Trees come in two forms: top-down where each node specifies its children, or bottom-up, with each node specifying its parent. The node's identifier, unlike with tables, is fixed and can not change – e.g. an address or index. Trees can form complex structures, any structure in this document, albeit with some complexity.

4.1 Threaded Interpretative Language

In a threaded interpretative language, procedures are a nil-terminated array of other nodes to call. Many Forth implementations use such a system. Special built-in nodes can be used to manipulate the stack. A layer may be used to allow replacing modules.

4.2 LISP's method: Trees based on Linked-lists

A tree can be made from linked lists – a node references the "first child" of a linked list of its children. The rules for trees based on linked-lists are:

LISP calls this a cons *list*
next *is called* cdr
firstChild *is called* car

- A node is the parent of exactly those nodes in the linked-list of its first child.
- Children of a level *n* node are at level *n+1*

A node has two fields next and firstChild. The next field may point to another node or nil.

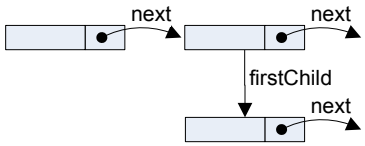


Figure 13: *Child lists*

A child may be a second kind of node. This node is used to hold the value – a number, a string, and so forth.

LISP calls value *car*

A LISP procedure is a list of nodes to call, where the firstChild is a list of the call's parameters. This is similar to a *threaded-interpretive language*. LISP determines whether each parameter element should be evaluated *prior* to the call. This is done with a complex procedure, which is inconsistent between LISPs; this is an instance of LISP's genetic capacity for making mistakes.

Thanks to John McCarthy for the genetic phrase

A variant of the child lists can use circular lists, pointing to the last child of the list; next points to the first child.

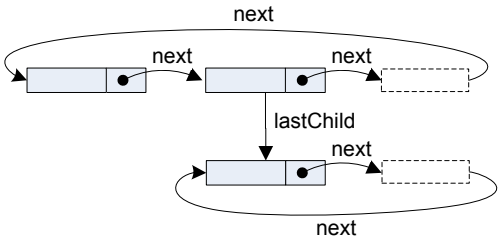


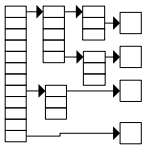
Figure 14: *Child rings*

4.3 Three-Level Indirection Tree

Three level indirection trees are most often used to implement very large arrays (e.g. vectors). This type of structure is fast at updating.

This structure is often used in file systems to distribute blocks of the file across the disk. The structure has less depth at the tail – to allow growing the file quickly.

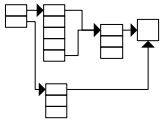
Figure 15: *Three level tree*



4.4 Directed Acyclic Graph (DAG)

Directed acyclic graphs are a compressed form of trees. Many nodes can point to the same node, but they can't loop back. If it is the same leaf, they don't need duplicates. Common with other types of trees.

Figure 16: *Directed acyclic graph*



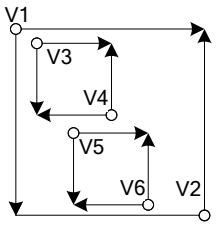
4.5 Range Tree (rtree)

Range trees are generalized span trees. They serve the same role as a direction graph (which we will discuss later), including:

- Finding all items within a geometric region,
- Attributing (marking up) maps and other geometric items – e.g. adding place names locations images and textures.
- Creating regions – e.g. states, counties, etc – on a map.

Each subtree in a range tree specifies the perimeter around an n-dimensional rectangle.

Figure 17: *Example of Range tree*



The tree can be structured in any manner described earlier. The key difference is that each node has an origin and size vectors. Every child node's origin and size is contained within its parent. This allows finding all the items within a region.

5 Networks and Graphs

Networks may be represented in several forms; the most common type use node with pointers, a table, or a matrix. These build on and generalize the tree structures. Networks are the base structures for more types, such as the finite automatons in the next section.

5.1 Direction graph

Direction graphs are made of nodes with two items for each dimension – the first of the next cell in the "previous" direction along that dimension, and the other for the next cell in the direction along the given dimension. Where the Range tree specifies the perimeter around a n-dimensional rectangle, a direction graph specifies how cells link together. This makes it easier to walk-around (traverse a graph).

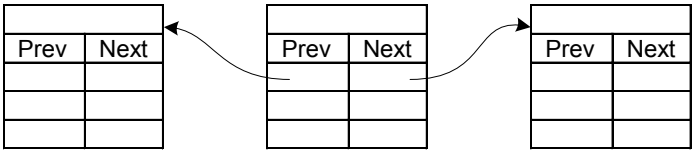


Figure 18: *Direction graph*

Infocom's Z-machine used 4-links to neighbors to the east, west, north and south. The diagram below describes how a floor map can be dissected into 5 cells. To construct the direction graph:

1. Draw a floor map.
2. Draw straight lines all the way thru for each of the walls.

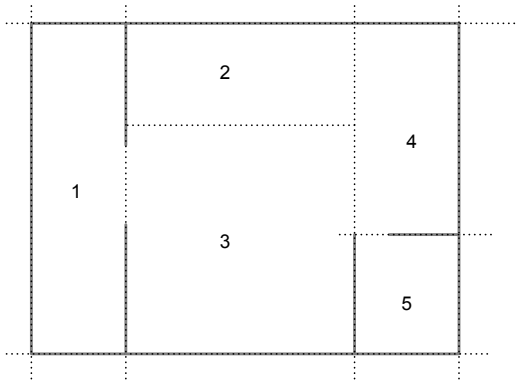


Figure 19: *Example floor map*

At this point there is a choice: either number the boxes, or number the lines. Lets do it with the boxes; you can skip boxes that you can't get into. For each box:

3. Add a row to the table below
4. Fill in spot for the box number
5. If there is no wall to the right, add the number for the box to the right. The direction with only a "wall" has a nil for its neighbor in that direction.
6. Repeat for the left, up and down.

This can be made more interesting with a door. The system can use a table to answer if one can traverse a link:

From	*To*	*Check*
1	3	Door1IsOpen()
3	1	Door1IsOpen()

Table 6: *Edge check*

The `Can()` checks to see if there is an entry in the edge check table; if there is, it queries the function to see if traversal is allowed. In this example going between room 1 and 3 checks to see if Door1 is open. The table is directed – if the check for going from 3 to 1 wasn't present, Door1 would behave more like a trap door, allowing us to always to go into room 1, but only sometimes to go into room 3. Doors are finite automatons, which will be discussed later.

5.2 Type Conversion

Many languages and context have implicit conversion methods. A table of conversion allows us to automatically convert between types:

From Type	*To Type*	*Cost*	*Conversion Action*
float	double	0	
double	float	1	

Table 7: *Type conversion*

A lossless conversion has a cost of 0, while lossy ones have higher numbers. Converting from a float to a double is lossless; converting from a double to a float has some loss. Typically this table is converted into a matrix (representing a weighted graph) and a standard kit of algorithms is applied to determine the lowest loss (least costly) set of conversion steps to follow.

In most cases, you want an incremental algorithm, e.g. Floyd Warshall's or Djisktra's. These construct a sequence of conversion actions to make. In rare cases you may want to pre-compute every possible conversion, use the all points shortest graph algorithm. The key is that these find the *shortest path* to follow, and avoid cycles: going from float to double back to float is silly.

5.3 Unit conversion

Converting between numerical units follows a similar process. Typically the conversion from one unit to another employs a formula of the form: $v \cdot c_1 + c_0$

From Unit	*To Unit*	C_1	C_0
Dinar	dollar	0	
Celsius	Fahrenheit	1	

Table 8: *Unit conversion*

This is again converted to a graph, and the algorithms are applied. You can use the same incremental techniques mentioned in the previous section. Or you can pre-compute a matrix that converts from each unit to every other unit; this is called a closure. The advantage, unlike with type conversion, unit conversions are more widely distributed across units than types.

First, start with a directed graph. The computed the closure is used in a sparse form like so:

$$converted\ value = \mathbf{Closure} \cdot \begin{vmatrix} 0 \\ value \\ 0 \\ \vdots \\ 1 \end{vmatrix}$$

These benefits greatly from fast matrix and vector representations (see section 7).

6 Finite Automatons

Finite automatons create dynamic and responsive behaviours by adding state sensitivity to tables and networks. This allows construction of decision trees, routing tables, parsers and other sophisticated behaviour. States themselves may have enter and exit actions, and transition actions.

State	*Enter Action*	*Exit Action*

Table 9: *State transition actions*

From State	*To State*	*Action*

Table 10: *State transition actions*

6.1 Decision Trees

A decision tree can be used to perform bitwise pattern matching, IP Address routing, or classify an item using feature sets. These trees are models examining the interaction between or cause-and-effect of different factors. In some contexts, they are known as Patricia tree and radix trees. The tree itself is a *directed acyclic graph*. Each node is a rule, in one of three forms:

- If the item matches, go to Rule X; otherwise go to rule Y
- If the item matches, return X; otherwise go to rule Y.
- Perform no match, the Items is X or perform action Z (covered in tables above)

Algorithms such as ID3 or C4.5 can be used to construct the tree.

6.2 Deterministic Finite Automatons (DFA's)

Deterministic finite automatons (DFA's) are structurally simple and easy to understand. Each state is fetched from a look-up table and is used to map input events or symbols to the next state. This simplicity makes them one of the fastest matching techniques.

DFA's may be created from NFA's (see the next section). Alas many such DFA's are inelegant. The translation from NFA to DFA can create some pretty hairy monsters with a huge numbers of states and nodes.

6.3 NFA Parser using State and Hash

NFAs perform faster than DFA's for some types of matching specifications; DFA's can be converted into NFA's for these cases. The NFA interpreter must keep a list of possible states it may currently be in. The node transition reduces or expands this list.

This can be implemented as a single table using state numbers or use pointers to further tables. A parser may include a precedence number, used with grouping arithmetic operations. The primary key (in a hash table) is the <*state number, symbol*> pair; the precedence is the secondary key, and the table is kept sorted as described in the hash table section.

State	*Symbol*	*Hash**	*Precedence*	*Next State*	*Action*

Table 11: *Hash table example*

The main drawback of this form, rather than a tree-form, is the expense to modify the grammar.

6.4 Hashed Trie

A trie is both a tree and a DFA, thru the use of special entries and an extra column. A hashed trie is used for performance – it's key look up time is O(*key length*). This is structure could be used for UNIX-style mount-tables, and performing morphological analysis such as spell checking. The Hashed Tries is special because it allows fast matching, the ability to enumerate all possible strings it will match (in sorted order, no less) and return the parent of a given matched node.

'trie' is short for retrieval structure

A hashed trie can be implemented with a state-number (or a state selector), using a single table (as below). States include entries with a nil symbol whose next state is used to link back to the parent. Combined with the optional next sibling node one can enumerate all possible strings in sorted order. The parent node can be found by scanning the next-sibling chain to find the entry with a nil symbol.

State	*Symbol*	*Hash**	*Next State*	*Next Sibling**	*Action*

Table 12: *Hash trie example*

6.5 Hierarchical State machines

Hierarchical state machines are a chain of state machines, passing events from top-most state machine down until one processes the event. Their most common use is to provide a consistent look and feel in GUI systems.

The typical formulation is a series of operations like that in the example. A state machine may pass an event – or the output it's processing of the event – as the input to another state machine. These "other" state machine may include the standard library of processors or responses, and allow the programmer to selectively override some handlers or inject events

1. Get event (message)

2. If my handler 1, call my handler
3. If ok to call system handler 1, call system handler 1
4. …
5. down to handler n

7 Matrix

A variety of stylized forms of matrices have been defined in mathematics, and concrete structures exist to better serve these. Most often these special forms are used to speed multiplication, but they may also speed other operations. Most special cases matrices being:

- Zero and identity matrices, allowing especially fast at multiplication and addition
- Dirac matrices of ones and zeros, which can be compactly implemented as bitmaps
- Triangular matrices (upper left and lower right) which are faster for multiplication
- Compact sparse matrices, which are very fast at multiplication

Some of the special forms relate to how the matrix cells are structured and the impact on key matrix operations:

- Row-indirection (faster for LU decomposition)
- Column-indirection (faster for vector concatenation)

Vectors with one or two non-zero elements can be treated as special forms, tracking the row(s) and value(s).

7.1 Compact Sparse Matrix

A compact sparse matrix has three parts:

- A value array,
- A column index array (to index each value within a row), and
- An extent array that holds the starting index in the value for each row.

HOW TO MAKE A MATRIX. Scan the matrix,

```
for (Y = 0; Y < Height; Y++)
{
  NumElems = 0;
  for (X = 0; X < Width; X++)
   if (0 != Matrix[X,Y])
     {
        Append Matrix[X,Y] onto values
        ColIndex[NumElems] = X;
        NumElems++;
     }

  if (Extent is Empty)
    append NumElems
   else
    append NumElems + previous value in Extent
}
```

HOW TO MULTIPLY AGAINST A VECTOR:

```
Sum = 0;
Ofs = Extent[Row]
RowWidth = Extent[Row+1] - Ofs
for (X = Ofs; X < Ofs+RowWidth; X++)
 Sum += Value[X] * Vector[ColIndex[X]]
Result[Y] = Sum;
```

8 3D shapes

3-D (and any other) shapes employ a number of interesting structures. Many 3D objects describe shapes by lists of triangles, and describe scenes of 3D objects as trees of the objects and their transforms.

8.1 Vertex Lists

DirectX 3D and OpenGL work by passing lists of points describing triangles. These lists allow a specific type of compression reducing data movement – almost every shape reuses many of the points. This is akin to the dictionary in data compression: a table of vertices, and a brief list of which vertices to use. This approach also allows computing patterns, so that software doesn't have to fully enumerations of all the points, filling huge regions of memory.

Each point can be given a set of attributes, such as

- The color at the point; this gives triangle gradient shading.
- The surface normal for the point, which is used for lighting effects on triangles.
- A mapping to a point in a texture image.

Destination Point	*Source Image Point*

Table 13: *Texture mapping*

This section is laid out from the most generic, to handling special cases:

- Vertex lists
- Strips, where every triangle shares two vertices with its neighbor
- Fans, where every triangle shares the same vertex

Table 14: *Vertex list*

Vertex
1
2
3

Which of these to use? The latter ones reduce the number of vertices needed to be moved to draw the image.

Type	*#Vertices*
List	3 * number of triangles
Strip	number of triangles + 2
Fans	number of triangles + 2

Table 15: *Costs of different formats*

VERTEX LIST. Every three points forms a triangle to draw on the screen.

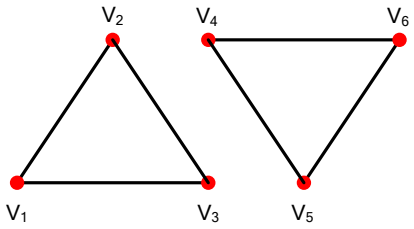


Figure 20: *Example of triangle list*

VERTEX STRIP. Two points from the previous triangle is reused

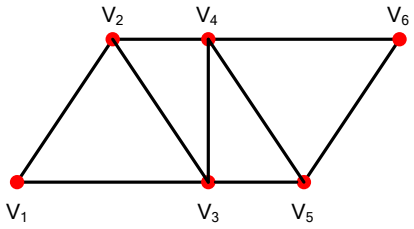


Figure 21: *Example of triangle strip*

VERTEX FANS. The first point is reused, and the last points from the previous triangle is reused

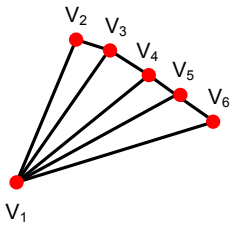


Figure 22: *Example of triangle fan*

8.2 Textures

Textures combined with vertices allow for fast, compact animation. For example, one can make Pixar's Luxo Jr animation that fits on old floppy – or uses bitrate compatible with old modems. Initially the animation would have to send:

- Textures for background and lamp surfaces
- The initial vertices
- The links between the vertices, and with the texture.

The animation from that point is just a matter of sending the new vertex locations.

8.3 Scene Graphs

A scene graph is very useful for representing 2D and 3D shapes and scenes. It is a tree of shapes and affine transforms; each node is one of three kinds:

1. An affine transform, with a child list of other nodes
2. An image
3. A vertex list (see previous section).

9 Unification

Unification is useful for solving expressions or rearranging them into a solvable form. Variables have three parts: a value, an indication whether it is bound, and a link to another variable.

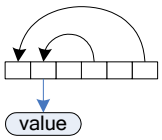


Figure 23: *Unification structure*

SIMPLE UNIFICATION. The first steps of unification are inferring values and sameness of variables. When there is an A=B (or implicitly with a functional call): If A and B are both variables, then they are linked together. If B (or A) is a value, then the chain is checked to see that the variable either is (1) bound to the same value or (2) isn't bound, causing the variable to be bound to the other variable or value.

UNIFYING LISTS. This unification process can be expanded to complex structures. Lists are a good example. A free variable is unified with list by binding the variable to the list. Two lists unify by going over each positions in the lists and unifying:

```
For X = next item in List A, Y = next item in list B,
        unify X and Y.
        stop if failure.
```

Trees, matrices, and tables can be unified in a similar manner.

10 Appendix: Structure equivalence

This is an appendix to describe how to determine whether two structures are related or can do similar things. This analysis ignores performance variations (e.g. computational, memory or IO complexity).

Abstract structures are structures, except they are defined in terms of functions or relations on the structure – set membership, disjunction, etc. In this document, references have been pointers but they could be array indices, even relative offsets for indices and pointers. Abstraction goes a step further – given a structure or a node, you have a defined way to know what the *next* node is (in abstract lists). It may be complex process to determine this *next* node.

In relational databases, looking for structure equivalence is called *Schema matching*.

1. Compare structures A and B for equality, using methods below
2. Find the most significant structure that A and B are both partially equally to. Such a structure may be in this document, or a new one that was identified by this process.

There are three means of equivalence presented below. They are presented in order of simplicity (and performance of evaluation).

- Comparing the fields
- Defining common functions
- Defining common relations

The latter two are important since they define an intermediary form of using the structure – many different structures can be manipulated by the same algorithms (or a small number of variations). Simply using fields expands how often a given algorithm must be implemented.

10.1 Strict field equivalence

Two structures can be compared on a field-by-field basis. This method uses or creates a mapping of the names of one structure onto those in the other. Such a mapping would be unnecessary if all structures employed the same name for the same purpose; in practice the

names vary as a matter of style and practical concern. (This is an identity only in the most trivial of structure equivalences). Most often a structure is a subset of another – not all fields in one structure are in another. Determining this: given by a human, or inferring by substantial similarity. The table below provides an example mapping the fields of a list and a tree.

List	*Tree*
next	lastChild nextSibling

Table 16: *field equivalence example*

Each structure field is mapped to unique name. Then a disjunction of the two sets of unique names is performed.

- If all of A's fields are employed by B, but B has fields that aren't shared with A, then A is ≤ B. In this example, a list is a subset of tree. Otherwise,
- If A and B have the same fields (i.e. there is an equivalent or comparable field), the structures are functionally equal

If A and B share none of the same field, a more complex analysis is need to determine possible relationship.

10.2 Equivalent by functions

The next technique requires each structure to provide a definition for well known functions, and creating a mapping between functions in with each structure. In many cases, the functions simply access a field. But consider the two kinds of trees mentioned in section 4.2. Both trees define a `next(n)` function for each node. The tree that is built on the *firstChild* field has a simple `firstChild(n)` function. The tree that is built on the *lastChild* field can define a `firstChild(n)` function as `next(firstChild(n))`.

- A is ≤ B if all of the functions defined for A are defined for B, but B has other well-defined functions that A does not
- If A and B both have the same functions defined, they are functionally equal
- A is ≈ B, if all of the functions defined for well-known type C are defined by A and B. Recommend finding the most powerful well-known type that this is true for.

10.3 Equivalent by relation

The last technique employs the relations. Relations go beyond functions. They are used, for example, in SQL statements. For example, two nodes are siblings if they have the same parent, but are not the same person:

```
Sibling(n1,n2): Parent(n1,p), Parent(n1,p), n1 != n2.
```

The procedure to implement the `Parent()` relationship may be a list scan, or a pointer access. If it were written as a function, you'd typically write `p=Parent(n1)`. With relations, there is (implicitly) two other functions `n1 <= Children(p)`, and `IsChildOf(p, n1)`.

- A is ≈ B, if each of A's conditions has an equivalent condition on B (and vice-versa)
- A is ≤ B if all of the conditions on the relation for A are defined for B, and each of the data sources is for A are less-than or equal to those defined for B. (B may have further conditions)

11 Appendix: Suggestions

This is an appendix to offer some suggestions to make a structure generic.

Do: functions should be named after a noun or adverb. Rather than `nextToEast(n)`, prefer `next(n, direction)`.

An intermediary form – many different structures can be manipulated the same (few) algorithms.

Suggested names for functions

- next(n)
- next(n,direciotn)
- firstChild(n)
- value(n)
- value(n, field)
- nodeForName(name)
- endOfChildList(n)
- lastChild(n)
- cost(from, to)
- action(from, to)
- parent(n)

Suggestions for relation names

- parent(n,p)
- child(p,n)
- value(n,v)
- value(n,field,v)
- value(f,t,v)
- cost(f,t,c)
- action(f,t,a)
- next(n1,n2)
- next(n1,n2,direction)

Beauty seldom alights in the complex, while arcs and nodes, properly drawn, give a sense of elegance. The diagram may have simplicity, balance or symmetry, parallel development, even tension.