Let \( \text{list} \) be a list of \( n \) names. Write a subroutine \( \text{insertblank(list,n,k)} \) which, given \( k \) in \([1,n]\) inserts a blank record "" into the position \( k \) and pushes the existing records from \( k \) to \( n-1 \) down to positions \( k+1 \) and \( n \). We assume the existing record at position \( n \) is blank.

Inserting a blank at position \( k=2 \) in the list on the left, should give the list on the right.

| list(1)='ann' | list(1)='ann' |
| list(2)='bob' | list(2)='bob' |
| list(3)='jack' | list(3)='jack' |
| ... | ... |
| list(8)='renee' | list(9)='renee' |
| list(9)='stacie' | list(10)'stacie' |
| list(10)='' | list(10)='' |

Just setting \( \text{list(2)}='' \) isn't enough, names from index \( k \) to 9 must first be shifted down to later positions.

Setting \( \text{list(k)}='' \) is the last step.

Write a program which asks the user for the index \( k \), then calls this subroutine and then prints the resulting list.

Hint: Get the program working first with \( k \) initialized to say 4. Then uncomment lines which ask the user for the value of \( k \).
Let \texttt{list} be a list of \texttt{n} names. Write a subroutine \texttt{deletek(list,n,k)} which, given \texttt{k} in \([1, n]\) deletes the \texttt{k}th element of \texttt{list} and then closes any gaps in the list so that only the last element is a blank name.

Deleting the \texttt{k=2} element (\texttt{bob}) from the \texttt{list} on the left, should give \texttt{list} on the right with empty last term.

| list(1)='ann' | list(1)='ann' |
| list(2)='bob' | list(2)='jack' |
| list(3)='jack' | list(3)='john' |
| ... | ... |
| list(9)='stacie' | list(8)='stacie' |
| list(10)='stacie' | list(9)='' |

Blanking \texttt{list(n)='bob'} to \texttt{list(n)=''} isn’t enough, names from index \texttt{n} to 9 must be shifted up to earlier positions.

The last step is to set \texttt{list(9)=''}.

Write a program which asks the user for the index \texttt{k}, then calls \texttt{deletek} and then prints the resulting.

Hint: Get the program working first with \texttt{k} initialized to say 4. Then uncomment lines which ask the user for the value of \texttt{k} and then reads \texttt{k} from the keyboard.
**Classwork 21.2(6) insertname.f95.**

Let \( \text{list} \) be a list of 9 names in lexicographical order with an empty 10th position. Write a subroutine \( \text{insertname}(\text{name}, \text{list}, n) \) which inserts \text{name} into the \text{list} in the correct place so that the result is in lexicographical order.

---

Inserting the name \text{name}='hope' into \text{list} on the left, gives the \text{list} on the right with no empty name. To insert name in the alphabetically correct location, you must shift the names after the insertion down the list, e.g., \text{stacie} might have to move from index 9 to index 10. The order in which these shifts are done is important.

Hint: Find the first index such that \( \text{list}(\text{index}) < \text{name} \). Move items currently at and below \text{index} down as in the subroutine \text{insertblank}. Then insert the new name in the \text{index} position.

---

```
list(1)='ann'
list(2)='bob'
list(3)='jack'
list(4)='john'
list(5)='mandy'
list(6)='naomi'
list(7)='nick'
list(8)='renee'
list(9)='stacie'
list(10)='
```

---

```
list(1)='ann'
list(2)='bob'
list(3)='jack'
list(4)='hope'
list(5)='jack'
list(6)='naomi'
list(7)='nick'
list(8)='renee'
list(9)='stacie'
list(10)='
```
Let list be a list of 9 names in lexicographical order with an empty 10th position. Write a subroutine deletename(name,list,n) which deletes name from list and then closes any gaps in the list so that only the last element is a blank name. You may assume name occurs in list.

Deleting the name bob from the list on the left, should give the list on the right with empty last term.

Blanking list(n)='bob' to list(n)='' isn't enough, names from index n to 9 must be shifted up to earlier positions.

The last step is to set list(9)=''.

Hint: First find the index k such that list(k) = name. Use the subroutine deletek to delete this position and move the remainder of the list up to close the gap.

Hint: Get the program working first with name initialized to say john. Then uncomment lines which ask the user for the name and then reads name from the keyboard.
**TYPES AND POINTERS**

For a phone directory, instead of having separate vectors for names and numbers,

```plaintext
name(1)='ann'; number(1)='8907'
name(2)='bob'; number(2)='3526'
name(3)='jack'; number(3)='9360'
name(4)='john'; number(4)='6639'
name(5)='mandy'; number(5)='1104'
```

we'd like to have one vector, `record`, where each record is a *name* and a *number*. Matrices and vectors must be all integer, all real, or all character of the same size, no mixing of data types. A record with a mix of data types, requires a *derived type* defined with a `type ... endtype` statement. Note how the derived type `record` (a mix of a 7-character *name* and a 4-character *number*) is defined and the two ways it can be assigned (2 ways with vectors: `v=[9,5]` and `v(1)=9; v(2)=5`).

**Example**

```fortran
!type.f95
program test_type
  type phone_record
    character(7)::name
    character(4)::number
  endtype
  type(phone_record)::record(5)
  record(1)= phone_record('ann','8907')
  record(2)= phone_record('bob','3526')
  record(3)= phone_record('jack','9360')
  record(4)%name = 'john'
  record(4)%number = '6639'
  record(5)%name = 'mandy'
  record(5)%number = '1104'
  do i=1,5
    print*,record(i)
  enddo
endprogram
```

Deleting a record, requires moving all subsequent records up to fill the hole. Inserting a record, requires moving all records down to make space for the new record. This is too time-consuming for a million-record phone directory. Instead, large databases use *pointers* to form *linked-lists* (slow search) or *binary trees* (for binary search). Regular variables have separate memory locations. `b=a`, copies what is in `a` to `b`. But since `a` and `b` have are in separate locations, subsequent changes to one will not affect the other. `p => a` points pointer `p` to the location or address of `a`. Thus `p` is an alias for `a`. Since they are two names for the same location, changing `p` changes `a`, changing `a` changes `p`. Pointers are declared with the keyword `pointer`. The variables they point to are declared to be *targets*.

A vector of records can be replaced by a *linked list* in which each record has, in addition to a name and number, a pointer which points to the next record. The payoff is that, deleting or adding a record means changing a couple of
pointers, not moving million of items up or down in a vector.

The linked-list below has an artificial head record and tail record marking the beginning and end of the list. All real records will lie in between.

Suppose we want to insert a new record

\[
\text{new} = ['cc', '3542', \text{next}].
\]

Find the first record \( p \) such that

\[
'cc' = \text{new}\%\text{name} < p\%\text{next}\%\text{name}.
\]

\[
\begin{align*}
\text{head} &= [' ','0000', \text{next}] \\
['aa','4324', \text{next}] \\
p &= ['bb','5282', \text{next}] \\
p\%\text{next} &= ['cc','3542', \text{next}] \\
\text{new} &= ['cc','3542', \text{next}] \\
p\%\text{next} &= ['dd','0641', \text{next}] \\
\text{tail} &= ['zzzz','9999', \text{null()}]
\end{align*}
\]

First set, \( \text{new}\%\text{next} \Rightarrow p\%\text{next} \)
and then set \( p\%\text{next} \Rightarrow \text{new} \)

Now suppose we wish to delete the record with name 'cc'. First record \( p \) such that 'cc' is the name of the following record, i.e.,

\[
'cc' = p\%\text{next}\%\text{name}
\]

\[
\begin{align*}
\text{head} &= [',',0000', \text{next}] \\
[',',4324', \text{next}] \\
p &= [',',5282', \text{next}] \\
p\%\text{next} &= [',',3542', \text{next}] \\
p\%\text{next}\%\text{next} &= [',',0641', \text{next}] \\
\text{tail} &= ['zzzz','9999', \text{null()}]
\end{align*}
\]

To delete \( p\%\text{next} \), just by pass it, i.e., set \( p\%\text{next} \Rightarrow p\%\text{next}\%\text{next} \)

The old record is still in memory but it is no longer on the linked-list. A deallocate statement makes the memory location available again.
program linked_list

containssubroutine insert(new)
type(phone_record),target::new
  p => head
do
    if( LLT(new%name, __________ ) ) exit;
    p => p%next
  enddo
new%next => __________
p%next => ____________
return
end subroutine

subroutine delete(name)
  character(7)::name
  p => head
do
    if( name == _________ ) exit;
    p => p%next
  enddo
  p%next => _________
end subroutine

subroutine list_printer
  p => head%next
do
    if(p%name == 'zzzz') exit
    print*, p%name, p%number
    p => p%next
  enddo
end subroutine

endprogram