Symbol Tables

- A **symbol table** is a major data structure used in a compiler:
  - Associates **attributes** with identifiers used in a program
  - For instance, a **type attribute** is usually associated with each identifier
  - A symbol table is a necessary component because:
    - Definition (declaration) of identifiers appears once in a program
    - Use of identifiers may appear in many places of the program text
  - Identifiers and attributes are entered by the analysis phases
    - When processing a definition (declaration) of an identifier
    - In simple languages with only global variables and implicit declarations:
      - The scanner can enter an identifier into a symbol table if it is not already there
    - In block-structured languages with scopes and explicit declarations:
      - The parser and/or semantic analyzer enter identifiers and corresponding attributes
  - Symbol table information is used by the analysis and synthesis phases
    - To verify that used identifiers have been defined (declared)
    - To verify that expressions and assignments are semantically correct – **type checking**
    - To generate intermediate or target code

Symbol Table Interface

- The basic operations defined on a symbol table include:
  - **allocate** – to allocate a new empty symbol table
  - **free** – to remove all entries and free the storage of a symbol table
  - **insert** – to insert a name in a symbol table and return a pointer to its entry
  - **lookup** – to search for a name and return a pointer to its entry
  - **set_attribute** – to associate an attribute with a given entry
  - **get_attribute** – to get an attribute associated with a given entry
- Other operations can be added depending on requirement
  - For example, a **delete** operation removes a name previously inserted
    - Some identifiers become invisible (out of scope) after exiting a block
- This interface provides an abstract view of a symbol table
- Supports the simultaneous existence of multiple tables
- Implementation can vary without modifying the interface

Basic Implementation Techniques

- The first consideration is how to **insert** and **lookup** names
- Variety of implementation techniques exist depending on performance desired
- **Unordered List**
  - Simplest to implement
  - Implemented as an array or a linked list
  - Linked list can grow dynamically – alleviates problems of a fixed size array
  - Insertion is fast $O(1)$, but lookup is very slow for large tables – $O(n)$ on average
- **Ordered List**
  - If an array is sorted, it can be searched using binary search – $O(\log_2 n)$
  - New entry must be inserted at appropriate location – Expensive $O(n)$ on average
  - Useful when entire set of names is known in advance – table of reserved words
- **Binary Search Tree**
  - Can grow dynamically
  - Insertion and lookup are $O(\log_2 n)$ on average
- **Hash Table**
  - Probably the most commonly used data structure to implement symbol tables
  - Insertion and lookup can be very fast – $O(1)$

Hash Tables and Hash Functions

- A **hash table** is an array indexed by an integer range 0 to *TableSize* – 1
- A **hash function** maps an identifier name into a table index
  - A hash function, $h(name)$, should depend solely on *name*
  - $h(name)$ should be computed quickly
  - *h* should be **uniform** and **randomizing** in distributing names over the index range
  - All table indices should be mapped with equal probability
  - Similar names should not cluster to the same table index
- Hash functions can be defined in many ways:
  - A string can be treated as a sequence of integer words
    - Several characters are fit into an integer word
    - Strings longer than one integer word are folded into one word using exclusive-or
    - Hash value is obtained by taking integer word modulo *TableSize*
  - We can also compute a hash value character by character:
    - $h(name) = (c_0 + c_1 + \ldots + c_{n-1})$ **mod** *TableSize*, where *n* is the length of *name*
    - $h(name) = (c_0 * c_1 * \ldots * c_{n-1})$ **mod** *TableSize*
    - $h(name) = (c_{n-1} + \alpha\, (c_{n-2} + \ldots + \alpha\, (c_1 + \alpha\, c_0)))$ **mod** *TableSize*
    - $h(name) = (c_0 * c_{n-1} * n)$ **mod** *TableSize*

Resolving Collisions – Open Addressing

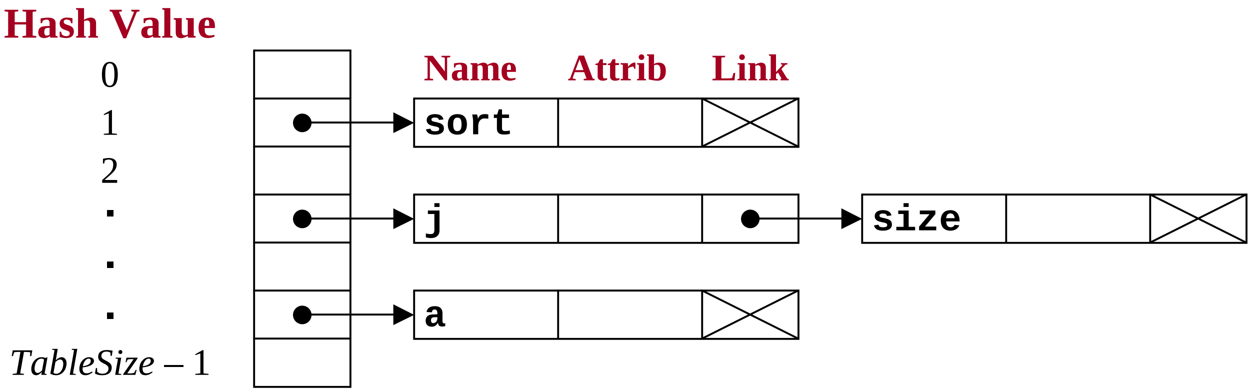
- A **collision** occurs when $h(name_1) = h(name_2)$ and $name_1 \neq name_2$
- Collisions are inevitable because
  - The name space of identifiers is much larger than the table size
- How to deal with collisions?
  - If entry $h(name)$ is occupied, try $h_2(name)$, $h_3(name)$, etc.
  - This approach is called **open addressing**
  - $h_2(name)$ can be $h(name) + 1$ **mod** *TableSize*
  - $h_3(name)$ can be $h(name) + 2$ **mod** *TableSize*

linear probing

Hash Value	**Name**	**Attributes**
0		
1		
2	`i`	
.	`size`	
.	`temp`	
.	`j`	
TableSize – 1		

Resolving Collisions – Chaining by Separate Lists

- Drawbacks of open addressing:
  - As the array fills, collisions become more frequent – reduced performance
  - Table size is an issue – dynamically increasing the table size is a difficulty
  - We can however allocate multiple tables and link them together
- An alternative to open addressing is **chaining by separate lists**
  - The hash table is an array of pointers to linked lists called **buckets**
  - Collisions are resolved by inserting a new identifier into a linked list
  - Number of identifiers is no longer restricted to table size
  - If number of identifiers greatly exceed *TableSize* then lookup is *O*(*n*/*TableSize*)
  - It is also possible to organize each chain as a binary search tree



Hash Value

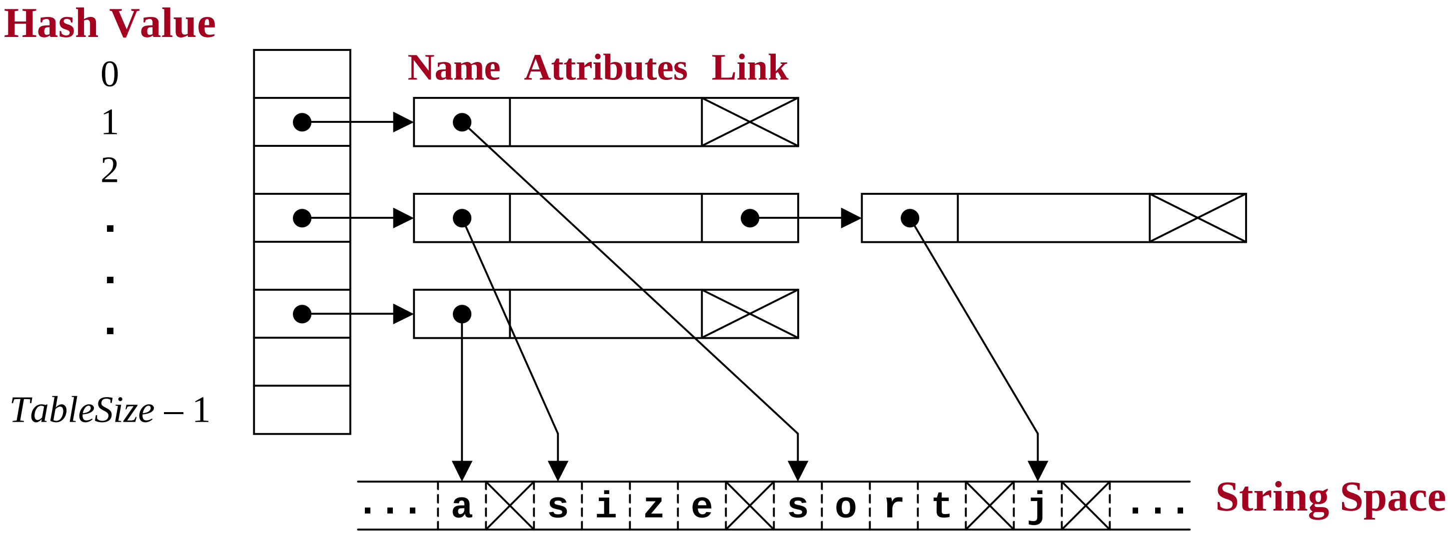
0
1
2
.
.
.
TableSize – 1

Name **Attrib** **Link**

`sort`
`j`
`size`
`a`

String Space Array

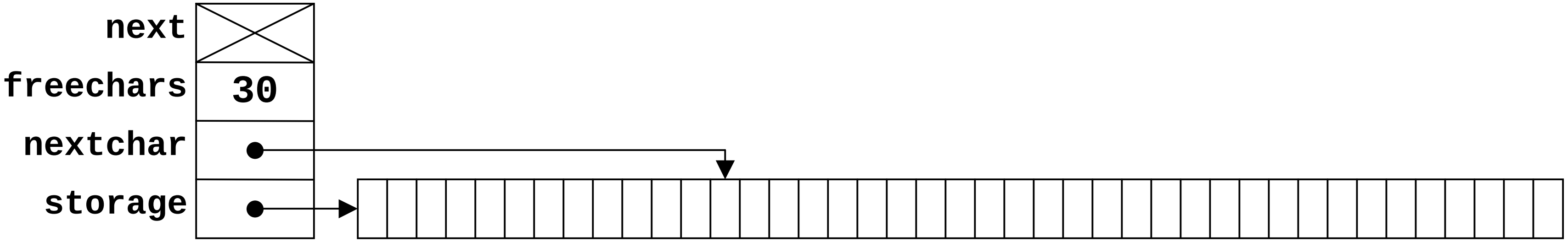
- The length of names entered into a symbol table may vary greatly
  - Entering a name directly into a symbol entry leads to considerable inefficiencies
  - Enough space in a name field has to accommodate the longest possible name
  - To reduce storage waste, we use a character array to store all names
  - The character array, called a **string space**, can be allocated and deleted in one step
  - Choosing the size of a string space array is a difficult problem
    - A small array cannot accommodate many names; a large array wastes space
  - To allow growth, array segments should be allocated and linked dynamically



String Segment Class Definition

- A string space can be implemented as a linked list of string segments
- Each string segment is an array of characters allocated dynamically
- We keep track of insertion location and the number of free characters

```
class StrSeg {                       // String Segment Class
friend class StrSpc;                 // StrSpc can access private members
public:
  StrSeg(int size);                  // Allocate storage of given size
  ~StrSeg();                         // Free storage of this segment
  char *insert(char *s,int l);  // Insert s of length l
private:
  StrSeg *next;                      // Next segment in string space
  int    freechars;                  // Free characters in storage[]
  char   *nextchar;                  // Next free position in storage[]
  char   *storage;                   // Storage array of this segment
};
```



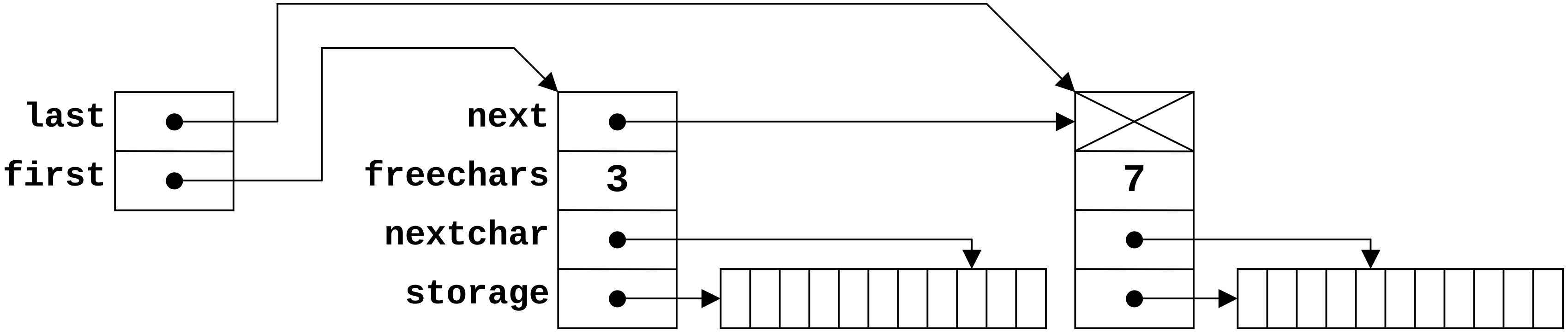
String Segment Class Implementation

```
StrSeg::StrSeg(int size) {         // Constructor of this segment
  storage = new char[size];        // Dynamic array of size chars
  freechars = size;                // All characters are initially free
  nextchar = storage;              // insertion point starts at storage
  next = 0;                        // No next segment at this time
}

StrSeg::~StrSeg() {                // Destructor of this segment
  delete [] storage;               // Free storage of this segment
  if (next) delete next;           // Free next segment if any
}

char *StrSeg::insert(char *str, int len) {
  if (freechars<=len) return 0;    // No storage available
  char *strptr = nextchar;         // Insertion address of str
  for (int i=0; i<len; i++)        // Copy str character by character
    strptr[i] = str[i];
  strptr[len] = '\0';              // mark end of string
  nextchar += (len+1);             // Update nextchar
  freechars -= (len+1);            // Update freechars
  return strptr;                   // Pointer to inserted string
}
```

String Space Class Definition

- A string space is implemented as a linked list of string segments
- A new string segment is linked at the end of linked list and pointed by *last*
- This queue arrangement will preserve the order of inserted strings

```
class StrSpc {                          // String Space Class
public:
  StrSpc(){first=0; last=0;}            // Empty string space
  ~StrSpc(){if(first)delete first;}     // Delete all string segments
  char *insert(char *s, int len);       // insert s of length len
private:
  StrSeg *last;                         // Pointer to last segment
  StrSeg *first;                        // Pointer to first segment
};
```



String Space Class Implementation

```
const int DefSegSize = 500;          // Default size of segment array

char *StrSpc::
insert(char *str, int len) {         // Insert str of length len
  if (len<0) return 0;               // Insertion is not possible

  int segsize = DefSegSize;          // Compute segment array size
  if (len >= segsize)                // Long strings are handled
    segsize = len+1;

  if (last == 0) {                   // First segment in string space
    last = new StrSeg(segsize);      // Allocate new segment
    first = last;
  }

  char *strptr=last->insert(str,len);// First attempt to insert

  if (strptr == 0) {                 // First attempt not successful
    last->next=new StrSeg(segsize);  // Allocated new segment
    last=last->next;                 // Update last pointer
    strptr = last->insert(str,len);  // Second attempt to insert
  }

  return strptr;                     // Pointer to inserted string
}
```

Symbol and Symbol Table Class Definition

```
class Symbol {              // Symbol class definition
friend class SymTable;      // SymTable can access private members
public:
  Symbol() {next=0; id=0;}  // Construct empty symbol
  char *name() {return id;} // Return name of this symbol
private:
  Symbol *next;             // Symbols can be linked
  char *id;                 // Attributes can be added later
};

class SymTable {            // Symbol Table class definition
public:
  SymTable(int size);       // Allocate hash table of given size
  ~SymTable();              // Free table, symbols, and string space
  Symbol *insert(char *s);  // Insert s and return pointer to symbol
  Symbol *lookup(char *s);  // Lookup s and return pointer to symbol
                            // Other public methods are listed here
private:
  Symbol **htable;          // Hash table allocated dynamically
  StrSpc strspc;            // String space facility
};
```