Lists and Strings

A List ADT

- A **list** *L* of items of type *T* is a sequence of items of type *T* on which the following operations are defined:
  - **Initialize** the list *L* to be the empty list.
  - Determine whether or not the list *L* is **empty**.
  - Find the length of a list *L* (where the **length** of *L* is the number of items in *L* and the length of the empty list is 0).
  - **Select** the i-th item of a list *L*, where $1 \leq i \leq length(L)$.
  - **Replace** the i-th item *X* of a list *L* with a new item *Y* where $1 \leq i \leq length(L)$.
  - **Delete** any item *X* from a nonempty list *L*.
  - **Insert** a new item *X* into a list *L* in any arbitrary position (such as before the first item of *L*, after the last item of *L* or between any two items of *L*).

Lists

- Lists are more general kinds of containers than stacks and queues.
- Lists can be represented by **sequential representations** and **linked representations**.

Sequential List Representations

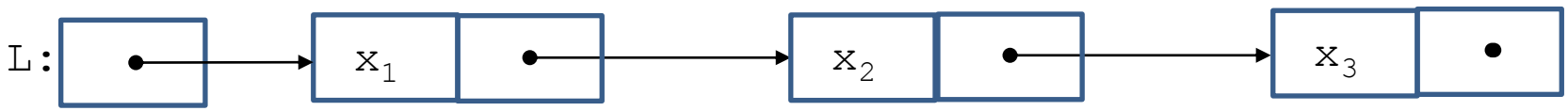
- We can use an array `A[0:MaxSize-1]` as we show graphically (items are stored **contiguously**):

Advantages and Disadvantages

- Advantages:
  - Fast access to the i-th item of the list in *O(1)* time.
- Disadvantages:
  - Insertions and deletions may require shifting all items i.e., an *O(n)* cost on the average.
  - The size of the array should be known in advance. So if we have small size, we run the risk of overflow and if we have large size, we will be wasting space.

One-Way Linked Lists Representation

- We can use chains of linked nodes as shown below:



Declaring Data Types for Linked Lists

The following statements declare appropriate data types for our linked lists from earlier lectures:

```
typedef char AirportCode[4];
typedef struct NodeTag {
                        AirportCode Airport;
                        struct NodeTag *Link;
                } NodeType;
typedef NodeType *NodePointer;
```

We can now define variables of these datatypes:

```
NodePointer L;
```

or equivalently

```
NodeType *L;
```

Accessing the i^{th} Item

```
void PrintItem(int i, NodeType *L)
{
     while ((i > 1) && (L != NULL)){
          L=L->Link;
          i--;
       }
       if ((i == 1) && (L != NULL)){
          printf("%s", L->Item);
       } else {
          printf("Error - attempt to print an
item that is not on the list.\n");
       }
}
```

Computational Complexity

- Suppose that list *L* has exactly n items. If it is equally likely that each of these items can be accessed, then the average number of n pointers followed to access the i^{th} item is:

$$Average = \frac{(1 + 2 + \cdots + n)}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n}{2} + \frac{1}{2}$$

- Therefore, the average time to access the i^{th} item is $O(n)$.
- The complexity bound is the same for inserting before or after the i^{th} item or deleting it or replacing it.

Comparing Sequential and Linked List Representations

List Operation	Sequential Representation	Linked List Representation
Finding the length of *L*	*O(1)*	*O(n)*
Inserting a new first item	*O(n)*	*O(1)*
Deleting the last item	*O(1)*	*O(n)*
Replacing the i^{th} item	*O(1)*	*O(n)*
Deleting the i^{th} item	*O(n)*	*O(n)*

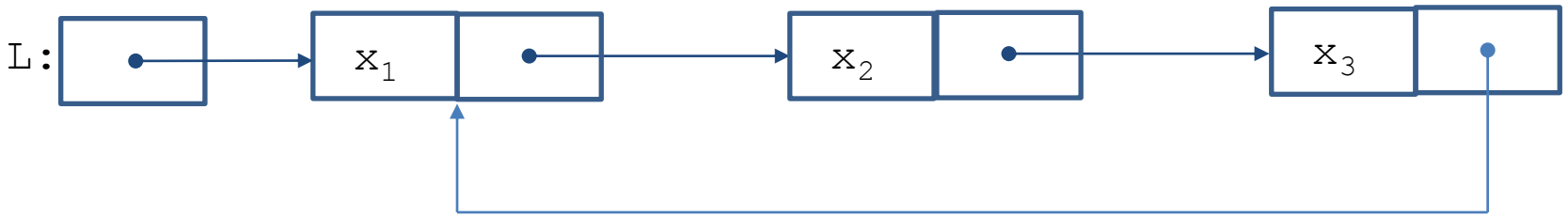
The above table gives **average running times**. But time is not the only resource that is of interest. **Space** can also be an important resource in some applications.

Other Linked List Representations

- Circular linked lists
- Two-way linked lists
- Linked lists with header nodes

Circular linked lists

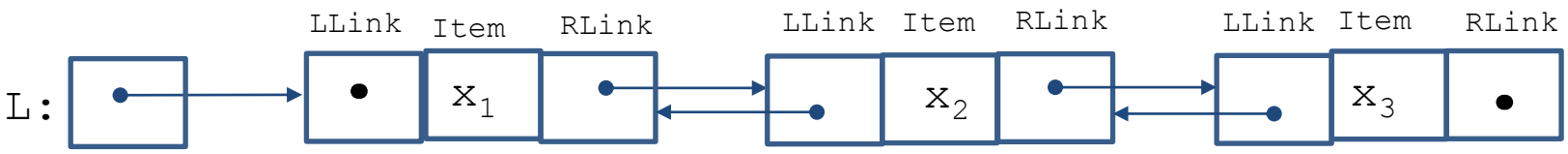
- A **circular linked list** is formed by having the link in the last node of a one-way linked list point back to the first node.



- The advantage of a circular linked list is that any node on it is accessible by any other node.

Two-Way Linked Lists

- Two-way linked lists are formed from nodes that have pointers to both their right and left neighbors on the list.

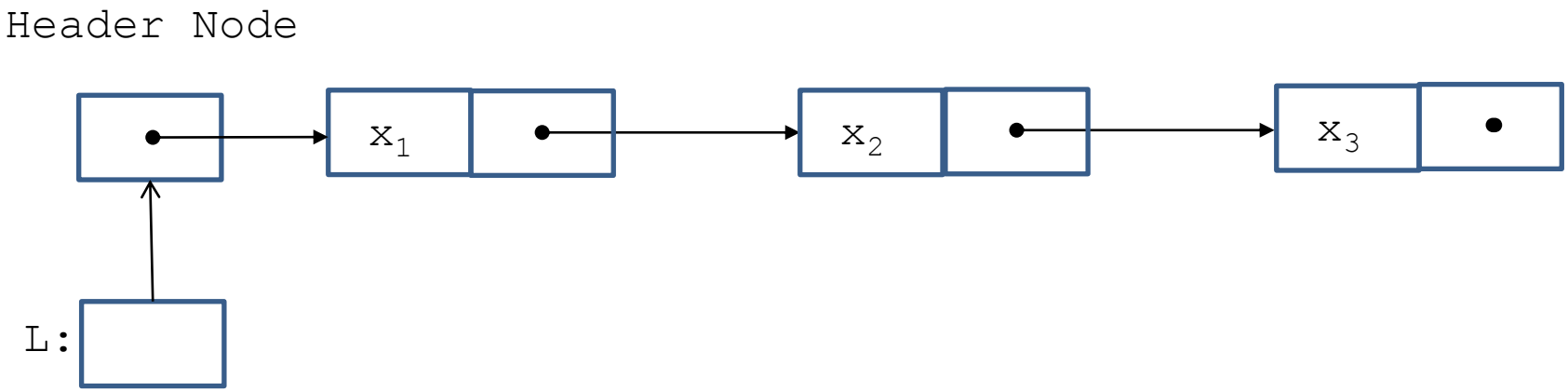


Two-Way Linked Lists (cont'd)

- Given a pointer to a node *N* in a two-way linked list, we can follow links in either direction to access other nodes.
- We can insert a node *M* either before or after *N* starting only with the information given by the pointer to *N*.

Linked Lists with Header Nodes

- Sometimes it is convenient to have a special **header node** that points to the first node in a linked list of item nodes.



Linked Lists with Header Nodes (cont'd)

- Header nodes can be used to hold information such as the number of nodes in the list etc.

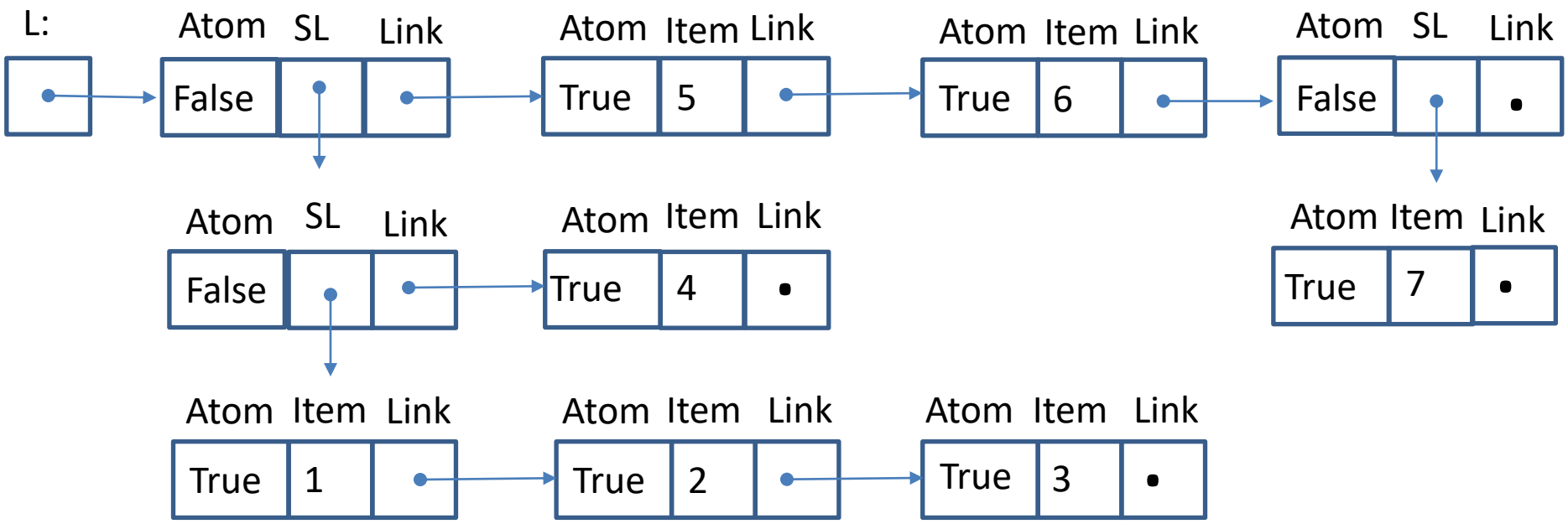
Generalized Lists

- A **generalized list** is a list in which the individual list items are permitted to be sublists.
- **Example**: $(a_1, a_2, (b_1, (c_1, c_2), b_3), a_4, (d_1, d_2), a_6)$
- If a list item is not a sublist, it is said to be **atomic**.

- Generalized lists can be represented by sequential or linked representations.

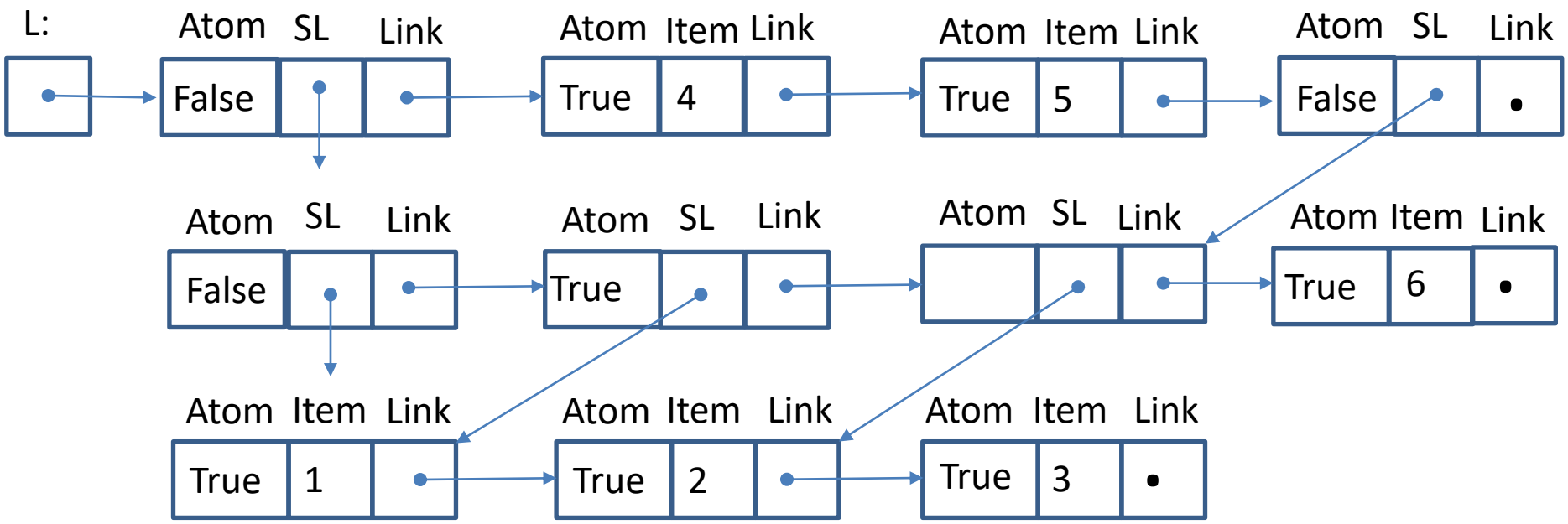
Generalized Lists (cont'd)

- The generalized list *L=(((1, 2, 3), 4), 5, 6, (7))* can be represented without shared sublists as follows:



Generalized Lists (cont'd)

- The generalized list *L=(((1, 2, 3), (1, 2, 3), (2, 3), 6), 4, 5, ((2, 3), 6)* can be represented with shared sublists as follows:



A Datatype for Generalized List Nodes

```
typedef struct GenListTag {
                GenListTag *Link;
                int Atom;
                union SubNodeTag {
                      ItemType Item;
                      struct GenListTag *Sublist;
                } SubNode;
        } GenListNode;
```

Printing Generalized Lists

```
void PrintList(GenListNode *L)
{
      GenListNode *G;

      printf("(");
      G=L;
      while (G != NULL){
             if (G->Atom){
                printf("%d", G->SubNode.Item);
             } else {
                printList(G->SubNode.SubList);
             }
             if (G->Link != NULL) printf(",");
             G=G->Link;
      }
      printf(")");
}
```

Applications of Generalized Lists

- Artificial Intelligence programming languages LISP and Prolog offer generalized lists as a language construct.

- Generalized lists are often used in Artificial Intelligence applications.

- More in the courses “Artificial Intelligence” and “Logic Programming”.

Strings

- **Strings** are sequences of characters. They have many applications:
  - Word processors
  - E-mail systems
  - Databases
  - …

Strings in C

- A **string** in C is a sequence of characters terminated by the null character "\0".
- **Example**: To represent a string S=="canine" in C, we allocate a block of memory B at least seven bytes long and place the characters "canine" in bytes B[0:5]. Then, in byte B[6], we place the character "\0".

A String ADT

- In C's **standard library** you can access a collection of useful string operations by including the header file `<string.h>` in your program.

- These functions define a **predefined string ADT.**

Examples of String Operations

- Let us assume that `S` and `T` are string variables (i.e., of type `char *`). Then:
  - `strlen(S)`: returns the number of characters in string `S` (not including the terminating character `'\0'`).
  - `strstr(S,T)`: returns a pointer to the first occurrence of string `S` in string `T` (or `NULL` if there is no occurrence of string `S` in string `T`).
  - `strcat(S,T)`: concatenate a copy of string `T` to the end of string `S` and return a pointer to the beginning of the enlarged string `S`.
  - `strcpy(S,T)`: make a copy of the string `T` including a terminating last character `'\0'`, and store it starting at the location pointed to by the character pointer `S`.

Concatenating Two Strings

```
char *Concat(char *S, char *T)
{
   char *P;
   char *temp;

   P=(char *)malloc(1+strlen(S)+strlen(T));
   temp=P;
   while ((*P++=*S++)!='\0')
   ;
   P--;
   while ((*P++=*T++)!='\0')
   ;
   return(temp);
}
```

Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.

  Chapter 8, Sections 8.1-8.5.
- Robert Sedgewick. Αλγόριθμοι σε C.

  Κεφ. 3.