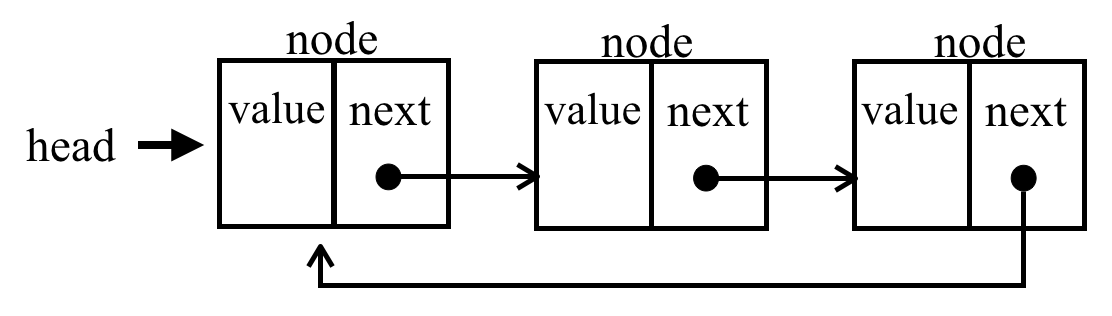
**Chapter 6 Questions**

A \_\_**generic data type**\_ is a type for which the operations are defined but the types of the items being manipulated are not.

Some languages have special language constructs that allow you to define generic data types. The construct is called a \_**template**\_\_\_\_\_\_\_\_\_.

A \_**template**\_ allows you to write a description of a class type with “blanks” left in the description to be filled in by the client code.



The image above is called a \_**circular link list**\_\_.

**6.3**

A task that is difficult on a linear linked list is traversing the list in \_**reverse**\_.

In a case where we need to access the node that precedes a given node, a \_**doubly linked list**\_ is useful.

In a doubly linked list, the \_**nodes\_\_** are linked in both directions.

Each node of a doubly linked list contains three parts: \_**info**\_, \_**next**\_ and \_\_**back**\_\_.

Each node of a doubly linked list contains three parts. The \_**info**\_ part holds the data stored in the node.

Each node of a doubly linked list contains three parts. The \_\_**next**\_ part is the pointer to the following node.

Each node of a doubly linked list contains three parts. The \_**back**\_ part is the pointer to the preceding node.



The image above is a \_**linear doubly linked list**\_. (note: 4 words)

The algorithms for the insertion and deletion operations on a doubly linked list are somewhat more complicated than the corresponding operations on a singly linked list. The reason is clear, there are more \_**pointers**\_ to keep track of in a doubly linked list.

Concerning the operations on a doubly linked list, we have to be careful about inserting an item into an \_**empty list**\_. It is a special case.

**6.4**

“In writing the insertion and deletion algorithms for all implementations of linked lists, we see that special cases arise when we are dealing with the first node or the last node. One way to simplify these algorithms is to ensure that we never insert or delete items at the ends of the list.”

In reference to the paragraph above, how can we be sure about the ends of the list? By using predefined keys as \_**identification**\_.

In reference to the paragraph above, if we can determine the range of possible values for the key, it is a simple matter to set up dummy \_**nodes**\_with values outside of the range of valid key values.

In reference to the paragraph above and the generation of dummy nodes, a \_**header**\_ node containing a value smaller than any other valid list element key can be placed at the start of the list.

In reference to the paragraph above and the generation of dummy nodes, a \_**trailer**\_ node containing a value larger than any other valid list element key can be placed at the end of the list.

**6.5**

A shallow copy works well with the regular = operator as long as our objects do not use \_\_\_\_\_\_\_\_\_\_. Pointers

If we are copying one object to another without pointers, our regular = operator works fine. The \_\_\_\_\_\_\_\_\_\_ will handle writing the code for us. compiler

If we are copying one object to another without pointers, our regular = operator works fine. This is called making a \_\_\_\_\_\_\_\_\_\_ copy. Shallow

A deep copy requires the programmer to write a copy constructor. This is needed because the objects we are copying is using \_\_\_\_\_\_\_\_\_\_. Pointers.

If we are copying one object to another with pointers, our regular = operator does not work properly. To make it work, we can \_\_\_\_\_\_\_\_\_\_ the = operator. Overload

If we are copying one object to another with pointers and an overloaded = operator, we are making a \_\_\_\_\_\_\_\_\_\_ copy. Deep

The assignment operator = normally causes \_**shallow**\_ copying.

The syntax for overloading a symbol is the word \_**operator**\_ followed by the symbol to be overloaded. (operator=)