Linked Lists

In this section of notes you will learn how to create and manage a dynamic list.

Arrays

Easy to use but suffer from a number of drawbacks:

1. Fixed size
2. Adding/Deleting elements can be awkward

Arrays: Fixed Size

- The size of the array cannot be dynamically changed once the memory has been allocated
- The following example won't work:

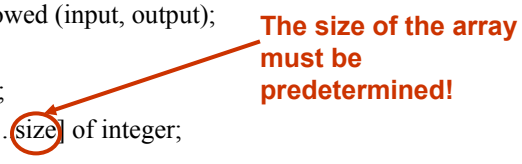
```
program notAllowed (input, output);
var
    size : integer;
    arr  : array [1..size] of integer;
begin
    write('Enter the size of the array: ');
    readln(size);
end.
```

- The workaround is to allocate more space than you need

Arrays: Fixed Size

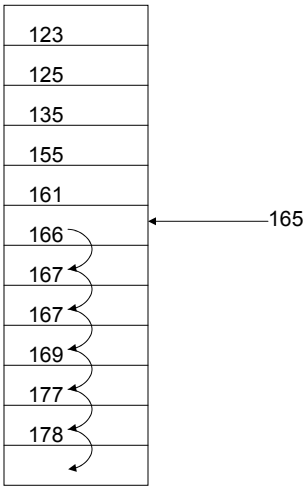
- The size of the array cannot be dynamically changed once the memory has been allocated
- The following example won't work:

```
program notAllowed (input, output);
var
    size : integer;
    arr  : array [1..size] of integer;
begin
    write('Enter the size of the array: ');
    readln(size);
end.
```

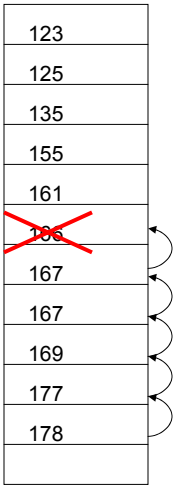


- The workaround is to allocate more space than you need

Arrays: Adding Elements In The Middle

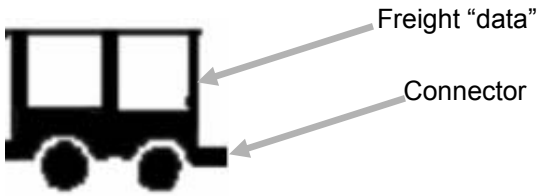


Arrays: Deleting Elements From The Middle



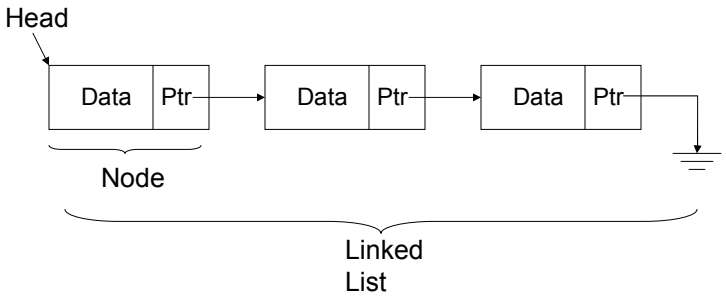
What's Needed

- A composite type that stores data dynamically and can allow for the quick addition and removal of elements



Alternative To Arrays: Linked Lists

- More complex coding may be required
- Some list management functions are more elegant (and faster)



Common List Functions

1) Declaring the list
2) Creating a new list
3) Traversing the list (display)
4) Adding a node to the list
5) Searching the list
6) Removing a node from the list

Note: These list functions will be illustrated by portions of an example that is a modified version of the investors program from the section on sorting, but implemented as a linked list rather than as array. The complete program can be found in Unix under:
/home/231/examples/linkedLists/investors.p

1. Declaring A Linked List

Part III: What is does the entire freight car consist of? (Data and link)

Part I: What is the data?

Part II: What is the connector linking?

Format:

(* **Part I**: Defining a new type for the data (necessary if the data field is not a built-in type *)

(* **Part II**: Defining a pointer to the new type "Node" *)
Name of the list pointer = ^ Node;

(* **Part III**: Defining a new type, a "Node" *)
type
  Node = record
    data : *Name of the list data*;
    nextPointer : *Name of the list pointer*;
  end;

1. Declaring A Linked List (2)

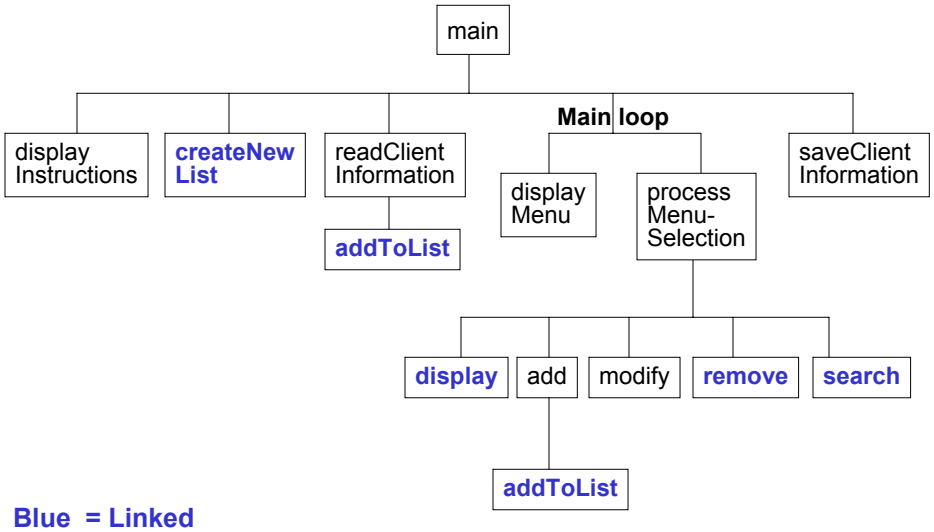
Example:

```
type
   (* Part I: Defining a new type for the data (necessary because a "Client" is
   not a built-in type *)
   Client =  record
               firstName : string [NAME_LENGTH];
               lastName  : string [NAME_LENGTH];
               income    : real;
               email     : string [EMAIL_LENGTH];
             end; (* Declaration of record Client *)

   (* Part II: Defining a pointer to the new type "Node" *)
   NodePointer = ^ Node;
```


1. Declaring A Linked List (3)

```
(* Part III: Defining a new type, a "Node" *)
Node = record
        data        : Client;
        nextPointer : NodePointer;
      end; (* Declaration of record Node *)
```

Outline Of The Example Program



Blue = Linked list functions

Main Procedure: Example Program

```
begin
  var tamjClientList        : NodePointer;
  var menuSelection         : char;
  displayInstructions;
  createNewList(tamjClientList);
  readClientInformation(tamjClientList);
  repeat
  begin
     displayMenu;
     readln(menuSelection);
     processMenuSelection(tamjClientList,menuSelection);
  end; (* repeat-until *)
  until (menuSelection = 'Q') OR (menuSelection = 'q');
  saveClientInformation(tamjClientList);
end.
```

Reminder: How Should Pointers Be Passed

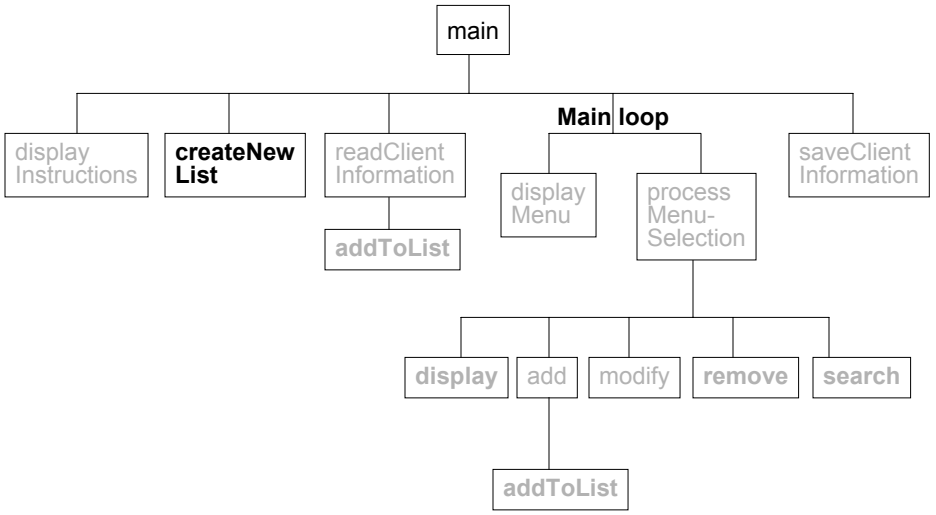
Passing pointers as value parameters:

- When the pointer itself doesn't have to change in the module (e.g., it doesn't point to a different node).
- The node/nodes pointed to by the pointer can still be changed.

Passing pointers as variable parameters:

- When the pointer itself does have to change in the module (e.g., it is made to point to a different node).
- The node/nodes pointed to by the pointer can still be changed.

2. Creating A New List



2. Creating A New List (2)

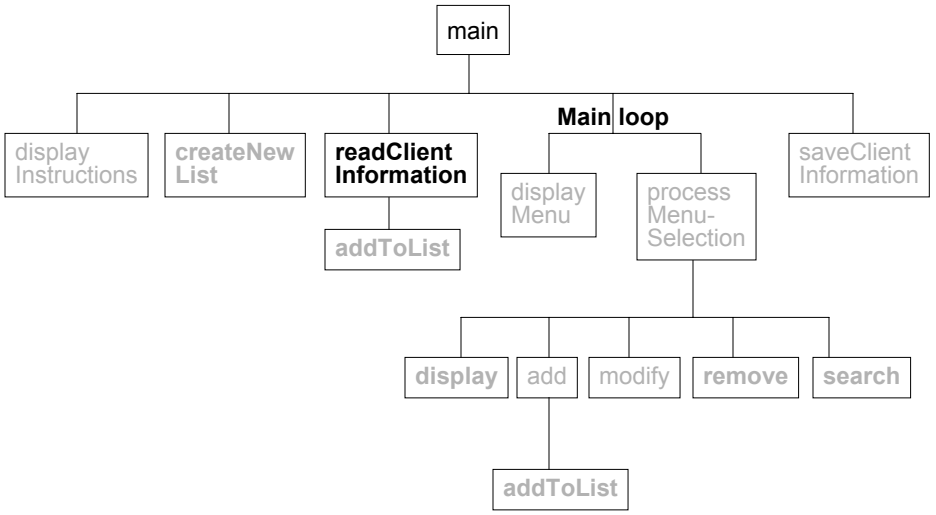
Description:

The pointer to the beginning of the list is passed into the procedure as a variable parameter and initialized to NIL signifying that the new list is empty.

Example:

```
procedure createNewList (var aClientList : NodePointer);
begin
  aClientList := NIL;
end; (* createNewList *)
```


Reading The Client Information From A File



Reading The Client Information From A File (2)

```
procedure readClientInformation (var aClientList  : NodePointer);
var
  newNode        : NodePointer;
  newClient      : Client;
  investorData   : text;
  inputFileName : string [MAX_FILE_NAME_LENGTH];
begin;
  writeln;
  write('Enter the name of the input file: ');
  readln(inputFileName);
  reset(investorData, inputFileName);
  writeln('Opening file ', inputFileName, ' for reading');

  if EOF (investorData) then
  begin
    writeln('File ', inputFileName, ' is empty, nothing to read.');
  end
```


Reading The Client Information From A File (3)

```
 else
  begin
    while NOT EOF (investorData) do
    begin
       new(newNode);
       with newClient do
       begin
          readln(investorData, firstName);
          readln(investorData, lastName);
          readln(investorData, income);
          readln(investorData, email);
          readln(investorData);
       end; (* with-do: single client records *)
       newNode^.data := newClient;
       addToList (aClientList, newNode);
    end; (* While: reading from file *)
```

Reading The Client Information From A File (4)

```
  end; (* else *)
  close(investorData);
end; (* readClientInformation *)
```


Processing The Main Menu Of Options

```
procedure processMenuSelection (var aClientList     : NodePointer;
                                    menuSelection : char);
begin
  case (menuSelection) of
    'D', 'd' :
    begin
      display (aClientList);
    end;

    'A', 'a' :
    begin
      add (aClientList);
    end;

    'R', 'r' :
    begin
      remove (aClientList);
    end;
```

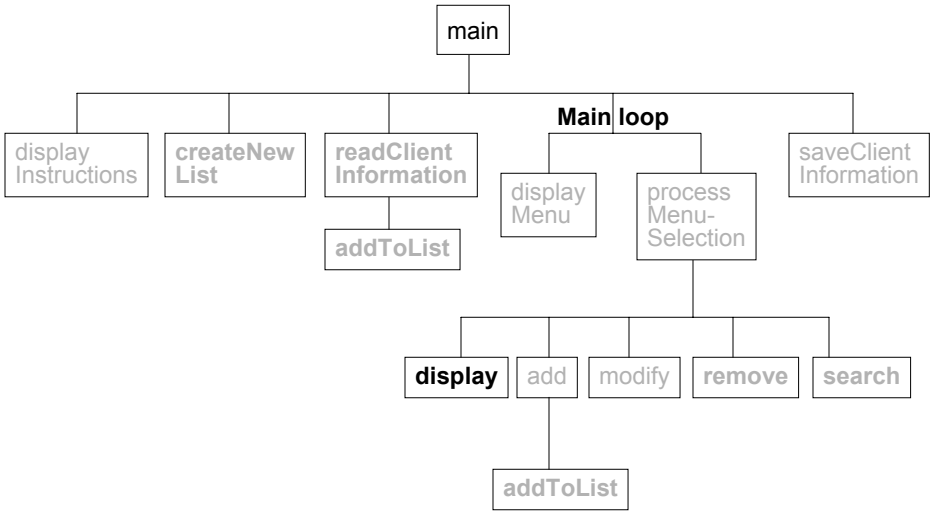
Processing The Main Menu Of Options (2)

```
'M', 'm' :
begin
  modify(aClientList);
end;

'S', 's' :
begin
  search(aClientList);
end;

'Q', 'q' :
begin
  writeln;
  writeln('Thank you for using the investor 2000 (TM) program.');
  writeln('Come again!');
  writeln;
end;
```


Processing The Main Menu Of Options (3)

```
    else
    begin
      writeln;
      writeln('Please enter one of the following options: d, a, r, m, s or q');
      writeln;
    end;
  end; (* case *)
end; (* End of procedure processMenuSelection *)
```

3. Traversing The List: Display



3. Traversing The List: Display (2)

Description:

Steps (traversing the list to *display* the data portion of each node onscreen)

1. Start by initializing a temporary pointer to the beginning of the list.
2. If the pointer is NIL then display a message onscreen indicating that there are no nodes to display and stop otherwise proceed to next step.
3. While the temporary pointer is not NIL:
   a) Process the node (e.g., display the data onscreen).
   b) Move on to the next node by following the node's nextPointer (set the pointer to point to the next node).

3. Traversing The List: Display (3)

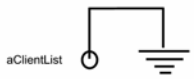
Example:

```
procedure display  (aClientList : NodePointer);
var
  i    : integer;
begin
  writeln('CLIENT LIST':19);
  for i := 1 to 20 do
    write('--');
  writeln;

  if (aClientList = NIL) then
  begin
    writeln;
    writeln('List is empty, no clients to display.');
    writeln;
  end;
```

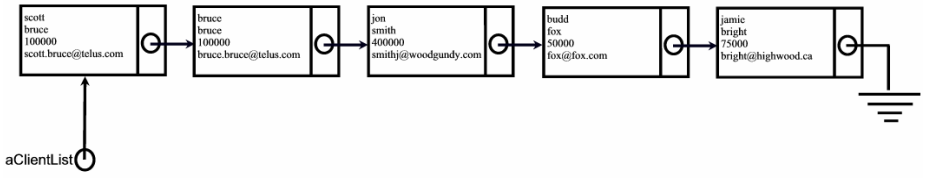

3. Traversing The List: Display (4)

```
  while (aClientList <> NIL) do
  begin
    writeln('First name: ':20, aClientList^.data.firstName);
    writeln('Last Name: ':20, aClientList^.data.lastName);
    writeln('Income $':20, aClientList^.data.income:0:2);
    writeln('Email: ':20, aClientList^.data.email);
    writeln;
    aClientList := aClientList^.nextPointer;
  end; (* While: Traversing the list *)
end; (* displayList *)
```

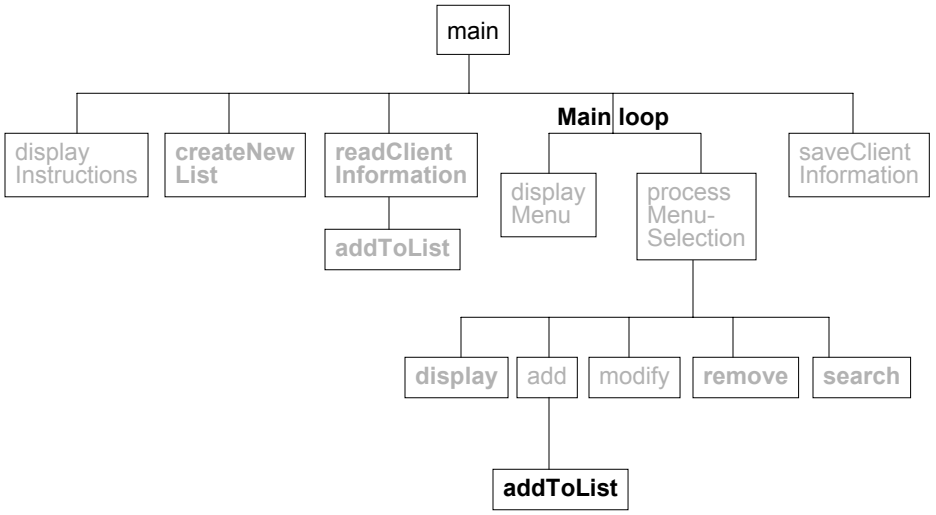
3. Traversing The List: Display (5)



3. Traversing The List: Display (6)



4. Outline For Adding A Node To The End Of The List



4. Adding A Node To The End Of The List

Description:

Variables

1. There are two pointers to the list:
   a) Current pointer – traverses the list from beginning to end.
   b) Previous to first pointer – points to the node that just before to the end of the list.

4. Adding A Node To The End Of The List (2)

Steps:

1. Assign the current pointer to the front of the list.
2. If the current pointer is NIL, then the list is empty. Add the node to the front of the list by changing the head pointer and stop.
3. Otherwise traverse the list with two pointers, one pointer (the current pointer) goes past the end of the list (to the NIL value), the other pointer (previous pointer) stays one node behind the current pointer.
4. Attach the new node to the last node in the list (which can be reached by the previous pointer).
5. Whether the node is attached to an empty or non-empty list, the next pointer of the new node becomes NIL (to mark the end of the list).

4. Adding A Node To The End Of The List (3)

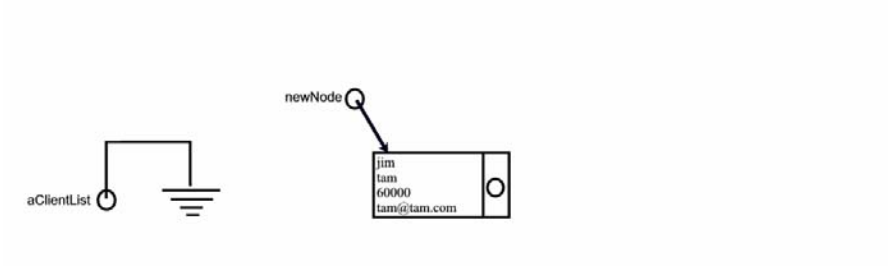
Example:

```
procedure addToList  (var aClientList : NodePointer;
                                newNode    : NodePointer);
var
  currentNode  : NodePointer;
  previousNode : NodePointer;
begin
   if (aClientList = NIL) then
  begin
    aClientList := newNode;
  end (* If: Adding a new node to the front of the list. *)
```

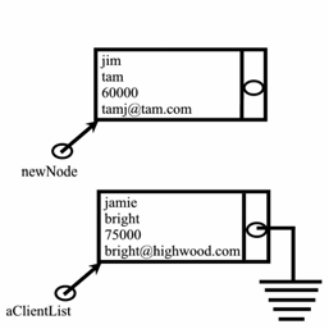
4. Adding A Node To The End Of The List (4)

```
  else
  begin
    currentNode := aClientList;
    while (currentNode <> NIL) do
    begin
      previousNode := currentNode;
      currentNode := currentNode^.nextPointer;
    end; (* While : Found the last element in the list. *)
    previousNode^.nextPointer := newNode;
  end; (* Else: Adding a new node to a non-empty list. *)
  newNode^.nextPointer := NIL;
end; (* addToList *)
```

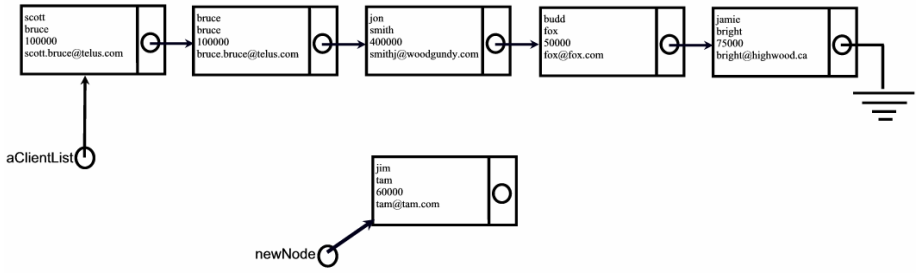

4. Adding A Node To The End Of The List (5)



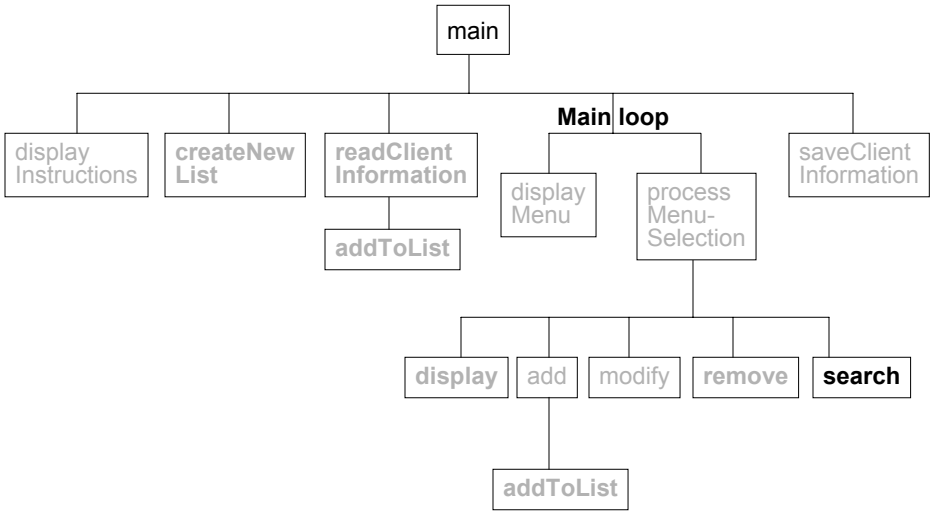
4. Adding A Node To The End Of The List (6)



4. Adding A Node To The End Of The List (7)



5. Searching The List



5. Searching The List (2)

Main variables:

1. A temporary pointer – used to traverse the list.
2. The search key – in this example it's a string that represents that the last name of a client.
3. A boolean variable that stores that status of the search (the search flag). (Start the search by assuming that it's false that there's a match and the flag is set to true when a successful match occurs.

5. Searching The List (3)

Steps:

1. The temporary pointer starts at the beginning of the list. Since the search has not yet begin, set the search flag to false.
2. If the temporary pointer is NIL then the list is empty. Display a status message (e.g., “client list is empty”) to the user and end the search.
3. While the end of the list has not been reached (when the temporary pointer is NIL) :
   a) Compare the last name field of each client to the search key and if there’s match display all the fields of the client onscreen and set the boolean to true.
   b) Move the temporary pointer onto the next client in the list via the client’s nextPointer field.
4. When the entire list has been traversed and the search flag is still false indicate to the user that no successful matches have been found.

5. Searching The List (4)

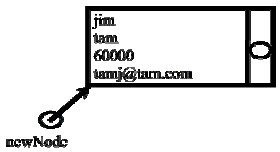
Example:

```
procedure search (aClientList : NodePointer);
var
  desiredName : string [NAME_LENGTH];
   isfound        : boolean;
begin
if (aClientList = NIL) then
  begin
    writeln('Client list is empty: Nothing to search.');
  end (* If: Empty list, stop the search. *)
  else
  begin
      write('Enter last name of contact that you wish to search for: ');
      readln(desiredName);
      isFound := false;
      writeln;
```

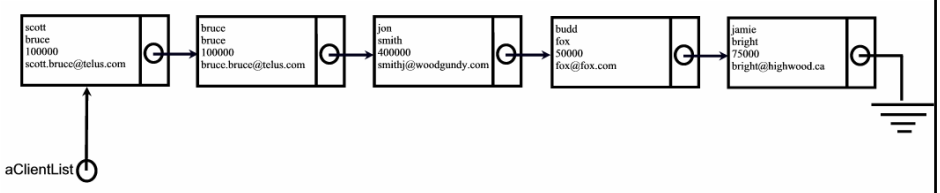
5. Searching The List (5)

```
    while (aClientlist <> NIL) do
    begin
      if (desiredName = aClientList^.data.lastName) then
      begin
        isFound := true;
        writeln('Found contact':20);
        writeln('First name :':20, aClientList^.data.firstName);
        writeln('Last name :':20, aClientList^.data.lastName);
        writeln('Income $':20, aClientList^.data.income:0:2);
        writeln('Email :':20, aClientList^.data.email);
        writeln;
      end; (* If: Match was found. *)
      aClientList := aClientList^.nextPointer;
    end; (* While: Finished traversing the list. *)
    if (isFound = False) then
      writeln('No clients with the last name of '", desiredName, "' were '
            'found in list');
  end; (* Else: Non-empty list was searched. *)
end; (* search *)
```


Searching The List (6)

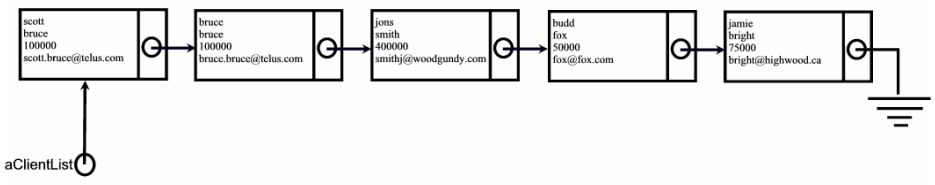


5. Searching The List (7)



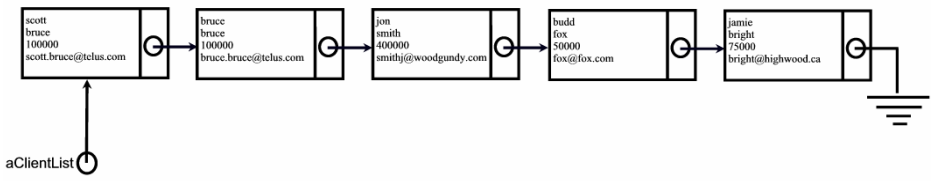
Desired name: bruce

5. Searching The List (8)



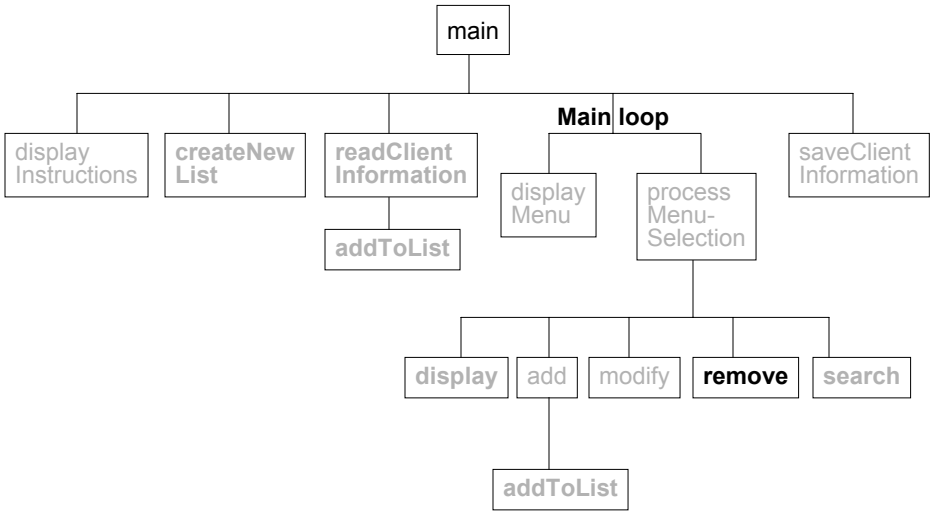
desiredName: tam

5. Searching The List (9)



desiredName: fox

6. Removing A Node From The List



6. Removing A Node From The List (2)

Description:

Main variables:

1. A temporary pointer that points to the node to be deleted. It is needed so that the program can retain a reference to this node and free up the memory allocated for it after the node has been 'bypassed' (step 4A and 4 B on the next slides).
2. A previous pointer that points to the node just prior to the one to be deleted. The nextPointer field of this pointer will be set to skip over the node to be deleted and will instead point to the node that immediately follows.
3. The head pointer. The actual pointer (and not a copy) is needed if the first node is deleted.
4. The search key – in this example it is a string that represents that the last name of a client.
5. A boolean variable that stores that status of the search (the search flag). (Start the search by assuming that it's false that there's a match and the flag is set to true when a successful match occurs.

6. Removing A Node From The List (3)

Steps

1. Initialize the main variables.
   a) The temporary pointer starts at the front of the list.
   b) The boolean flag is set to false (no matches have been found yet).
   c) The previous pointer is set to NIL (to signify that there is no element prior to the first element).
2. If the list is empty (temporary pointer is NIL) display a status message to the user (e.g., "client list is empty") and end the removal process.
3. While the end of the list has not been reached (temporary pointer is not NIL) AND no matches have been found yet (boolean flag is false) :
   a) Compare the search key with the last name field of the client node referred to by the temporary pointer.
   b) If there's a match then set the search flag to true (it's true that a match *has* been found now).
   c) If no match has been found set the previous pointer to the client referred to by the temporary pointer and move the temporary pointer to the next client in the list.

6. Removing A Node From The List (4)

4. (At this pointer either the whole list has been traversed or there has been successful match and the search has terminated early):
   a. If the search flag is set to true then a match has been found.
      i. If the first node is the one to be deleted (previous pointer is NIL) then set the head pointer to the second client in the list.
      ii. If any other node is to be deleted then bypass this node by setting the nextPointer field of the node referred to by the previous pointer to the node immediately following the node to be deleted.
      iii. In both cases the temporary pointer still refers to the node to be deleted. Free up the allocated memory using the temporary pointer.
   b. If the search flag is set to false no matches have been found, display a status message to the user (e.g., "no matches found").

6. Removing A Node From The List (5)

Example:

```
procedure remove (var aClientList : NodePointer);
var
  desiredName   : string[NAME_LENGTH];
  previousFirst : NodePointer;
  temp          : NodePointer;
  isFound       : boolean;
begin
  isFound := false;
  previousFirst := NIL;
  temp := aClientList;
```

6. Removing A Node From The List (6)

```
(* Case 1: Empty list *)
if (temp = NIL) then
begin
  writeln('List is already empty, no clients to remove.');
end (* If: empty list *)


(* Case 2: Non-empty list *)
else
begin
  write('Enter last name of client to remove: ');
  readln(desiredName);
```


6. Removing A Node From The List (7)

```
while (temp <> NIL) And (isfound = false) do
begin
  if (temp ^.data.lastName = desiredName) then
  begin
    isfound := true;
  end (* If: Found a match *)
  else
  begin
    previousFirst := temp;
    temp := temp^.nextPointer;
  end; (* Else: No match found, continue search *)
end; (* While loop:  To iterate through the client list. *)
```

6. Removing A Node From The List (8)

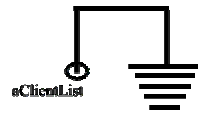
```
(* Case 2A or 2B: Removing a node in the list. *)
if (isFound = true) then
begin
  writeln('Removing first instance of client with surname of ',
        desiredName, ':');
  writeln('First name :':15, temp^.data.firstName);
  writeln('Last name :':15, temp^.data.lastName);
  writeln('Income $':15, temp^.data.income:0:2);
  writeln('Email :':15, temp^.data.email);
  writeln;

 (* Case 2A: Removing the first node from the list. *)
  if (previousFirst = NIL) then
  begin
    aClientList := aClientList^.nextPointer;
  end (* If: Removing the first node. *)
```


6. Removing A Node From The List (9)

```
    (* Case 2B: Removing any node except for the first. *)
    else
    begin
      previousFirst^.nextPointer := temp^.nextPointer;
    end; (* Else: removing any node except for the first. *)

    dispose(temp);
  end (* If: Match found and a node was deleted. *)

  (* Case 2C: The entire list was searched but no matches were found. *)
  else
  begin
    writeln('No clients with a surname of ', desiredName, ' found in the '
          'list of clients.');
  end; (* Else: No matches found. *)
end; (* Else: Non-empty list. *)
end; (* remove *)
```

Removing A Node From The List (10)

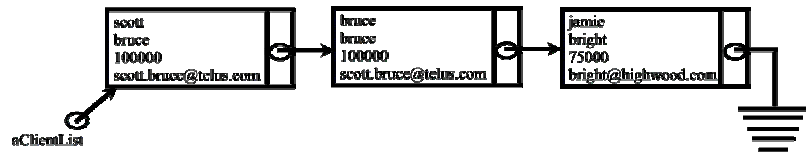
Case 1: Empty List



desiredName: bruce
isFound

Removing A Node From The List (11)

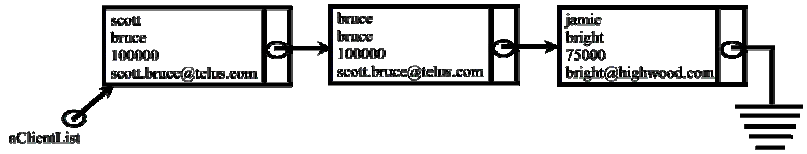
Case 2A: Removing the first node



desiredName: bruce
isFound

6. Removing A Node From The List (12)

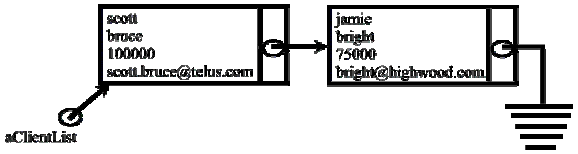
Case 2B: Removing any node except for the first



desiredName: bright
isFound

6. Removing A Node From The List (13)

Case 2C: Removing a node a non-empty list but no matches were found.



desiredName: tam
isFound

You Should Now Know

- What is a linked list
- What are the advantages of using a linked list over using an array
- What is the disadvantage of using a linked list over using an array
- Common list operations
  - Declaring a list
  - Creating a new list and initializing the list with data
  - Traversing the list (e.g., to display the contents of the nodes)
  - Adding new nodes to the list
  - Searching the list
  - Deleting an existing node from the list