CHAPTER 2

Data Structures for Language Processing

The space–time tradeoff in data structures, i.e. the tradeoff between memory requirements and the search efficiency of a data structure, is a fundamental principle of systems programming. A language processor makes frequent use of the search operation over its data structures. This makes the design of data structures a crucial issue in language processing activities. In this chapter we shall discuss the data structure requirements of language processors and suggest efficient data structures to meet these requirements.

The data structures used in language processing can be classified on the basis of the following criteria:

1. Nature of a data structure—whether a *linear* or *nonlinear* data structure
2. Purpose of a data structure—whether a *search* data structure or an *allocation* data structure.
3. Lifetime of a data structure—whether used during language processing or during target program execution.

A *linear data structure* consists of a linear arrangement of elements in the memory. The physical proximity of its elements is used to facilitate efficient search. However, a linear data structure requires a contiguous area of memory for its elements. This poses problems in situations where the size of a data structure is difficult to predict. In such a situation, a designer is forced to overestimate the memory requirements of a linear data structure to ensure that it does not outgrow the allocated memory. This leads to wastage of memory. The elements of a *nonlinear data structure* are accessed using pointers. Hence the elements need not occupy contiguous areas of memory, which avoids the memory allocation problem seen in the context

of linear data structures. However, the nonlinear arrangement of elements leads to lower search efficiency.

Example 2.1 Figure 2.1(a) shows memory allocation for four linear data structures. Parts of the data structures shaded with dotted lines are not in current use. Note that these parts may remain unused throughout the execution of the program, however the memory allocated to them cannot be used for other data structure(s). Figure 2.1(b) shows allocation to four nonlinear data structures. Elements of the data structures are allocated noncontiguous areas of memory as and when needed. This is how two memory areas are allocated to E while three memory areas are allocated to F and two memory areas are allocated to H. No parts of the data structures are currently unused. Free memory existing in the system can be allocated to any new or existing data structures.

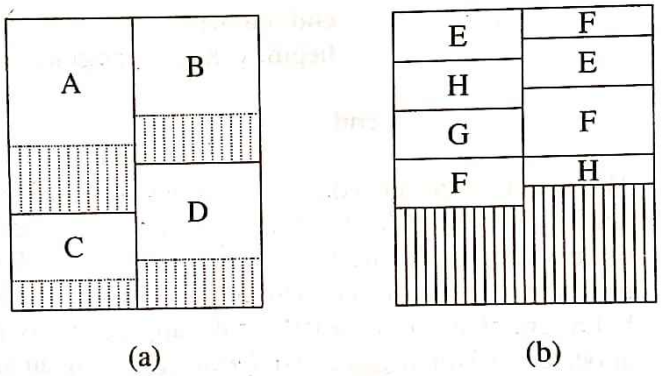


Fig. 2.1 Linear and nonlinear data structures

Search data structures are used during language processing to maintain attribute information concerning different entities in the source program. These data structures are characterized by the fact that the entry for an entity is created only once but may be searched for a large number of times. Search efficiency is therefore very important. *Allocation data structures* are characterized by the fact that the address of the memory area allocated to an entity is known to the user(s) of that entity. Thus no search operations are conducted on them. Speed of allocation or deallocation and efficiency of memory utilization are the important criteria for the allocation data structures.

A language processor uses both search and allocation data structures during its operation (see Ex. 2.2). Search data structures are used to constitute various tables of information. Allocation data structures are used to handle programs with nested structures of some kind. A target program rarely uses search data structures. However, it may use allocation data structures (see Ex. 2.3).

Example 2.2 Consider the Pascal program

```
Program Sample(input,output);
  var
      x,y : real;
      i   : integer;
  Procedure calc(var a,b : real);
     var
        sum : real;
     begin
        sum := a+b;
        ...
  end calc;
  begin { Main program }
  ...
end.
```

The definition of procedure `calc` is nested inside the main program. Symbol tables need to be created for the main program as well as for procedure `calc`. Let us call these $Symtab_{\texttt{Sample}}$ and $Symtab_{\texttt{calc}}$, respectively. These tables are search data structures for obvious reasons. During compilation, the attributes of a symbol, e.g. symbol a, are obtained by searching the appropriate symbol table. Memory needs to be allocated to $Symtab_{\texttt{Sample}}$ and $Symtab_{\texttt{calc}}$ using an allocation data structure. The addresses of these tables are noted in a suitable manner. Hence no searches are involved in locating $Symtab_{\texttt{Sample}}$ or $Symtab_{\texttt{calc}}$.

Example 2.3 Consider the following Pascal and C program segments

```
Pascal  :  var p : ↑integer;
           begin
              new(p);

   C    :  float *ptr;
           ptr = (float *) calloc(5, sizeof(float));
```

The Pascal call `new(p)` allocates sufficient memory to hold an integer value and puts the address of this memory area in `p`. The C statement `ptr = ..` allocates a memory area sufficient to hold 5 float values and puts its address in `ptr`. Accesses to these memory areas are implemented through pointers `p` and `ptr`. No search is involved in accessing the allocated memory.

2.1 SEARCH DATA STRUCTURES

A search data structure (or *search structure* for short) is a set of entries, each entry accommodating the information concerning one entity. Each entry is assumed to contain a *key* field which forms the basis for a search. Throughout this section we will use examples of entries in a symbol table used by a language processor. Here, the key field is the *symbol* field containing the name of an entity.

Entry formats

Each entry in a search structure is a set of fields. It is common for an entry in a search structure to consist of two parts, a fixed part and a variant part. Each part consists of a set of fields. Fields of the fixed part exist in each entry of the search structure. The value in a *tag* field of the fixed part determines the information to be stored in the variant part of the entry. For each value v_i in the tag field, the variant part of the entry consists of the set of fields SF_{v_i} (see Ex. 2.4). Note that the fixed and variant parts may be nested, i.e. a variant part may itself consist of fixed and variant parts, etc.

Example 2.4 Entries in the symbol table of a compiler have the following fields:

Fixed part: Fields *symbol* and *class* (*class* is the tag field).

Variant part:

Tag value	*Variant part fields*
variable	*type, length, dimension info.*
procedure name	*address of parameter list, number of parameters.*
function name	*type of returned value, length of returned value, address of parameter list, number of parameters.*
label	*statement number.*

Fixed and variable length entries

An entry may be declared as a *record* or a *structure* of the language in which the language processor is being implemented. (We shall use the word 'record' in our discussion.) In the *fixed length entry* format, each record is defined to consist of the following fields:

1. Fields in the fixed part of the entry
2. $\bigcup_{v_i} SF_{v_i}$, i.e. the set of fields in all variant parts of the entry.

All records in the search structure now have an identical format. This enables the use of homogeneous linear data structures like arrays. In turn, the use of linear organizations enables the use of efficient search procedures (we shall see this later in the section). However, this organization makes inefficient use of memory since many records may contain redundant fields.

In the *variable length entry* format, a record consists of the following fields:

1. Fields in the fixed part of the entry, including the tag field
2. $\{ f_j \mid f_j \in SF_{v_j} \text{ if } tag = v_j \}$.

This entry format leads to a compact organization in which no memory wastage occurs.

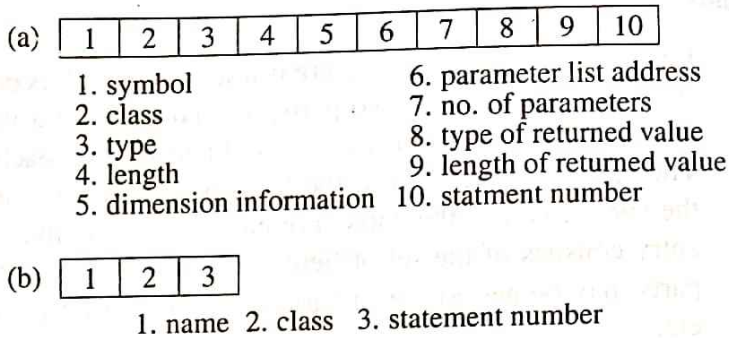


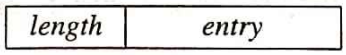
Fig. 2.2 (a) Fixed entry, (b) Variable length entry for label

Example 2.5 Figure 2.2 shows two entry formats for the symbol table of Ex. 2.4. Part (a) shows the fixed length entry format. When *class* = label, all fields excepting *name*, *class* and *statement number* are redundant. Part (b) shows the variable length entry format when *class* = label.

When a variable length entry format is used, the search method may require knowledge of the length of an entry. In such cases a record would consist of the following fields:

1. A *length* field
2. Fields in the fixed part of the entry, including the tag field
3. $\{ f_j \mid f_j \in \mathrm{SF}_{v_j} \text{ if } tag = v_j \}$.

We will depict this format as



Hybrid entry formats

A Hybrid entry format is used as a compromise between the fixed and variable entry formats to combine the access efficiency of the fixed entry format with the memory efficiency of the variable entry format. In this format each entry is split into two halves, the fixed part and the variable part. A *pointer* field is added to the fixed part. It points to the variable part of the entry. The fixed and variable parts are accommodated in two different data structures. The fixed parts of all entries are organized into an efficient search structure, e.g. a linear data structure. Since the fixed part contains a pointer to the variable part, the variable part does not need to be located through a search. Hence it is put into an allocation data structure which can be linear or nonlinear in nature. The hybrid entry format is depicted as

Operations on search structures

The following operations are performed on search data structures:

1. Operation *add:* Add the entry of a symbol.
2. Operation *search:* Search and locate the entry of a symbol.
3. Operation *delete:* Delete the entry of a symbol.

The entry for a symbol is created only once, but may be searched for a large number of times during the processing of a program. The deletion operation is not very common.

Generic search procedure

We give the generic procedure to search and locate the entry of symbol s in a search data structure as Algorithm 2.1.

Algorithm 2.1 (Generic search procedure)

1. Make a prediction concerning the entry of the search data structure which symbol s may be occupying. Let this be entry e.
2. Let s_e be the symbol occupying e^{th} entry. Compare s with s_e. Exit with success if the two match.
3. Repeat steps 1 and 2 till it can be concluded that the symbol does not exist in the search data structure.

The nature of the prediction varies with the organization of the search data structure. Each comparison of Step 2 is called a *probe*. Efficiency of a search procedure is determined by the number of probes performed by the search procedure. We use following notation to represent the number of probes in a search:

p_s : Number of probes in a successful search
p_u : Number of probes in an unsuccessful search

2.1.1 Table Organizations

A table is a linear data structure. The entries of a table occupy adjoining areas of the memory. Two points can be made concerning tables as search structures:

1. Given the location of an entry of the table, it is meaningful to talk of the *next* entry of the table or the *previous* entry of the table. A search technique may use this fact to advantage.
2. Tables using the fixed length entry organization possess the property of *positional determinacy*. This property states that the address of an entry in a table can be determined from its entry number. For example, the address of the e^{th}

entry is $a+(e-1).l$, where a is the address of the first entry and l is the length of an entry. This property facilitates the representation of a symbol s by e, its entry number in the search structure, in the intermediate code generated by a language processor (see Section 1.3.1). Positional determinacy may also be used to design efficient search procedures, as we shall see later in this section. Tables using variable length entries do not possess the property of positional determinacy, so one must step through the first $(e-1)$ entries of a table in order to locate the e^{th} entry. Hence variable length entries are generally avoided in linear data structures. In our discussion, we assume the use of fixed length entries in linear data structures.

Sequential search organization

Figure 2.3 shows a typical state of a table using the sequential search organization. We use the following symbols in our discussion:

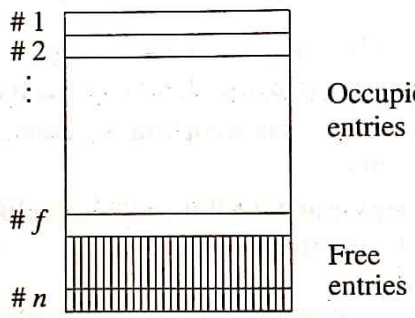


Fig. 2.3 Sequential search table

n : Number of entries in the table
f : Number of occupied entries

Search for a symbol

At any stage the search prediction in Algorithm 2.1 is that symbol s occupies the *next* entry of the table, where *next* = 1 to start with. From Algorithm 2.1, it follows that if all active entries in the table have the same probability of being accessed, we have

$p_s = f/2$ for a successful search
$p_u = f$ for an unsuccessful search

Following an unsuccessful search, a symbol may be entered in the table using an *add* operation.

Add a symbol

The symbol is added to the first free entry in the table. The value of f is updated accordingly.

Delete a symbol

Deletion of an entry can be implemented in two ways, physical deletion and logical deletion. In *physical deletion*, an entry is deleted by erasing or by overwriting. Thus, if the d^{th} entry is to be deleted, entries $d+1$ to f can be shifted 'up' by one entry each. This would require $(f-d)$ shift operations in the symbol table. An efficient alternative would be to move the f^{th} entry into the d^{th} position, thus requiring only one shift operation. Physical deletion causes changes in the entry numbers of symbols, which interferes with the representation of a symbol in the IC. Hence physical deletion is seldom used.

Logical deletion of an entry is performed by adding some information to the entry to indicate its deletion. This can be implemented by introducing a field to indicate whether an entry is active or deleted. The complete symbol table entry now looks as follows:

Active/deleted	*Symbol*	*Other info*

Binary search organization

All entries in a table are assumed to satisfy an ordering relation. For example, use of the '<' relation implies that the symbol occupying an entry is 'smaller than' the symbol occupying the next entry. At any stage the search prediction in Algorithm 2.1 is that s occupies the middle entry of that part of the table which is expected to contain its entry.

Algorithm 2.2 (Binary search)

1. $start := 1$; $end := f$;
2. While $start \leq end$
   (a) $e := \lceil \frac{start+end}{2} \rceil$; where $\lceil \ldots \rceil$ implies a rounded quotient. Exit with success if $s = s_e$.
   (b) If $s < s_e$ then $end := e - 1$;
   else $start := e + 1$;
3. Exit with failure.

For a table containing f entries, we have $p_s \leq \lceil log_2 f \rceil$ and $p_u = \lceil log_2 f \rceil$. Thus the search performance is logarithmic in the size of the table. However, the requirement that the entry number of a symbol in the table should not change after an *add* operation (due to its use in the IC), forbids both additions and deletions during language processing. Hence, binary search organization is suitable only for a table

containing a fixed set of symbols, e.g. the table of keywords in a PL. For the same reason, it cannot be used for a symbol table unless one can afford a separate pass of the language processor for constructing the table.

Hash table organization

In the hash table organization the search prediction in Algorithm 2.1 depends on the value of s, i.e. e is a function of s. Three possibilities exist concerning the predicted entry—the entry may be occupied by s, the entry may be occupied by some other symbol, or the entry may be empty. The situation in the second case, i.e. $s \neq s_e$, is called a *collision*. Following a collision, the search continues with a new prediction. In the third case, s is entered in the predicted entry.

Algorithm 2.3 (Hash table management)

1. $e := h(s)$;
2. Exit with success if $s = s_e$, and with failure if entry e is unoccupied.
3. Repeat steps 1 and 2 with different functions h', h'', etc.

The function h used in Algorithm 2.3 is called a *hashing function*. We use the following notation to discuss the properties of hashing functions:

n	:	Number of entries in the table
f	:	Number of occupied entries in the table
ρ	:	*Occupation density* in the table, i.e. f/n
k	:	Number of distinct symbols in the source language
k_p	:	Number of symbols used in some source program
S_p	:	Set of symbols used in some source program
N	:	Address space of the table, i.e. the space formed by the entries $1 \ldots n$
K	:	Key space of the system, i.e. the space formed by enumerating all symbols of the source language. We will denote it as $1 \ldots k$
K_p	:	Key space of a program, i.e. $1 \ldots k_p$

A hashing function has the property that $1 \leq h(symb) \leq n$, where *symb* is any valid symbol of the source language. If $k \leq n$, we can select a one-to-one function as the hashing function h. This will eliminate collisions in the symbol table since entry number e given by $e = h(s)$ can only be occupied by symbol s. We refer to this organization as a *direct entry organization.*

However, k is a very large number in practice hence use of a one-to-one function will require a very large symbol table. Fortunately a one-to-one function is not

needed. For good search performance it is adequate if the hashing function implements a mapping $K_p \Rightarrow N$ which is nearly one-to-one for any arbitrary set of symbols S_p.

The effectiveness of a hashing organization depends on the average value of p_s. For a given size of a hash table, the value of p_s can be expected to increase with the value of k_p.

Hashing functions

While hashing, the representation of s, the symbol to be searched, is treated as a binary number. The hashing function performs a numerical transformation on this number to obtain e. Let the representation of s have b bits in it and let the host computer use m bit arithmetic. Now, to apply the numerical transformation, we need to obtain an m bit representation of s. Let us call it r_s. If $b \leq m$, the representation of s can be padded with 0's to obtain r_s. If $b > m$, the representation of s is split into pieces of m bits each, and bitwise exclusive OR operations are performed on these pieces to obtain r_s. This process is called *folding*. The hashing function h is now applied to r_s. This method ensures that the effect of all characters in a symbol is incorporated during hashing.

A hashing function h should possess the following properties to ensure good search performance:

1. The hashing function should not be sensitive to the symbols in S_p, that is, it should perform equally well for different source programs. Thus, the value of p_s should only depend on k_p.
2. The hashing function h should execute reasonably fast.

The first property is satisfied by designing a hashing function which is a good randomizer over the table space. This makes its performance insensitive to S_p. Two popular classes of such hashing functions are described in the following.

1. *Multiplication functions:* These functions are analogous to functions used in random number generation, e.g. $h(s) = (a \times r_s + b) \bmod 2^m$, where a, b are constants and fixed point arithmetic is used to compute $h(s)$. The table size should be a power of 2, say 2^g, such that the lower order g bits of $h(s)$ can be used as e.
2. *Division functions:* A typical division hashing function is

$$h(s) = (\textit{remainder of } \frac{r_s}{n}) + 1$$

where n is the size of the table. If n is a prime number, the method is called *prime division hashing*.

Both multiplication and division methods perform well in practice. A multiplication method has the advantage of being slightly faster but suffers from the drawback

that the table size has to be a power of 2. Prime division hashing is slower but has the advantage that prime numbers are much more closely spaced than powers of 2. This provides a wider choice of table size.

Collision handling methods

Two approaches to collision handling are to accommodate a colliding entry elsewhere in the hash table using a *rehashing* technique, or to accommodate the colliding entry in a separate table using an *overflow chaining* technique. We discuss these in the following paragraphs.

Rehashing

This technique uses a sequence of hashing functions $h_1, h_2, \ldots$ to resolve collisions. Let a collision occur while probing the table entry whose number is provided by $h_i(s)$. We use $h_{i+1}(s)$ to obtain a new entry number. This provides the new prediction in Algorithm 2.1. A popular technique called *sequential rehashing* uses the recurrence relation

$$h_{i+1}(s) \quad = \quad h_i(s) \bmod n + 1$$

to provide a series of hashing functions for rehashing.

A drawback of rehashing techniques is that a colliding entry accommodated elsewhere in the table may contribute to more collisions. This may lead to clustering of entries in the table.

Example 2.6 Let $h(\mathtt{a}) = h(\mathtt{b}) = h(\mathtt{c}) = 5$ and $h(\mathtt{d}) = 6$ in a language processor using sequential rehashing to handle collisions. If symbols are entered in a table in the sequence a, b, c, d, they would occupy the entries shown below:

Symbol	*Entry number*
a	5
b	6
c	7
d	8

While entering d, collisions occur in entries numbered 6 and 7 before d is accommodated in the 8^{th} entry. Entries 5-8 form a cluster of size 4. A symbol x such that $h(\mathtt{x}) = 5$ suffers 4 collisions due to the cluster. Thus, the average number of collisions for a colliding entry = $\frac{1}{2} \times$(average size of cluster+1).

Table 2.1 summarizes the performance of sequential rehashing. For values of ρ in the range 0.6–0.8, performance of the hash table is quite adequate. p_s has values in the range of 1.75 to 3.0, while p_u has values in the range 2.5 to 5.0. This performance is obtained at the cost of over-commitment of memory for the symbol table. Taking $\rho = 0.7$ as a practical figure, the table must have $k_p^m/0.7$ entries, where k_p^m is the largest value of k_p in a practical mix of source programs. If a program contains less symbols,

the search performance would be better. However, a larger part of the table would remain unused. Note that unlike the sequential and binary search organizations, the hash table performance is independent of the size of the table; it is determined only by the value of ρ.

Table 2.1 Performance of sequential rehash

ρ	p_u	p_s
0.2	1.25	1.125
0.4	1.67	1.33
0.6	2.5	1.75
0.8	5.0	3.0
0.9	10.0	5.5
0.95	20.0	10.5

Hash table performance can be improved by reducing the clustering effect. Various rehashing schemes like sequential step rehash, quadratic and quadratic quotient rehash have been devised with this in view. Bell (1970) and Ackerman (1974) deal with these schemes. Price (1971) and Dhamdhere (1983) contain comprehensive treatments of hash table organizations.

Overflow chaining

Overflow chaining avoids the problems associated with the clustering effect (see Ex. 2.6) by accommodating colliding entries in a separate table called the *overflow table*. Thus, a search which encounters a collision in the primary hash table has to be continued in the overflow table. To facilitate this, a pointer field is added to each entry in the primary and overflow tables. The entry format is as follows:

Symbol	*Other info*	*Pointer*

A single hashing function h is used. All symbols which encounter a collision are accommodated in the overflow table. Symbols hashing into a specific entry of the primary table are chained together using the pointer field. On encountering a collision in the primary table, that is, in step 3 of Algorithm 2.3, one chain in the overflow table has to be searched.

Example 2.7 Figure 2.4 shows the organization of a hash table with overflow chaining. Symbols a, b, c and d of Ex. 2.6 are entered in the table as follows: a is entered in the 5^{th} entry of the hash table. Symbol b collides with it. Hence b is put in the next available entry of the overflow table and the *pointer* field of the 5^{th} entry of the hash table is set to point at it. c collides with a in the hash table and with b in the overflow table. A new entry is created for c and the *pointer* field of b's entry is set to point to it. d does not suffer a collision.

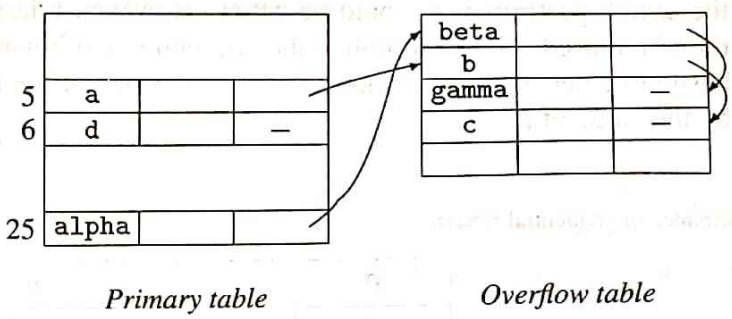


Fig. 2.4 Hash table organization with overflow chaining

Symbols `alpha`, `beta` and `gamma` form another chain since they all hash into entry 25 of the hash table.

The main drawback of the overflow chaining method is the extra memory requirement due to the presence of the overflow table, which must have $(k_p^m - 1)$ entries. An organization called scatter table organization is often used to reduce the memory requirements. In this organization, the hash table merely contains pointers, and all symbol entries are stored in the overflow table. Effectively, the hash table is merely a routing table (hence the name *scatter table*). Now the hash table should be large to ensure low values of ρ in it, and the overflow table needs to contain k_p^m entries.

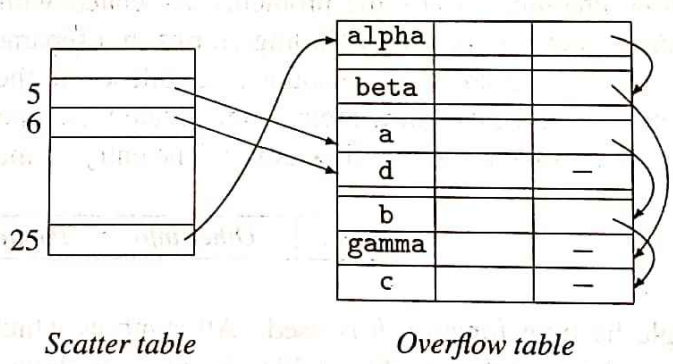


Fig. 2.5 Scatter table organization

Example 2.8 Figure 2.5 shows the scatter table organization for the symbols of Ex. 2.7. If each pointer requires 1 word and each symbol entry requires l words, the memory requirements are

$$\begin{aligned} &= \frac{k_p^m}{0.7} + k_p^m \times l \text{ words} \\ &= k_p^m \times (1.43 + l) \text{ words} \\ &\simeq k_p^m \times l \text{ words} \end{aligned}$$

This compares favourably with the memory requirements if a rehashing technique was used to handle collisions, viz.

$$= (\frac{k_p^m}{0.7}) \times l \text{ words}$$
$$\cong 1.43\, k_p^m \times l \text{ words}$$

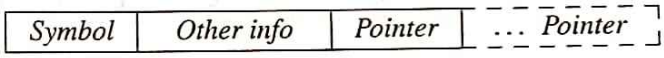
Memory requirements in the case of hash with overflow chaining (see Ex. 2.7) would have been

$$= (\frac{k_p^m}{0.7}) \times l + (k_p^m - 1) \times l \text{ words}$$
$$\cong 2.43\, k_p^m \times l \text{ words}$$

Note that both hash with overflow and scatter table organizations would perform better than the sequential rehash organization for the same value of ρ.

2.1.2 Linked List and Tree Structured Organizations

Linked list and tree structured organizations are nonlinear in nature, that is, elements of the search data structure are not located in adjoining memory areas. To facilitate search, each entry contains one or more pointers to other entries.



These organizations have the advantage that a fixed memory area need not be committed to the search structure. The system simply allocates a *header* element to start with. This element points to the first entry in the linked list or to the root of the tree. Other entries are allocated as and when needed.

Linked lists

Each entry in the linked list organization contains a single pointer field. The list has to be searched sequentially for obvious reasons. Hence its search performance is identical with that of sequential search tables, i.e. $p_s = l/2$ and $p_u = l$.

Binary trees

Each node in the tree is a symbol entry with two pointer fields—the *left_pointer* and the *right_pointer*. The following relation holds at every node of the tree: If s is the symbol in the entry, the left pointer points to a subtree containing all symbols $< s$ while the right pointer points to a subtree containing all symbols $> s$. This relation is used by the search procedure in a manner analogous to Algorithm 2.2. Algorithm 2.4 contains the search procedure. We use the notation $x.y$ to represent field y of node x and the notation $(p)^*.y$ to represent field y of the node pointed to by pointer p.

Algorithm 2.4 (Binary tree search)

1. *current_node_pointer* := address of root of the binary tree;

2. If $s = (current\_node\_pointer)^*.symbol$, then exit with success;
3. If $s < (current\_node\_pointer)^*.symbol$ then
   current_node_pointer := (*current_node_pointer*)*.*left_pointer*;
   else *current_node_pointer* := (*current_node_pointer*)*.*right_pointer*;
4. If *current_node_pointer* = *nil* then
   exit with failure.
   else goto Step 2.

The best search performance is obtained when the tree is balanced. This performance is identical with that of the binary search table. In the worst case, the tree degenerates to a linked list and the search performance is identical with sequential search.

Example 2.9 Parts (a)-(d) of Figure 2.6 show various stages in the building of a binary tree when symbols are entered in the sequence p, c, t, f, h, k, e. Part (e) shows a balanced binary tree with the same symbols which gives better search performance.

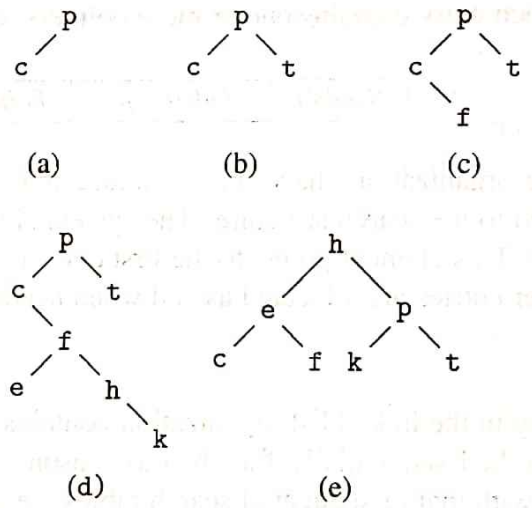


Fig. 2.6 Tree structured table organization

Nested search structures

Nested search structures are used when it is necessary to support a search along a secondary dimension within a search structure. Since the search structure cannot be linear in the secondary dimension, a linked list representation is used for the secondary search. Examples of nested search structures can be found in the handling of records of Pascal, PL/1, Cobol, etc., structures of C or handling of procedure parameters in any source language. Nested search structures are also known as *multi-list structures*.

Example 2.10 Figure 2.7 shows the symbol table entries for the Pascal record

```
personal_info  : record
                 name : array [1..10] of character;
                  sex : character;
                   id : integer;
                 end;
```

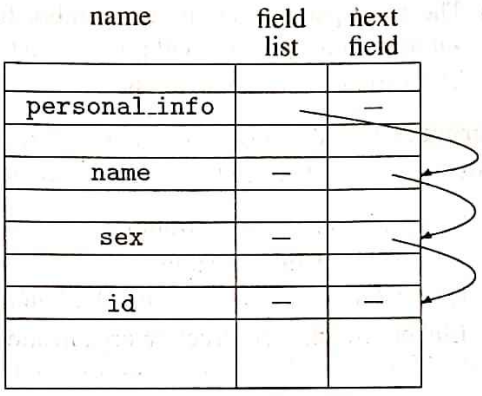


Fig. 2.7 Multi-list structure

Each symbol table entry contains two additional fields, *field list* and *next field*. *field list* of `personal_info` points to the entry for `name`. *next_field* of `name` points to `sex`, etc. This organization permits the field `name` to be searched in the primary as well as secondary dimensions, i.e., as a symbol in the table as well as a field of `personal_info`. This enables efficient handling of occurrences like `name` and `personal_info.name` in the Pascal program.

EXERCISE 2.1

1. Comment on the feasibility of designing a hashing function to implement a direct entry organization for a fixed set of symbols. (Sprugnoli (1977) and Lyon (1978) report some related work).
2. An assembler is to be designed to handle programs containing up to 500 symbols. However, it is found that an average program contains only about 200 symbols. It is decided that a hash organization with sequential rehashing must be used. Evaluate the following three design alternatives in terms of memory requirements and access efficiency.
   (a) Size of the table = 550 entries
   (b) Size of the table = 700 entries
   (c) Size of the table = 500 entries
3. An assembler supports an option by which it produces an alphabetical listing of all symbols used in a program. Comment on the suitability of the following organizations

to organize the symbol table:

(a) Binary search organization
(b) Linked list organization
(c) Binary tree organization
(d) A multi-list organization

4. Many language processors use the concept of split-entry symbol tables. Each entry in such a table consists of two parts:

   (a) The fixed part containing the symbol field, a tag field, fields common to all variants, and a pointer field pointing to the variant part.
   (b) The variant part containing the fields relevant to the variant.

   Comment on the advantages of this organization.
5. Discuss the problem of deletion of entries in the following symbol table organizations:

   (a) Sequential search organization
   (b) Binary search organization
   (c) Hash table organizations with rehash and overflow chaining techniques
   (d) Linked list and tree structure organizations.

2.2 ALLOCATION DATA STRUCTURES

We discuss two allocation data structures, stacks and heaps.

2.2.1 Stacks

A *stack* is a linear data structure which satisfies the following properties:

1. Allocations and deallocations are performed in a *last-in-first-out* (LIFO) manner—that is, amongst all entries existing at any time, the first entry to be deallocated is the last entry to have been allocated.
2. Only the last entry is accessible at any time.

Figure 2.8 illustrates the stack model of allocation. Being a linear data structure, an area of memory is reserved for the stack. A pointer called the *stack base* (SB) points to the first word of the stack area. The stack grows in size as entries are created, i.e. as they are allocated memory. (We shall use the convention that a stack grows towards the higher end of memory. We depict this as downwards growth in the figures.) A pointer called the *top of stack* (TOS) points to the last entry allocated in the stack. This pointer is used to access the last entry. No provisions exist for access to other entries in the stack. When an entry is *pushed* on the stack (i.e. allocated in the stack), TOS is incremented by l, the size of the entry (we assume $l = 1$). When an entry is *popped*, i.e. deallocated, TOS is decremented by l (see Figs. 2.8(a)-(c)). To start with, the stack is *empty*. An empty stack is represented by TOS = SB−1 (see Fig. 2.8(d)).

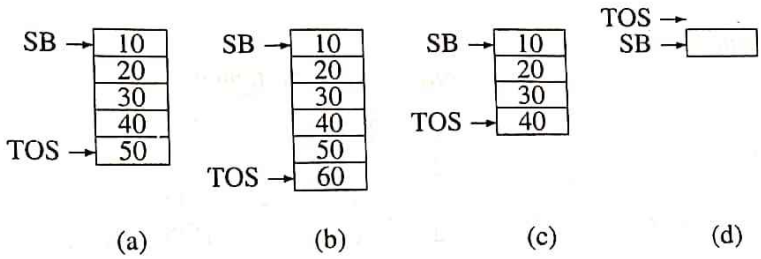


Fig. 2.8 Stack model of allocation

Extended stack model

The LIFO nature of stacks is useful when the lifetimes of the allocated entities follow the LIFO order. However, some extensions are needed in the simple stack model because all entities may not be of the same size. The size of an entity is assumed to be an integral multiple of the size of a stack entry. To allocate an entity, a *record* is created in the stack, where the record consists of a set of consecutive stack entries. For simplicity, the size of a stack entry, i.e. l, is assumed to be one word. Figure 2.9(a) shows the extended stack model. In addition to SB and TOS, two new pointers exist in the model:

1. A *record base pointer* (RB) pointing to the first word of the last record in stack.
2. The first word of each record is a *reserved pointer*. This pointer is used for housekeeping purposes as explained below.

The allocation and deallocation time actions in the extended stack model are described in the following paragraphs (see Fig. 2.9(b)–(c)).

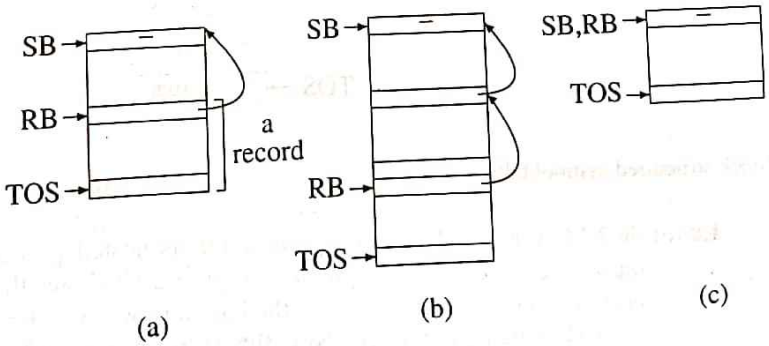


Fig. 2.9 Extended stack model

Allocation time actions

No.			*Statement*
1.	TOS	:=	TOS + 1;
2.	TOS*	:=	RB;
3.	RB	:=	TOS;
4.	TOS	:=	TOS + *n*;

The first statement increments TOS by one stack entry. It now points at the *reserved pointer* of the new record. The '*' mark in statement 2 indicates indirection. Hence the assignment TOS* := RB deposits the address of the previous record base into the reserved pointer. Statement 3 sets RB to point at the first stack entry in the new record. Statement 4 performs allocation of n stack entries to the new entity (see Fig. 2.9(b)). The newly created entity now occupies the addresses <RB> +l to <RB> +$l \times n$, where <RB> stands for 'contents of RB'.

Deallocation time actions

No.			*Statement*
1.	TOS	:=	RB − 1;
2.	RB	:=	RB*;

The first statement pops a record off the stack by resetting TOS to the value it had before the record was allocated. RB is then made to point at the base of the previous record (see Fig. 2.9(c)).

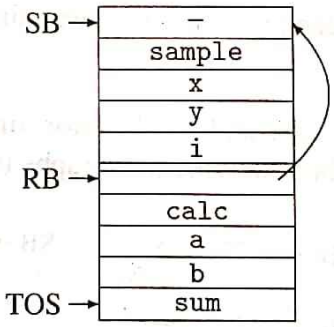


Fig. 2.10 Stack structured symbol table

Example 2.11 When a Pascal program contains nested procedures, many symbol tables must co-exist during compilation. Figure 2.10 shows the symbol tables of the main program and procedure `calc` of the Pascal program of Ex. 2.2 when the statement `sum := a+b` is being compiled. Note the address contained in the *reserved pointer* in the symbol table for procedure `calc`. It is used to pop the symbol table of `calc` off the stack after its `end` statement is compiled.

2.2.2 Heaps

A heap is a nonlinear data structure which permits allocation and deallocation of entities in a random order. An allocation request returns a pointer to the allocated area in the heap. A deallocation request must present a pointer to the area to be deallocated. The heap data structure does not provide any specific means to access an allocated entity. It is assumed that each user of an allocated entity maintains a pointer to the memory area allocated to the entity.

Example 2.12 Figure 2.11 shows the status of a heap after executing the following C program

```
float *floatptr1, *floatptr2;
int *intptr;
floatptr1 = (float *) calloc(5, sizeof(float));
floatptr2 = (float *) calloc(2, sizeof(float));
intptr = (int *) calloc(5, sizeof(int));
free(floatptr2);
```

Three memory areas are allocated by the calls on `calloc` and the pointers `floatptr1`, `floatptr2` and `intptr` are set to point to these areas. `free` frees the area allocated to `floatptr2`. This creates a 'hole' in the allocation. Note that following Section 2.1, each allocated area is assumed to contain a *length* field preceding the actual allocation.

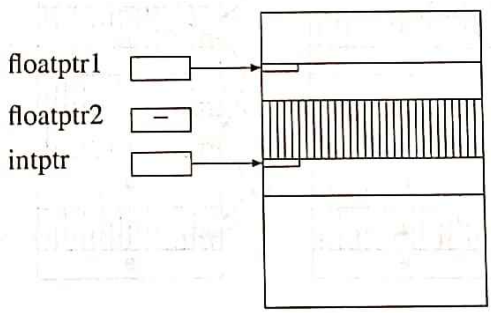


Fig. 2.11 Heap

Memory management

Example 2.12 illustrates how free areas (or 'holes') develop in memory as a result of allocations and deallocations in the heap. Memory management thus consists of identifying the free memory areas and reusing them while making fresh allocations. Speed of allocation/deallocation, and efficiency of memory utilization are the obvious performance criteria of memory management.

Identifying free memory areas

Two popular techniques used to identify free memory areas are:

1. Reference counts
2. Garbage collection.

In the reference count technique, the system associates a *reference count* with each memory area to indicate the number of its active users. The number is incremented when a new user gains access to that area and is decremented when a user finishes using it. The area is known to be free when its reference count drops to zero. The reference count technique is simple to implement and incurs incremental overheads, i.e. overheads at every allocation and deallocation. In the latter technique, the system performs garbage collection when it runs out of memory. Garbage collection makes two passes over the memory to identify unused areas. In the first pass it traverses all pointers pointing to allocated areas and *marks* the memory areas which are in use. The second pass finds all unmarked areas and declares them to be *free*. The garbage collection overheads are not incremental. They are incurred every time the system runs out of free memory to allocate to fresh requests.

To manage the reuse of free memory, the system can enter the free memory areas into a *free list* and service allocation requests out of the free list. Alternatively, it can perform *memory compaction* to combine these areas into a single free area.

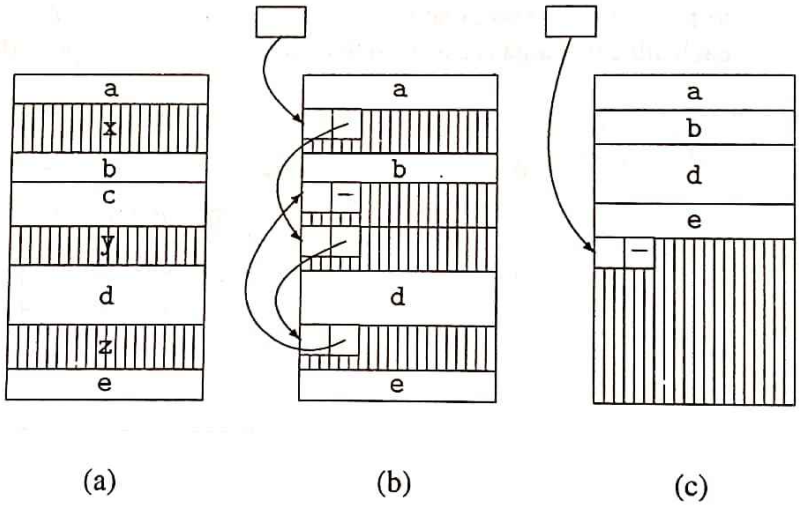


Fig. 2.12 (a) allocation status of heap, (b) free list, (c) after compaction

Example 2.13 Figure 2.12 shows free area management using free lists and memory compaction. Part (a) shows five areas named a-e in active use, and three free areas named x, y and z. The system has a *free area descriptor* permanently allocated. If a free list is to be used to facilitate memory allocation to fresh requests, the descriptor is used as a list header for the free list. The first word in each area is used to hold the count of words in the area and a pointer to the next free area in the list. Part (b) shows how area c is added to the free list consisting of x, y and z. If memory compaction is used, the *free area descriptor* describes the single free area resulting from com-

paction. The first word in this area contains a count of words in the area and a *null* pointer. Figure 2.12(c) shows the results of memory compaction.

Reuse of memory

When memory compaction is used, fresh allocations are made from the block of free memory. The *free area descriptor* and the count of words in the free area are updated appropriately. When a free list is used, two techniques can be used to perform a fresh allocation:

1. First fit technique
2. Best fit technique.

The first fit technique selects the first free area whose size is $\geq n$ words, where n is the number of words to be allocated. The remaining part of the area is put back into the free list. This technique suffers from the problem that memory areas become successively smaller, hence requests for large memory areas may have to be rejected. The best fit technique finds the smallest free area whose size $\geq n$. This enables more allocation requests to be satisfied. However, in the long run it, too, may suffer from the problem of numerous small free areas.

Example 2.14 Let the free list consist of two areas called $area_1$ and $area_2$ of 500 words and 200 words, respectively. Let allocation requests for 100 words, 50 words and 400 words arise in the system. The first fit technique will allocate 100 words from $area_1$ and 50 words from the remainder of $area_1$. The free list now contains areas of 350 words and 200 words. The request for 400 words cannot be granted. The best fit technique will allocate 100 words and 50 words from $area_2$, and 400 words from $area_1$. This leaves areas of 100 words and 50 words in the free list

Knuth (1973) discusses methods of overcoming the problems of first fit and best fit techniques. The *buddy* method may be used to merge adjoining free areas into larger free areas.

BIBLIOGRAPHY

(a) Data structures, general

1. Aho, A.V. and J.D. Ullman (1984): *Data Structures*, Addison-Wesley, New York.
2. Horowitz, E. and S. Sahni (1983): *Fundamentals of Data Structures*, Computer Science Press, California and Galgotia, New Delhi.
3. Knuth, D.E. (1985): *The Art of Computer Programming : Vol. I*, Addison-Wesley, Reading and Narosa, New Delhi.
4. Wirth, N. (1976): *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs.

(b) Table management

Table management crucially determines the speed of language processing. Most books on compilers include a good coverage of table management techniques. Gries (1971) and

Dhamdhere (1983) contain comprehensive treatments of this topic. Price (1977) is an important review article on table management. Horowitz and Sahni (1983) and Tremblay and Sorenson (1983) are good sources on algorithms for table management.

1. Ackermann, A.F. (1974): "Quadratic search for hash tables of size p^n," *Commn. of ACM*, **17** (3), 164-165.
2. Bell, J.R. (1970): "The quadratic quotient method," *Commn. of ACM*, **13** (2), 107-109.
3. Dhamdhere, D.M. (1983): *Compiler Construction – Principles and Practice*, Macmillan India, New Delhi.
4. Donovan, J.J. (1972): *Systems Programming*, McGraw-Hill Kogakusha, Tokyo.
5. Gries, D. (1971): *Compiler Construction for Digital Computers*, Wiley, New York.
6. Horowitz, E. and S. Sahni (1983): *Fundamentals of Data Structures*, Computer Science Press, California.
7. Knuth, D. (1973): *The Art of Computer Programming, Vol. III – Sorting and Searching*, Addison-Wesley, Reading.
8. Lyon, G. (1978): "Packed scatter tables," *Commn. of ACM*, **21** (10), 857-865.
9. Price, C.E. (1971): "Table lookup techniques," *Computing Surveys*, **3** (2), 49-65.
10. Sprugnoli, R. (1977): "Perfect hashing functions – a single probe retrieval method for static sets." *Commn. of ACM*, **20** (11), 841-850.
11. Tremblay, J.P. and P.G. Sorenson (1983): *An Introduction to Data Structures with Applications*, McGraw-Hill.