Data Structures for Symbol Table in Compiler Design

1. Linear List

A **Linear List** is a simple data structure where elements (such as identifiers or symbols) are stored sequentially, typically in an array or linked list.

Characteristics:

- Sequential search is used to locate symbols.
- Insertion is easy but search time is **O(n)** in the worst case.
- Best suited for small symbol tables.

Advantages:

- Simple to implement.
- No overhead of complex data structures.

Disadvantages:

- Inefficient for large symbol tables due to linear search time.

2. Search Tree

A **Search Tree** (commonly a Binary Search Tree or BST) organizes symbols in a hierarchical manner based on a comparison function (usually alphabetical order).

Characteristics:

- Average time complexity for search, insert, and delete is **O(log n)** (if balanced).
- Nodes represent symbols with pointers to left/right subtrees.

Advantages:

- Faster lookups compared to a linear list.
- Maintains order among symbols.

Disadvantages:

- If not balanced, performance can degrade to $O(n)$.
- Slightly more complex to implement.

Variants:

- AVL Tree
- Red-Black Tree

3. Hash Table

A **Hash Table** is a data structure that stores key-value pairs using a **hash function** to compute an index into an array of buckets or slots.

Characteristics:

- Expected $O(1)$ time for search, insert, and delete.
- Collisions are handled using techniques like:
  - Chaining
  - Open addressing (e.g., linear probing, quadratic probing)

Advantages:

- Very fast access time for large symbol tables.
- Efficient memory usage with proper hash function and load factor.

Disadvantages:

- Performance depends on the quality of the hash function.
- Collisions can degrade performance.

Summary Table

Data Structure	Avg. Search Time	Complexity	Suitable For
Linear List	O(n)	Simple	Small tables
Search Tree	O(log n)	Moderate	Medium tables
Hash Table	O(1)	Complex	Large tables

Hash Functions in Symbol Table

Mid-Square Method (Hash Function)

Definition

The **Mid-Square Method** is a hashing technique in which the key is **squared**, and then a **portion from the middle digits** of the result is extracted as the hash value.

This method works well when the keys are **numeric** and relatively uniformly distributed.

Steps Involved

1. **Square the key (`k`)**.
2. **Extract the middle r digits** from the squared number.
3. **Apply modulo** with the table size (if needed) to keep the value within bounds.

Example

Let's take a key `k = 123`:

1. **Square the key**: 123^2 = 15129
2. **Extract middle digits**: If we want 2 middle digits (assuming 5-digit number), we take `51`.
3. **Hash value**: h(k) = 51
   If the table size is, say, 100:h(k) = 51 mod 100 = 51

Characteristics

- Works best when key values are not clustered.
- Better than simple modulo methods for certain data distributions.
- Suitable for numeric keys.

Advantages

- Simple to implement.
- Tends to give better distribution than division for small keys.
- Reduces the effect of patterns in the keys.

Disadvantages

- Choice of "middle digits" can affect performance.
- Squaring can be costly for very large keys.
- Less effective for alphanumeric or long string keys.

folding method

Definition

The **Folding Method** is a hashing technique where the key is divided into parts, and those parts are combined (usually by **addition** or **XOR**) to compute the hash value.

This method is particularly useful when keys are long numbers or strings.

Steps Involved

1. **Divide the key** into equal-sized parts (usually groups of digits or characters).
2. **Combine** the parts using:
   - Addition
   - XOR (bitwise)
   - Any consistent mathematical operation
3. **Apply modulo** with the hash table size (if needed).

Example (Using Addition)

Let the key be: `12345678`
Assume we split it into parts of 4 digits: `1234` and `5678`

1. Add the parts: $1234 + 5678 = 6912$
2. If table size `m = 100`, then: $h(k) = 6912 \bmod 100 = 12$

Folding by Boundary Reverse (Variant)

If parts are folded by reversing alternate parts:

Key: `12345678`
Split: `1234`, `5678` → reverse `5678` → `8765`
Now,
$1234 + 8765 = 9999$
$h(k) = 9999 \bmod 100 = 99$

Advantages

- Easy to implement.
- Works well for both numeric and character-based keys.
- Reduces the impact of repeated patterns in data.

Disadvantages

- Choice of partitioning can affect performance.
- May still lead to collisions if parts are similar or repetitive.

Best Use Case

- When keys are **long identifiers**, like memory addresses or long numeric strings.
- When keys contain repeating patterns.

Division Method (Hash Function)

Definition

The **Division Method** is one of the simplest and most commonly used hashing techniques.
It computes the hash value by taking the **remainder** of the division of the key by the table size.

Formula $h(k) = k \bmod m$

Where:

- `h(k)` is the hash value,
- `k` is the key (usually a numeric representation of the symbol),
- `m` is the size of the hash table (preferably a **prime number** to reduce collisions).

Example

Let the key `k = 1234`, and the table size `m = 13`.

Then: $h(1234) = 1234 \bmod 13 = 12$
So, the key `1234` will be placed in slot `12` of the hash table.

Choosing `m` (Table Size)

- Should be a **prime number** not too close to a power of 2.
- Helps in spreading the keys more uniformly.
- Avoid values of `m` that are multiples of common patterns in keys (e.g., 10, 100, etc.).

Advantages

- Very simple and fast to compute.
- Efficient for integer keys.
- Easy to implement in both hardware and software.

Disadvantages

- Performance highly depends on a **good choice of** `m`.
- Poor distribution if `m` is not chosen wisely (e.g., if `m` is even or not prime).
- Vulnerable to **clustering** if keys share a common factor with `m`.

Storage allocation

1. Static Allocation

Definition

Static Allocation is a storage allocation method where **memory for all variables is allocated at compile time**. The addresses of all variables are known **before the program is executed**.

Characteristics

- Each variable is assigned a fixed memory location during compilation.
- No allocation or deallocation during runtime.
- Suitable for programs with **no recursion** and **fixed-size data**.

When is it Used?

- In **simple languages** or early-stage compilers.
- For **global variables** or constants.
- In embedded systems with limited memory.

Advantages

- Simple to implement.
- No runtime overhead for memory allocation or deallocation.
- Fast access due to fixed memory addresses.

Disadvantages

- **Inefficient memory usage**: memory is reserved even if variables are not used.
- **No support for recursion**: each recursive call needs a separate instance of variables.
- **Lack of flexibility**: cannot handle dynamic data structures like linked lists, trees, etc.

Example

```
int x;     // memory for x is allocated at compile time
float y;   // memory for y is allocated at compile time
```

Dynamic Allocation

Definition

Dynamic Allocation is a storage allocation method where memory is allocated **at runtime**, typically from the **heap**. It is used for data structures whose size or lifetime cannot be determined at compile time.

Characteristics

- Memory is allocated and deallocated **manually** by the programmer or **automatically** via the runtime environment.
- Enables creation of **dynamic data structures** such as:
  - Linked lists
  - Trees
  - Graphs
- Allocation is done using system/library functions like `malloc()`, `calloc()` in C/C++, or `new` in Java.

When is it Used?

- When data sizes are unknown until runtime.
- In applications requiring **dynamic memory management**, such as interpreters or systems with unpredictable input size.
- For objects and structures with variable lifetime.

Advantages

- **Efficient memory usage**: memory is allocated only when needed.
- **Supports complex and flexible data structures**.
- **Recursion and variable-sized data** are handled easily.

Disadvantages

- More complex implementation.
- May lead to memory leaks if deallocation is not done properly.
- Slower access time compared to static or stack allocation.
- Requires garbage collection or manual memory management.

Example in C

```
int* ptr = (int*) malloc(sizeof(int));  // dynamically allocates memory for an
integer
*ptr = 10;
free(ptr);  // deallocates the memory
```

In the symbol table:

- Only the type information may be stored initially.
- Actual memory location is determined and stored at runtime.

Hybrid Allocation

Definition

Hybrid Allocation is a combination of two or more memory allocation strategies — static, stack, and dynamic (heap) — to leverage the advantages of each and reduce their limitations.

It is the most commonly used approach in modern compilers and runtime systems.

Characteristics

- Static allocation is used for global/static variables and constants.
- Stack allocation is used for local variables with predictable lifetimes (e.g., function calls).
- Dynamic allocation (heap) is used for variable-size or user-defined data structures with unpredictable lifetimes.

Why Use Hybrid Allocation?

- To provide **efficiency**, **flexibility**, and **recursion support**.
- To optimize memory usage based on the **nature of data** and **lifetime of variables**.

Memory Segments in a Hybrid System

1. **Code Segment** – Stores program instructions (read-only).
2. **Data Segment (Static)** – Stores global/static variables.
3. **Stack Segment** – Stores function call information and local variables.
4. **Heap Segment** – Stores dynamically allocated memory.

Example (C/C++)

```
int x = 5;                 // static/global segment
void func() {
    int y = 10;            // stack segment
    int* ptr = malloc(4); // heap segment
    free(ptr);
}
```

Advantages

- Combines **speed** of static and stack allocation with the **flexibility** of dynamic allocation.
- Efficient use of memory based on variable type and lifetime.
- Supports recursion, dynamic structures, and predictable memory.

Disadvantages

- More **complex to implement and manage**.
- Requires careful coordination between compiler and runtime system.
- Possibility of **memory fragmentation** in the heap.