CC WEEK 10

Prepared for: 7th Sem, CE, DDU
Prepared by: Niyati J. Buch

Introduction to Symbol Table

- **Symbol table** is an essential data structure used by compilers to remember information about identifiers in the source program.
  - Identifiers like variables, procedures, functions, constants, labels, structures, etc.

- Good representation of symbol table → fast access of symbol table → fast compiler

- Symbol table must have efficient representation of scoped variables (global & local)

Introduction to Symbol Table

- Usually lexical analyzer and parser fill up the entries in the table.
  - Whenever the lexical analyser comes with a new token, if it finds it is an identifier, it installs it in the symbol table and returns the index of that symbol table as an attribute to the token or ID.
  - Parser added information like type of identifier.

- E.g. int x, y, z;
  - Tokens which identifiers like x y z are added to the symbol table by lexical analyzer
  - Its type INT will be added by parser

Introduction to Symbol Table

- Code generator and optimizer makes use of this information stored in the Symbol table.

- The symbol tables may vary from implementation to implementation even for the same language.

Information in Symbol Table

- **Name**
  - Name of the identifier
  - may be stored directly or as a pointer to another character string in an associated string table.

- **Type**
  - Type of the identifier like variable, label, procedure name, etc
  - For variables
    - <u>Basic</u> type like integer, real, float, etc
    - <u>Derived</u> type

Information in Symbol Table

- **Location**
  - offset within the program where the identifier is defined.
- **Scope**
  - Region of the program where the current definition is valid
- **Other attributes**
  - Array limits, parameters, return values, etc

Usage of Symbol Table information

- **Semantic Analysis**
  - Check correct semantic usage of language constructs
  - E.g. types of identifiers
    - x = y Are the types compatible for assignment?
    - a + b Are the types compatible for + operator?

- **Code Generation**
  - Types of variables provide their sizes
  - As individual variable needs to be allocated space during code generation

Usage of Symbol Table information

- **Error Detection**
  - Undefined variables
  - Recurrence of error messages can be avoided by marking the variable types as undefined
  - E.g. if a variable **x** is used in the program and its type is not defined then this will give error <u>everywhere</u> **x** is used.
  - Solution can be at first place where it found that x is undefined, add the type in symbol table as <u>undefined</u>

- **Optimization**
  - Two or more temporaries can be merged if their types are same and they need not be given space simultaneously .

Operations on the Symbol Table

- **Lookup**
  - Most frequent operation
  - Whenever an identifier is seen, it is needed to **check** its types, or **create** a new entry

- **Insert**
  - Second important operation
  - **Adding** new names to the table, happens mostly in lexical and syntax analysis phases

Operations on the Symbol Table

- **Modify**
  - When a name is defined, all the information may not be available.
  - The information may be **updated** later.

- **Delete**
  - Not very frequent
  - will occur particularly when any procedure ends .

Issues for Symbol Table design

- **Format of the entries**
  - Various formats like linear array, tree structure, table, etc
- **Access methodology**
  - Linear search, Binary search, Tree search, Hashing, etc.
- **Location of storage**
  - Primary memory, partial storage in secondary memory
- **Scope Issues**
  - In block-structured language, a variable defined in upper blocks must be visible to inner blocks
  - Not vice versa.

Simple Symbol Table

- Commonly used techniques:
  - Linear table
  - Ordered list (language dependent)
  - Tree (binary tree or similar)
  - Hash table

- Works well for single scope languages
  - All variables have single scope.
  - All variables are global.
  - It is not dependent on the position of the program at which those variables are defined.

Linear Table

- It is a simple array of records corresponding to an identifier in the program.

- **Example**:

  int x, y

  real z

  …

  procedure abc

  …

  L1: …

  …

Name	Type	Location
x	integer	Offset of x
y	integer	Offset of y
z	real	Offset of z
abc	procedure	Offset of abc
L1	label	Offset of L1

Linear Table

- If there is no restriction in the length of the string for the name of an identifier, a string table may be used with the name field holding the pointers.
- **Lookup**, insert, modify take O(n) time
- Insertion can be made O(1) by remembering the pointer to the next free index.
- Scanning most recent entries first may probably speed up the access
  - Due to program locality, a variable defined just inside a block is expected to be referred to more often than some earlier variables.

When to use linear table?

- An unordered table organization should be used only if the **expected size** of the symbol table is small,
  - since the average time for lookup and insertion is directly proportional to the table size.

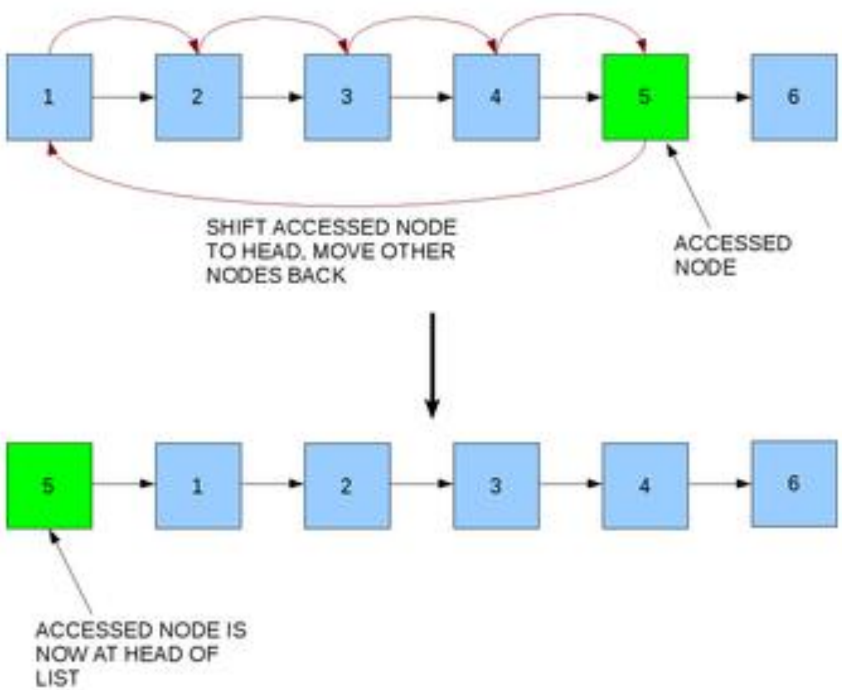
Ordered List

- It is a variation of linear tables in which the list organization is used.
- List is sorted and then **binary search** can be used with O(log n) time.
  - Lexical sorting on the variable/identifier name
- Insertion needs more time.
  - O(n)

Ordered List

- A variant of ordered list – self organizing list
  - A self-organizing list is a list that reorders its elements based on some self-organizing heuristic to improve average access time.
  - The aim of a self-organizing list is to improve efficiency of linear search by moving more frequently accessed items towards the head of the list.
  - A self-organizing list achieves near constant time for element access in the best case.

Techniques for rearranging the node

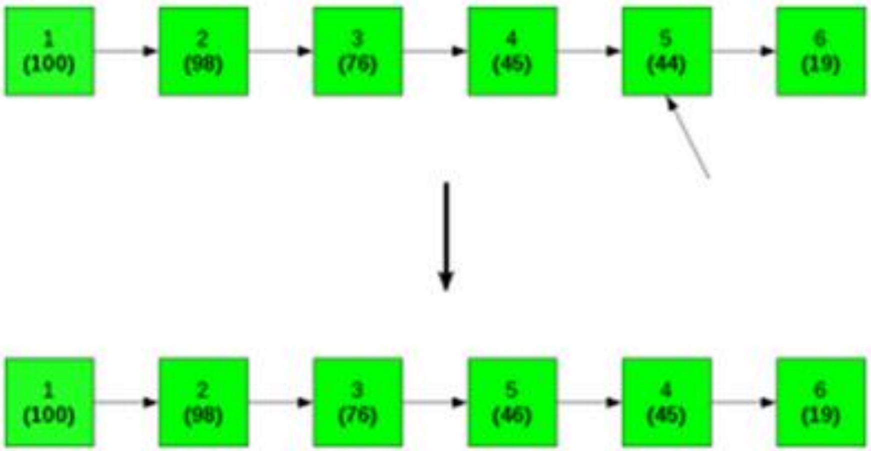


If the 5th node is selected, it is moved to the front

- **Move to front method (MTF)**
  - moves the element which is accessed to the head of the list.

https://en.wikipedia.org/wiki/Self-organizing_list

Techniques for rearranging the node

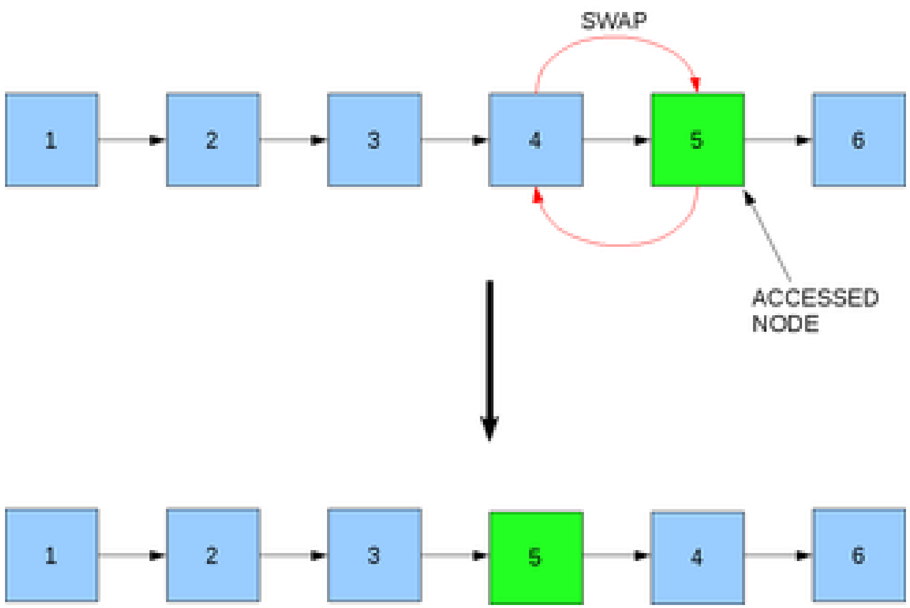


If the 5th node in the list is searched for twice, it will be swapped with the 4th

- **Count method**
  - The number of times each node was searched for is counted i.e. every node keeps a separate counter variable which is incremented every time it is called.
  - The nodes are then rearranged according to decreasing count.
  - Thus, the nodes of highest count i.e. most frequently accessed are kept at the head of the list

https://en.wikipedia.org/wiki/Self-organizing_list

Techniques for rearranging the node



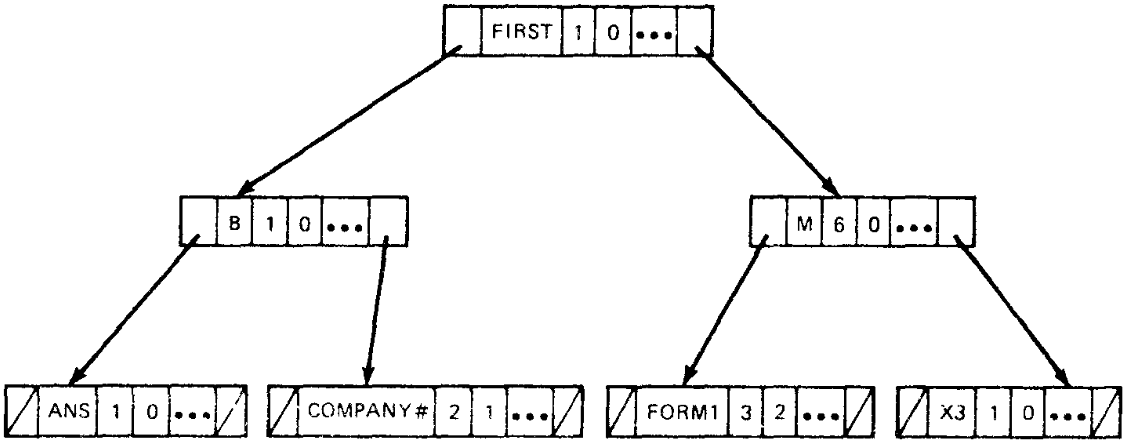
If the 5th node in the list is selected, it will be swapped with the 4th

- **Transpose Method**
  - This technique involves swapping an accessed node with its predecessor.
  - Therefore, if any node is accessed, it is swapped with the node in front unless it is the head node, thereby increasing its priority.

https://en.wikipedia.org/wiki/Self-organizing_list

Tree

- Each entry is represented by a node of the tree
- Based on string comparison of the names,
  - entries lesser than a reference node are kept in the left sub-tree,
  - entries greater than a reference node are kept in the right sub-tree.



- Because link fields define only the logical relationships between the table records, we are free to store the records in a contiguous area and in any order, provided the correct structural relationships between nodes are preserved.

Table Position	Name Field	Type	Dimension	Other Attributes	Left Pointer	Right Pointer
1	FIRST	1	0		2	5
2	B	1	0		3	4
3	ANS	1	0		0	0
4	COMPANY #	2	1		0	0
5	M	6	0		6	7
6	FORM1	3	2		0	0
7	X3	1	0		0	0

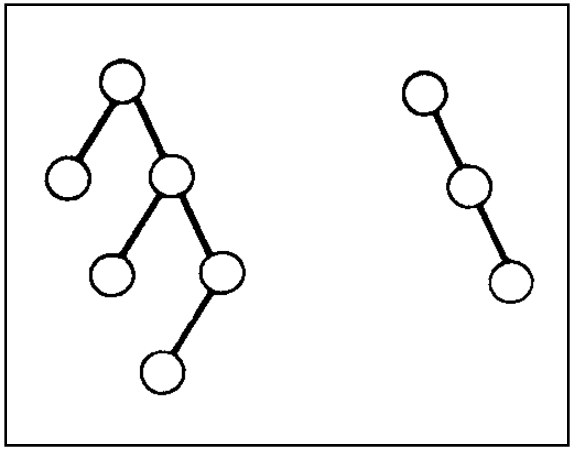
Type is encoded real=1, integer=2, char=3, procedure=6, etc

- A representation in which table records are physically adjacent.
- The pointer-field value of zero represents a NULL structural link.
- The root node is always located at the first record location in the table.

Points to consider while using tree

- Average lookup time is O(log n)
  - As it is binary search tree

- Worst case O(n)
  - Unbalanced trees
    - Left or right heavy tree
  - Solution??



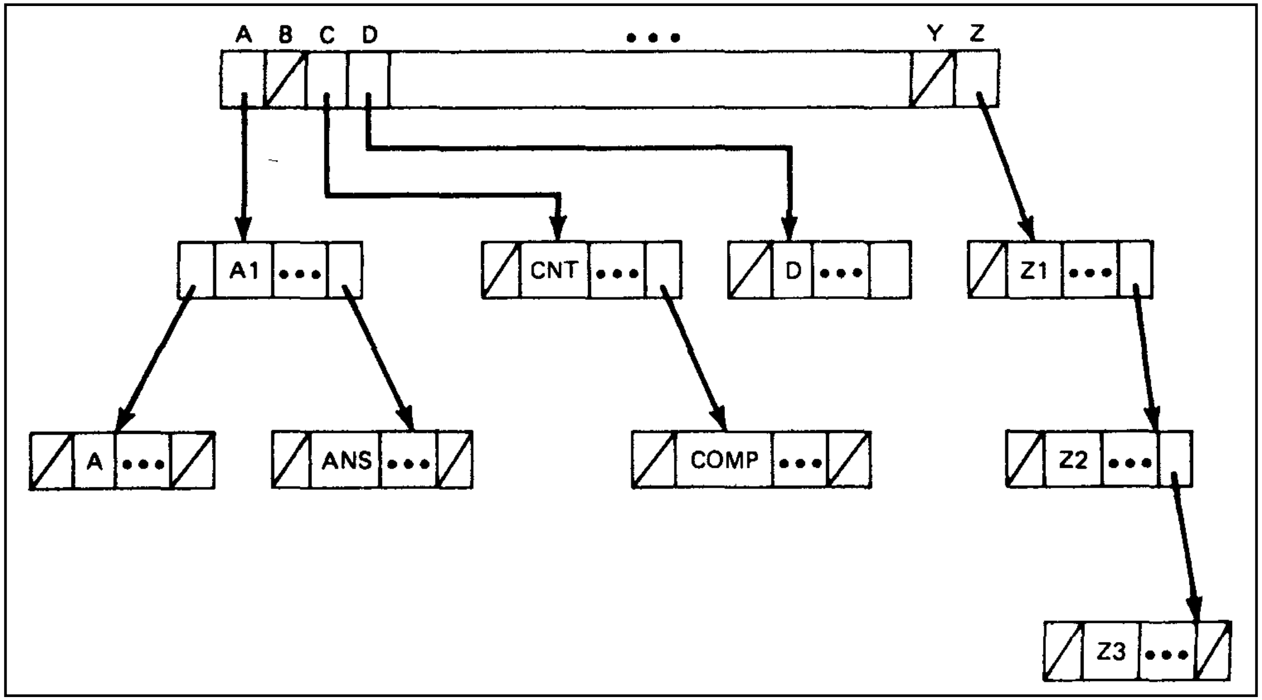
Points to consider while using tree

- Use balancing techniques
  - AVL tree
  - B tree
  - Red Black tree
  - Day–Stout–Warren (DSW) algorithm*
  - Etc.

*http://www.eecs.umich.edu/~qstout/pap/CACM86.pdf

Tries

- When a large number of variables appear in a symbol table, a special m-ary type of tree structure called a **trie** be used in conjunction with a binary tree.
- The basic idea is to use a **trie** node with 26 link fields (one for each letter in the alphabet) at the first level and to use a binary tree structure at lower levels of the structure.



Hash Table

- Most common method for implementing symbol tables in compilers
  - It minimizes the access time to O(1)
- Mapping is done using **hash function** that results in a unique location in the table organized as an array

- For numeric data, hash function like mod can be used.
- For strings (names of symbols in symbol table) summation of ASCII values can be taken and then result can be used with some hash function like mod and to get the index of array/table.

Hash Table

- Several symbols may be mapped to the same location
  - Collision resolution strategy is needed
    - Open addressing
    - Chaining
  - To keep the collisions reasonable, the size of the hash table is chosen to be between n and 2n for n keys.

Desirable Properties of Hash Function

- **Hash function** should depend on the name of the symbol.
- Equal emphasis should be given to each part of the name.
- **Hash function** should be quickly computable.
- **Hash function** should have uniform mapping of names to different parts of the table.
  - Similar names (like data1, data2) should not cluster to the same address
- The computed value must be in the range of the table index.

Summary so far

- If it is known that only a few variables (i.e., 10 or fewer) are going to appear in a program, **an unordered table** should be considered. (very rare)
- Adopting a binary-search strategy for **an ordered table** can improve the lookup operation if more than 25 records occur.
  - Ordering the table facilitates making a cross-reference listing.
    - **Cross reference listing** contains attributes like
      - name, type, dimension/size, etc
      - And the source line number at which a variable is declared (If explicitly declared) or first referenced (if implicitly declared)
      - and the source line numbers of all other references to the variable.

Summary so far (cont.)

- If insertion activity is high, **a tree-structured table** can provide a better performance overall-mainly because insertions can be handled with ease.
  - **Tree-structured tables** are particularly good if certain properties of a language are present
  - (e.g., six or fewer character names as in FORTRAN).
- The best method to use is **hashing**. (if memory is not an issue)
  - An average length of search of between one and three accesses is relatively easy to achieve.
  - The main disadvantage is that since the records are not ordered either physically or logically by variable name, such an organization is not helpful in building a cross-reference listing.

Common practice for compiler design

- It is common practice, when designing a compiler, to include a parameter which allows the user to estimate the size of the symbol-table requirements for a program.

- A possible strategy is to have the compiler select an appropriate symbol-table organization depending upon the table size requested.

Scoped Symbol Table

- **Scope** of a symbol defines the region of the program in which a particular definition of the symbol is valid i.e. the definition is visible.
- **Block structured languages** permit different types of scopes for the identifiers i.e. the scope rules for the language
  - **Global Scope**: visible throughout the program
  - **File-wide scope**: visible only with file (when program is distributed over multiple files)
  - **Local scope within a procedure**: visible only inside the procedure
  - **Local scope within a block**: visible only within the block

Scoping Rules

- **Scoping rules** are divided into two categories depending on the time at which the scope gets defined:
  - **Static or Lexical Scoping**
    - Scope is defined by syntactic nesting
    - This can be used efficiently by the compiler to generate correct references
  - **Dynamic or Runtime Scoping**
    - Scoping depends on the execution sequence of the program
    - Extra code is needed to decide which definition to use at runtime

Nested Lexical Scoping

```
Procedure P1
    int x
    ...
    Procedure P2
    ...
    end procedure
    Procedure P3
        int x
        x =
        ...
```

- To reach the definition of a symbol, apart from the current blocks that contain this innermost one, also have to be considered
- Current scope is the innermost scope
- There can be a number of open scope
- **Open scopes** include
  - One corresponding to the current scope
  - Others corresponding to each of the blocks surrounding it.

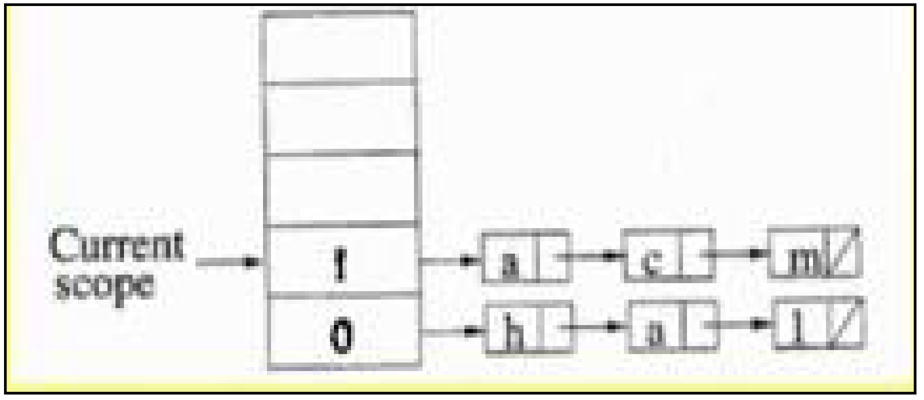
Nested Lexical Scoping

- So, hierarchically from one nesting level to the previous nesting level till it comes to the outermost level, the definition of a variable is checked/searched.
- **Visibility rules** are used to resolve conflicts arising due to the same variable being defined more than once.
- In this case, the innermost declaration closest to the reference is used.
- To implement the symbol tables with nested scope:
  1. **One table for each scope**
  2. **A single global table**

1. One Table per scope

- Maintain a different table for each scope
- A stack is used to remember the scopes of the symbol tables
- Here, **Lists, Trees, Hash tables** can be used.

Scoped Symbol Table: List



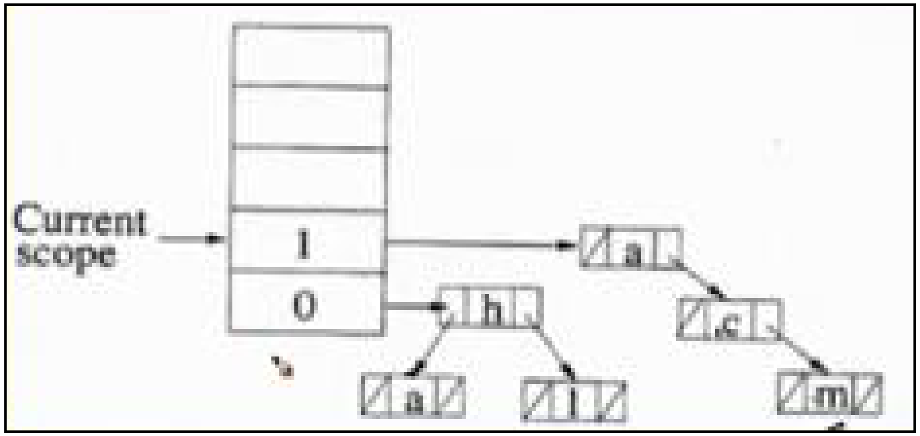
Current scope pointer points to current scope

- Nesting levels

```
{
    int h, a, l;
    {
        int a, c, m;
        ...
    }
    ...
}
```

*ImageRef: NPTEL course on Compiler Design by PROF. SANTANU CHATTOPADHYAY

Scoped Symbol Table: Tree



Current scope pointer points to current scope

- Nesting levels

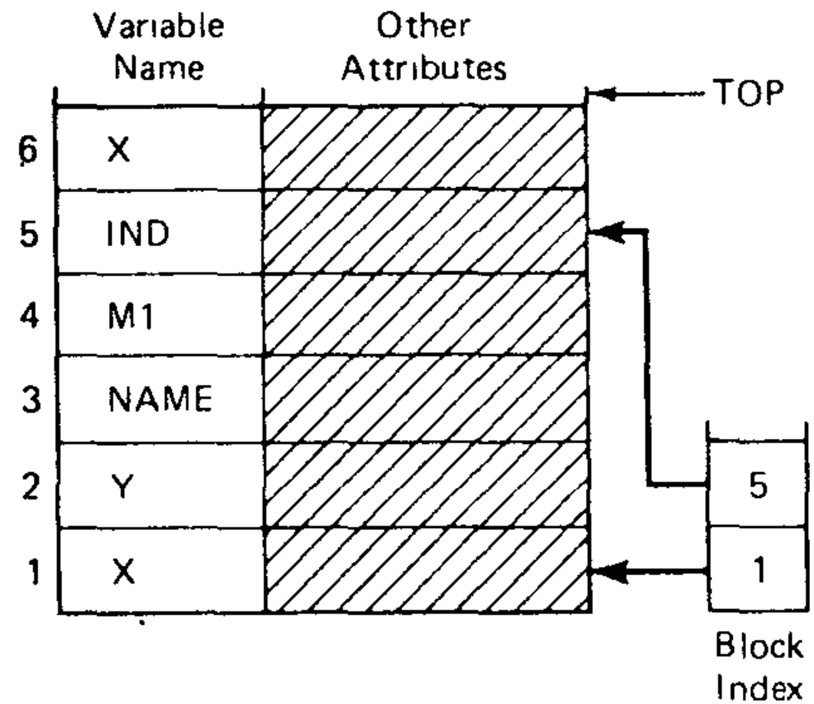
```
{
    int h, a, l;
    {
        int a, c, m;
        …
    }
    …
}
```

*ImageRef: NPTEL course on Compiler Design by PROF. SANTANU CHATTOPADHYAY

```
1  BBLOCK;
        REAL X, Y, STRING NAME,
        .
        .
        .
   2  M1.   PBLOCK (INTEGER IND),
              INTEGER X,
              .
              .
              .
              CALL M2(IND + 1),
              .
              .
              .
        END M1,

   3  M2.   PBLOCK (INTEGER J);
              .
              .
              .
        4  BBLOCK,
                ARRAY INTEGER F(J); LOGICAL TEST1;
              .
              .
              .
              END,
        END M2,
        .
        .
        .
        CALL M1 (X / Y),
        .
        .
        .
   END,
```

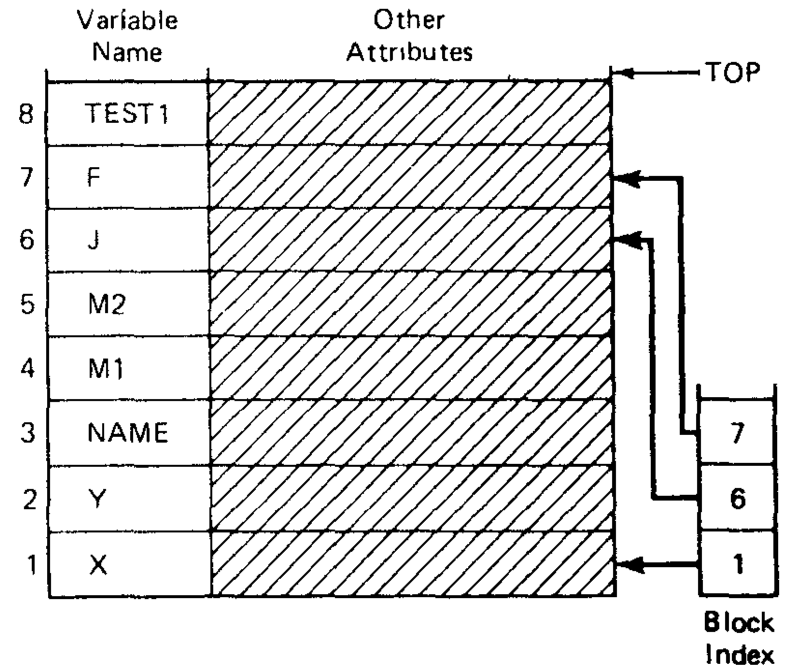
- Stack Symbol table just prior to completing the compilation of block 2



```
1  BBLOCK;
        REAL X, Y, STRING NAME,
        •
        •
        •
   2  M1.   PBLOCK (INTEGER IND),
              INTEGER X,
              •
              •
              •
              CALL M2(IND + 1),
              •
              •
              •
        END M1,

   3  M2.   PBLOCK (INTEGER J);
              •
              •
              •
        4  BBLOCK,
                 ARRAY INTEGER F(J); LOGICAL TEST1;
              •
              •
              •
              END,
        END M2,
        •
        •
        •
        CALL M1 (X / Y),
        •
        •
        •
   END,
```

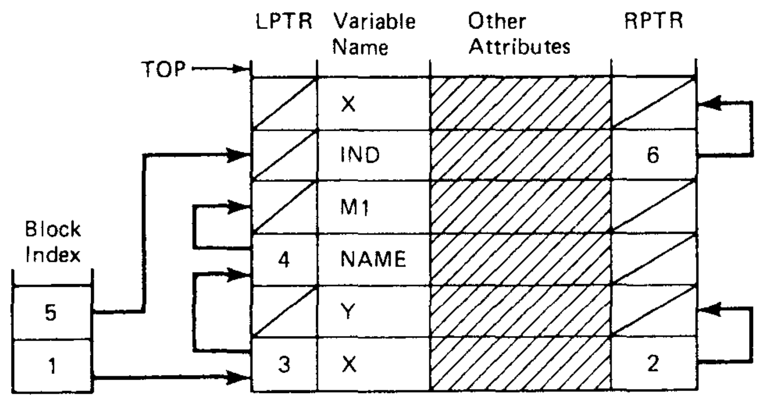
- Stack Symbol table just prior to completing the compilation of block 4



```
1  BBLOCK;
        REAL X, Y, STRING NAME,
        •
        •
        •
   2  M1.   PBLOCK (INTEGER IND),
              INTEGER X,
              •
              •
              •
              CALL M2(IND + 1),
              •
              •
              •
        END M1,

   3  M2.   PBLOCK (INTEGER J);
              •
              •
              •
        4   BBLOCK,
                  ARRAY INTEGER F(J); LOGICAL TEST1;
              •
              •
              •
              END,
        END M2,
        •
        •
        •
        CALL M1 (X / Y),
        •
        •
        •
   END,
```

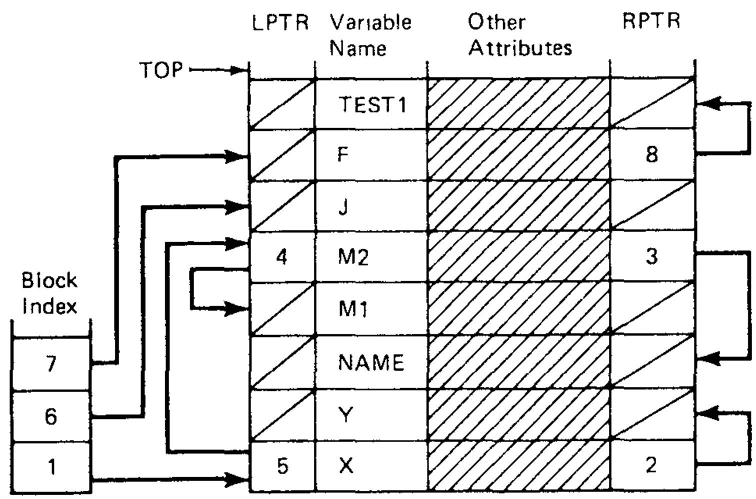
- Stack-implemented Tree-structured Symbol table just prior to completing the compilation of block 2



```
1  BBLOCK;
        REAL X, Y, STRING NAME,
        .
        .
        .
   2  M1.   PBLOCK (INTEGER IND),
            INTEGER X,
            .
            .
            .
            CALL M2(IND + 1),
            .
            .
            .
        END M1,

   3  M2.   PBLOCK (INTEGER J);
            .
            .
            .
      4     BBLOCK,
                ARRAY INTEGER F(J); LOGICAL TEST1;
            .
            .
            .
            END,
        END M2,
        .
        .
        .
        CALL M1 (X / Y),
        .
        .
        .
   END,
```

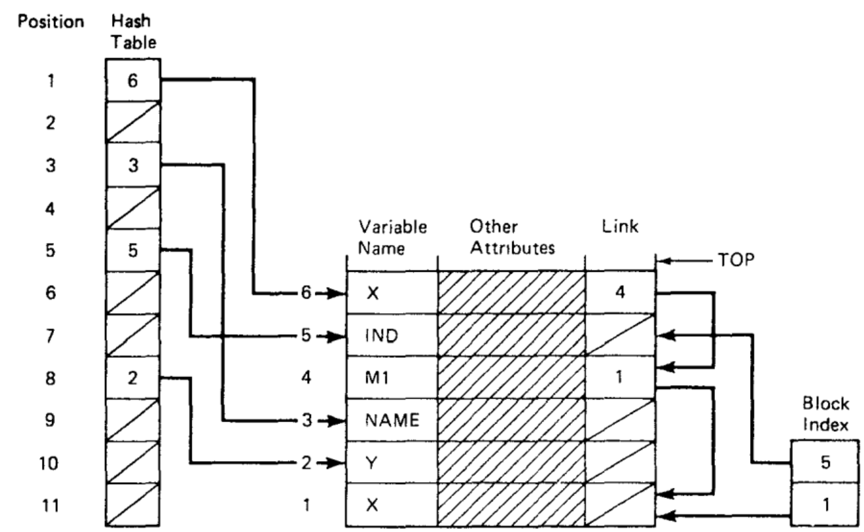
- Stack-implemented Tree-structured Symbol table just prior to completing the compilation of block 4



```
1   BBLOCK;
          REAL X, Y, STRING NAME,
          •
          •
          •
   2  M1.    PBLOCK (INTEGER IND),
             INTEGER X,
             •
             •
             •
             CALL M2(IND + 1),
             •
             •
             •
      END M1,

   3   M2.   PBLOCK (INTEGER J);
             •
             •
             •
       4   BBLOCK,
                 ARRAY INTEGER F(J); LOGICAL TEST1;
             •
             •
             •
           END,
      END M2,
      •
      •
      •
      CALL M1 (X / Y),
      •
      •
      •
   END,
```

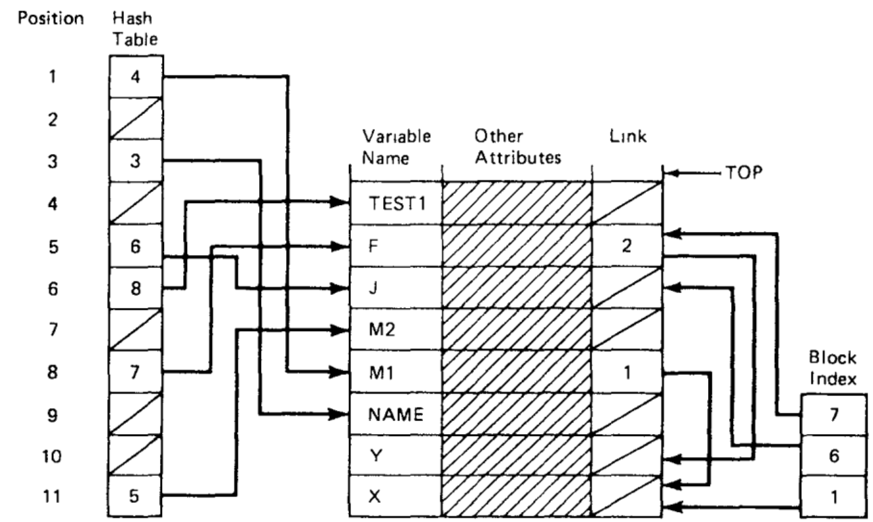
- Stack-implemented Hash Symbol table just prior to completing the compilation of block 2



```
1   BBLOCK;
          REAL X, Y, STRING NAME,
          •
          •
          •
  2   M1.   PBLOCK (INTEGER IND),
            INTEGER X,
            •
            •
            •
            CALL M2(IND + 1),
            •
            •
            •
      END M1,

  3   M2.   PBLOCK (INTEGER J);
            •
            •
            •
    4       BBLOCK,
                 ARRAY INTEGER F(J); LOGICAL TEST1;
            •
            •
            •
            END,
      END M2,
      •
      •
      •
      CALL M1 (X / Y),
      •
      •
      •
    END,
```

- Stack-implemented Hash Symbol table just prior to completing the compilation of block 4



Limitations of One Table per scope

- For a single-pass compiler, the table can be popped out and destroyed when the scope is closed but same is not true for multi-pass compiler.
  - Closing of scope means deleting all the variables declared in that scope when that scope ends.

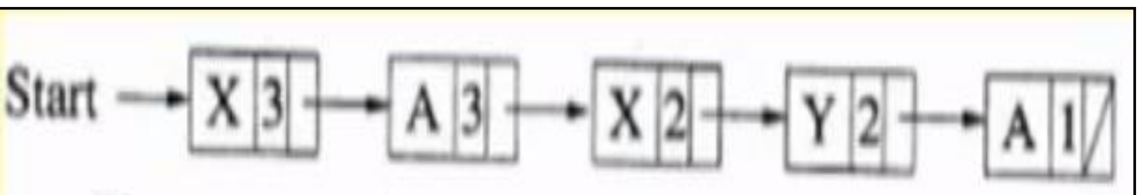
- Search may be expensive if the variable is defined much above in the hierarchy.

- What about the table size allotted to each block??
  - Underutilized or insufficient table size if estimation is not proper

One Table for All Scopes

- All identifiers are stored in a single table.
- Each entry in the symbol table has an extra field for identifying the scope.
- To search for an identifier, start with the highest scope number, then try the entries with next lesser scope numbers and so on.
- When a scope gets closed, all the identifiers with that scope number are removed from the table.
- More suitable for single pass compilers.
- Here also table can be represented as list, tree or hash table.

One Table for All Scopes -List



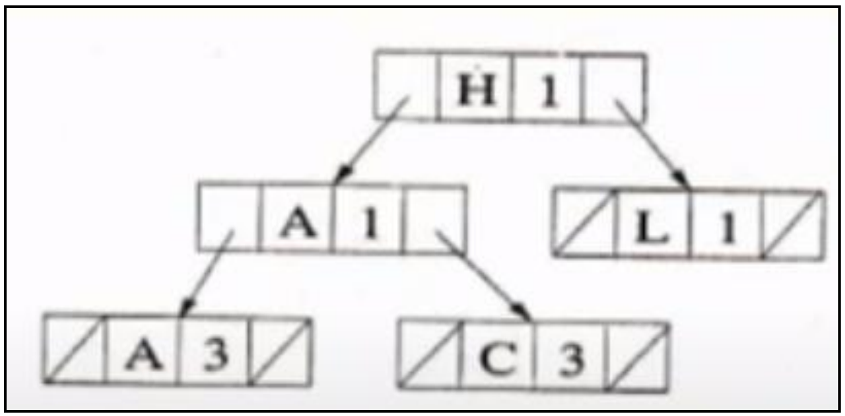
- Nesting Levels

```
{
    int A;
    {
        int X, Y;
        {
            int X, A;
            A = …
        }
    }
}
```

Start searching from Start and find the first occurrence

*ImageRef: NPTEL course on Compiler Design by PROF. SANTANU CHATTOPADHYAY

One Table for All Scopes -Tree

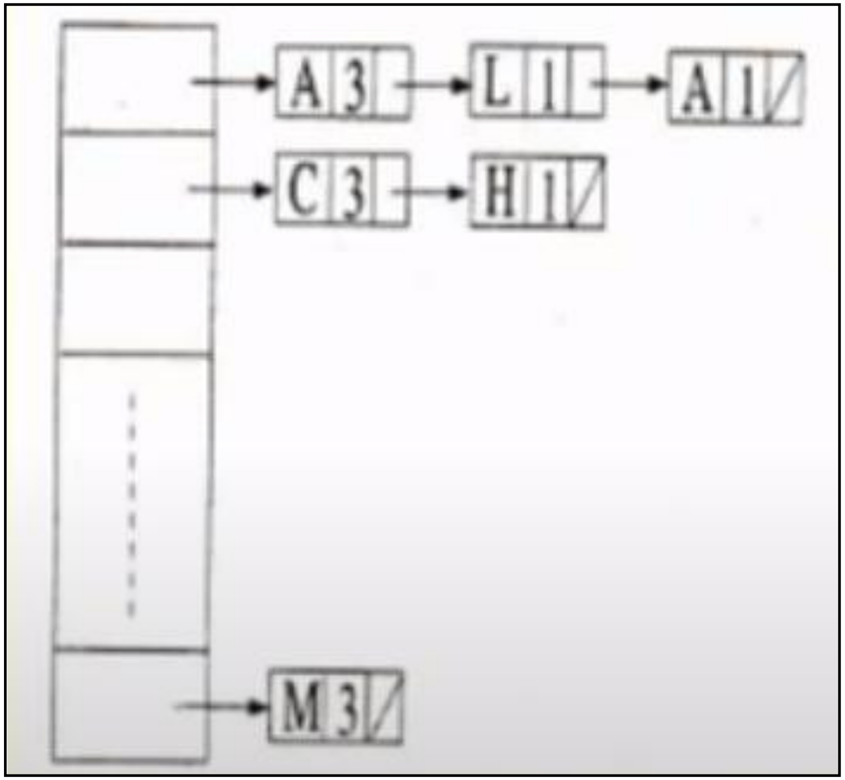


- Nesting Levels

```
{
    int H, A, L;
    {
        {
                int A, C;
        }
    }
}
```

*ImageRef: NPTEL course on Compiler Design by PROF. SANTANU CHATTOPADHYAY

One Table for All Scopes – Hash



- Nesting Levels

```
{
    int L, A, H;
    {
        {
                int A, C, M;
        }
    }
}
```

*ImageRef: NPTEL course on Compiler Design by PROF. SANTANU CHATTOPADHYAY

Advantage of using a Single Table

- Multiple tables need not be maintained.
- All information can be found from one table.
- But it totally depends on the compiler designer to choose between one table for all scope or one table for each scope.