SETL NEWSLETTER No. 68 S. Brown

Some Thoughts on Efficient Programming in SETLB.

October 26, 1972

The following suggestions apply to the current BALM implementation of SETLB and may not be applicable to future implementations.

It has been noted that some seemingly simple SETLB operations require large amounts of core. Specific examples can be brought to room 324 for analysis. However most programs can probably be made smaller by following the suggestions and by understanding the internal representation of SETL objects.

To compile and execute a SETLB program actually requires running two programs. The first is the SETLB preprocessor. It reads the SETLB program and translates it into BALMSETL. The second program is BALM. BALM reads the BALMSETL and produces 6600 code which is then executed. It is important for the SETLB user to udnerstand the use of COMPUTE;. Compute; actually has no meaning in SETLB but it is important to BALM. Each DO; COMPUTE; marks off a section of program which BALM will translate into 6600 code before reading the next line of input. Naturally the larger the section of code the more core space BALM will need in order to produce 6600 code. Once BALM has finished all extra space is released and can be used again for the next section of code. To minimize the amount of space BALM needs at one time the SETLB user should follow the practice of placing a DO; COMPUTE; around each SETLB procedure so that his program contains as many sections as possible.

Although unused core is reclaimed by the BALM garbage collector it is a good idea to avoid creating unnecessary copies of SETL items. For example:

```
S = X WITH. S;    and    X IN. S;
```

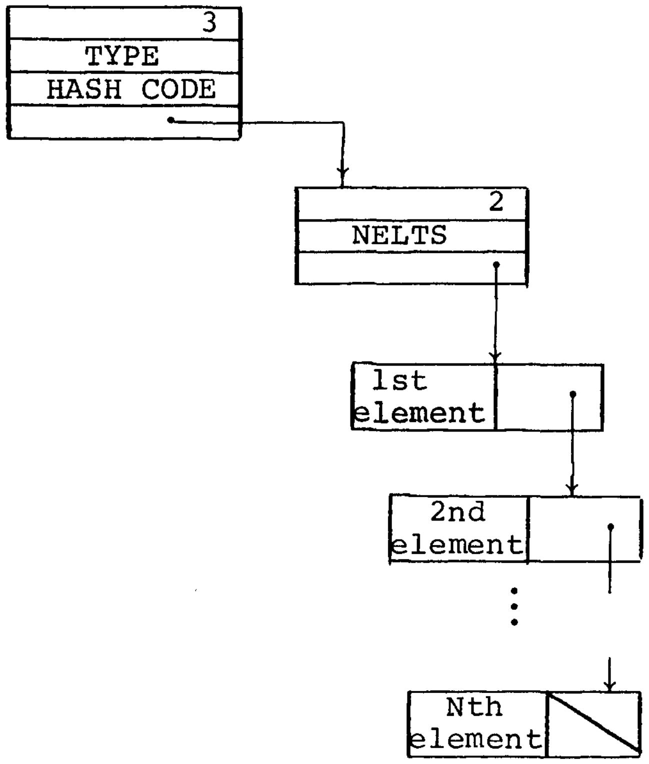
have equivalent results but in the first case a copy is made of s so that before the operation is complete enough core for two sets is required. The same is true of:

S = S LESS. X: and X OUT. S;

The internal representation of SETL items is very similar to the description in SETL Newsletter 49. Perhaps a knowledge of internal representation will be helpful in determining how to structure data and selecting operations.

The internal representation for a tuple is as follows:

TUPLE INTERNAL REPRESENTATION



The + operator produces a copy of each tuple whereas the component operation acts on the tuple itself. To add an element to a tuple the following are equivalent:

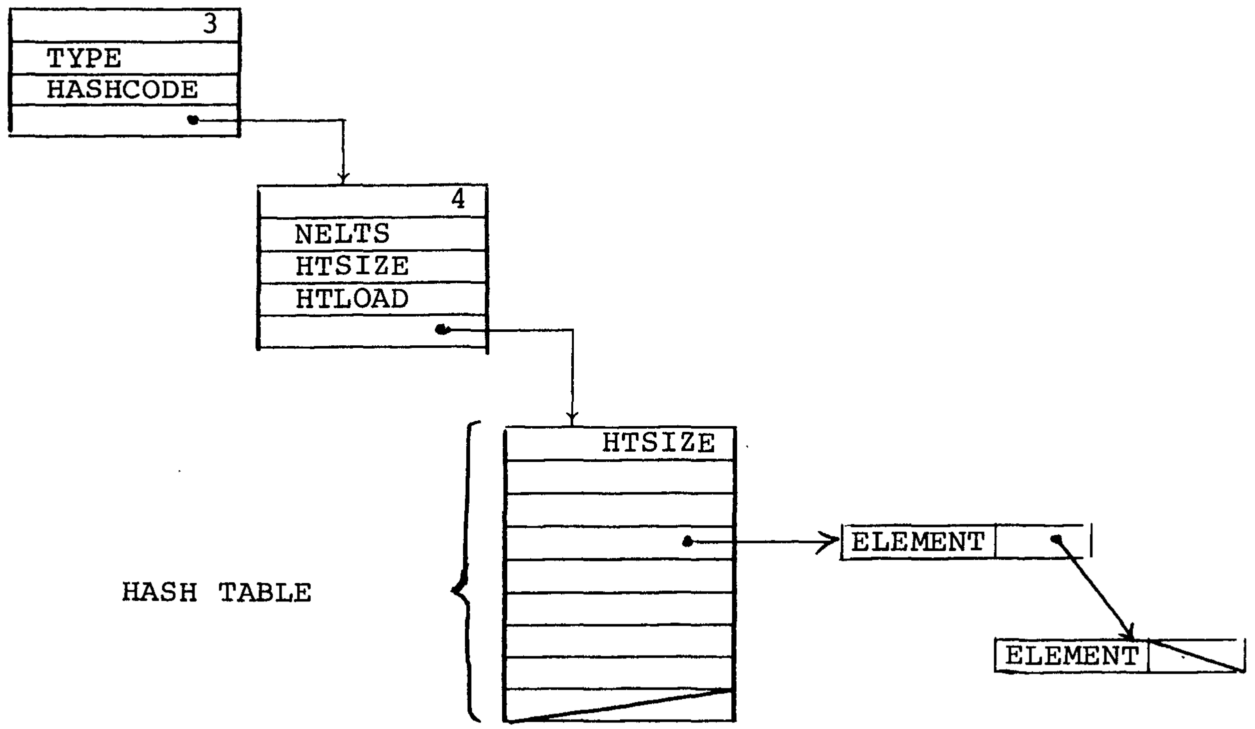
T = T + <NEW>; and

T((↓ T) + 1) = NEW;

But the second does not require a copy of T to be made during the operation.

Internal representation of sets is as follows.

SET INTERNAL REPRESENTATION

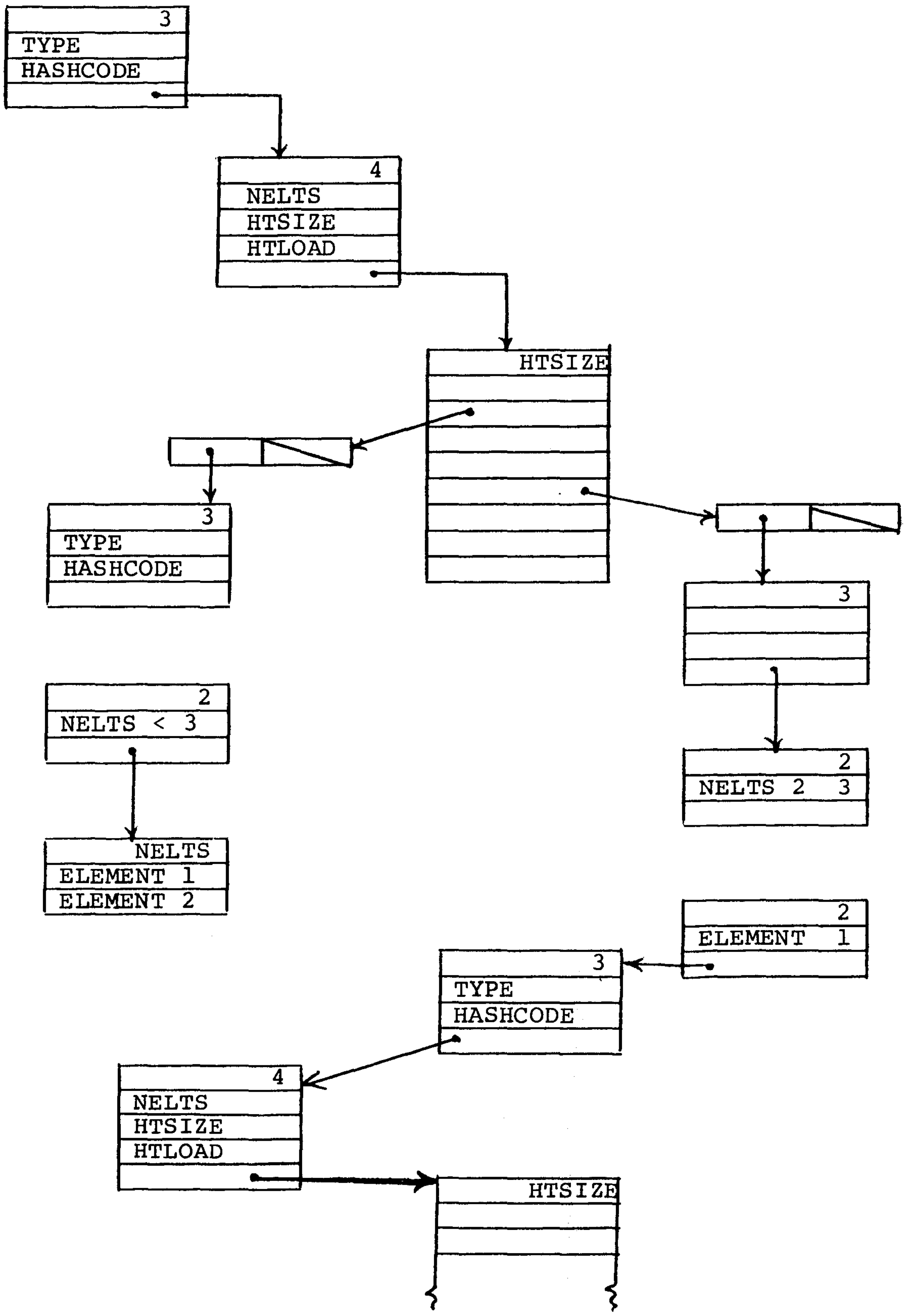


Each hashtable entry contains a pointer to the list of elements whose hashcode is that entry.

There is more overhead in the construction of a set than a tuple. Hashtable sizes are 8, 16, 32, 64, 128 and 256. This means that a set of only 1 element has a hashtable with 8 entries. Small sets take up a disproportionate amount of space. The hashtable size is determined by the number of elements in the set. If the number of elements increases or decreases beyond the next higher or lower hashtable size then a new hashtable is created and the elements are rehashed. Therefore sets which vary in size greatly throughout a program require extra amounts of space from time to time.

The internal representation of a set whose elements are tuples is as follows.

INTERNAL REPRESENTATION OF A SET WHOSE ELEMENTS ARE TUPLES



Note that a tuple is represented differently when it is a member of a set than when it is simply an item. This means that operations which test for membership or examine members of a set must convert a tuple from one representation to the other. These conversions require extra space during the operation.

The representation of tuples of more than 2 elements is detailed in SETL Newsletter No. 49. It should be noted that it is space consuming for sets of tuples whose first elements are different.