Link 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 must be stated when the program is compiled

The following example won't work:

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

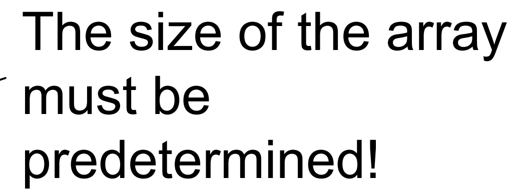
The workaround is to allocate more space than you need

Arrays: Fixed Size

The size of the array must be stated when the program is compiled

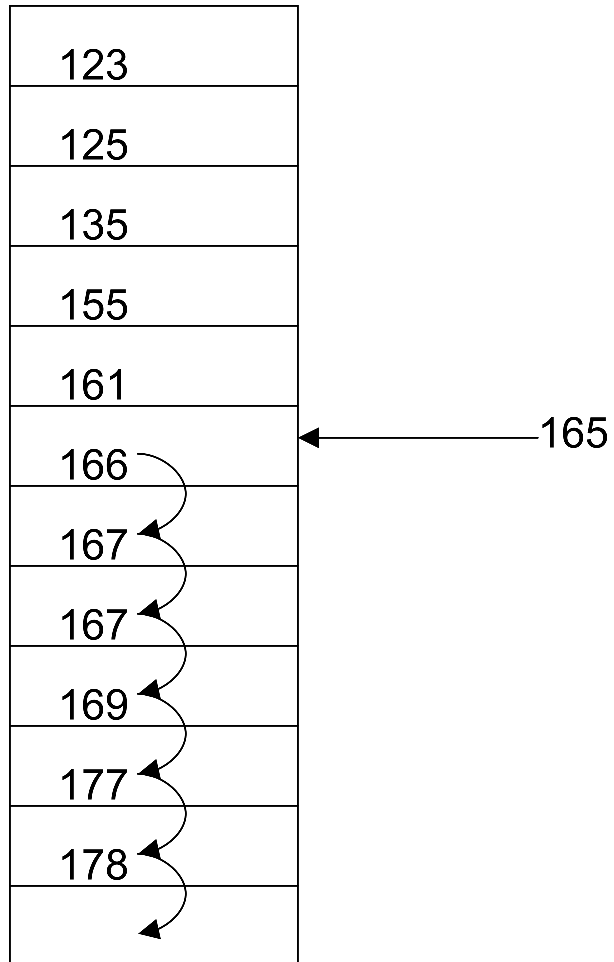
The following example won't work:

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

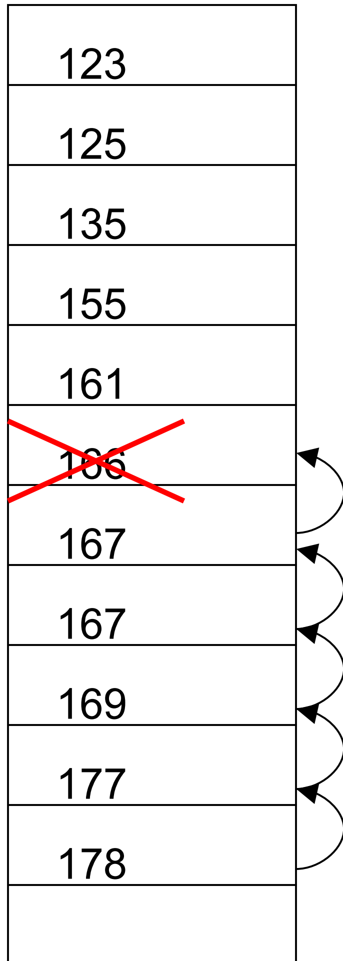


The workaround is to allocate more space than you need

Arrays: Adding Elements In The Middle



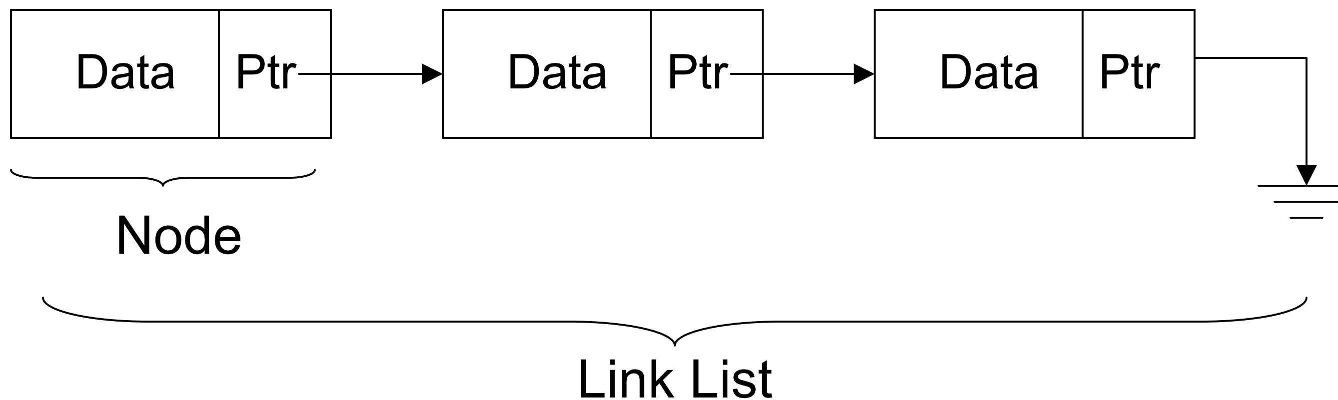
Arrays: Deleting Elements From The Middle



Alternative To Arrays: Link Lists

More complex coding may be required

Some list management functions are more elegant (and faster)



Some Link List Functions

1)Declaring a link list

2)Creating a new list

3)Traversing the list

4)Adding a node to the list

5)Searching the list

6)Deleting a node from the list

Note: These list functions will be illustrated by portions of an example program. This program is the investors program but implemented as a link list rather than as array. The complete program can be found in Unix under: /home/231/examples/link_lists/investors.p

Declaring A Link List

Syntax:

```
type
  Name of the list data = Type of the list data;
  Name of the list pointer = ^ Node;
  Node = Record
        data           : Name of the list data;
        nextPointer : Name of the list pointer;
  Name of link list = Name of list pointer;
```

Declaring A Link List (2)

Example:

```
type
    Client =  record
          firstName : array [1..NAMELENGTH] of char;
          lastName  : array [1..NAMELENGTH] of char;
          income      : real;
          email        :  array [1..EMAILLENGTH] of char;
        end; (* Declaration of record Client *)

   ListData = Client;
   ListPointer = ^ Node;
   Node = record
        data              : ListData;
        nextPointer : ListPointer;
      end; (* Declaration of record Node *)

  ClientList = ListPointer;
```

Creating A New List

Algorithm:

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 tamjClientList : ClientList);
begin
    tamjClientList := NIL;
end; (* Procedure *)
```

Traversing The List

Algorithm:

Steps

1. Start by initializing a pointer to the beginning of the list.
2. If the pointer is NIL then show message indicating that there are no nodes to display and stop.
3. Process the node.
4. Move on to the next node by following the node's nextPointer (set pointer to point to the next node).
5. Repeat 3 – 4 until the end of the list is reached (pointer is NIL).

Traversing The List (2)

Example:

```
procedure displayList (    tamjClientList : ClientList);
var
  currentNode : ListPointer;
begin
  currentNode := tamjClientList;
  writeln('CLIENT LIST':20);
  if (currentNode = NIL) then
  begin
    writeln;
    writeln('List is empty, no clients to display');
    writeln;
  end; (* if-then *)
```

Traversing The List (3)

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

Adding A Node To The The List

Algorithm:

Steps

1. Assign a pointer to the front of the list.
2. If the pointer is NIL then the list is empty and add the node to the front of the list and stop.
3. Otherwise traverse the list with two pointers, one pointer (current pointer) goes to the end of the list, the other stays one node behind it (previous pointer).
4. Attach the new node to the last node in the list (the one reached by the previous pointer).
5. The next pointer of the new node becomes NIL (indicating that this is the end of the list).

Adding A Node To The List (2)

Example:

```
procedure addToList  (var tamjClientList: ClientList;
                          newNode        : ListPointer);
var
    currentNode   : ListPointer;
    previousNode : ListPointer;
begin

    (* Empty list add new node to front *)
    if (tamjClientList = NIL) then
    begin
        tamjClientList := newNode;
        newNode^.nextPointer := NIL;
    end
```

Adding A Node To The List (3)

```
  else
  begin
    currentNode := tamjClientList;
    while (currentNode <> NIL) do
    begin
      previousNode := currentNode;
      currentNode := currentNode^.nextPointer;
    end; (* while *)
    previousNode^.nextPointer := newNode;
    newNode^.nextPointer := NIL;
  end; (* else *)
end; (* Procedure *)
```

Searching The List

Algorithm:

The procedure is run in order to find a node(s) that has a field which matches some desire value. Either the node or nodes will be found in the list or else the procedure will have searched every node in the list and have found no matches. A flag will be set to true or false indicating the success or failure of the search.

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 occurs just prior to the first successful match.

    Note: The second pointer does not directly come into play when the user only wants to search the list. They are essential when the person wishes to erase a node from the list. Since the erase procedure calls the search procedure, it is is passed in but it's value is not used when the person just wishes to perform a search without performing a deletion.
2. A boolean that indicates the status of the search.

Searching The List (2)

Steps

1. Current pointer starts at the beginning of the list. Since the search has not yet begin, previous is set to NIL and the flag is set to false.
2. A check is performed to determine if the node is a match. If this is the case and the flag is still false (indicating that we haven't found a previous node that was a successful match) set the flag to true (since a match was found). Since the search function requires a list of all matches (and not just the first instance) don't stop searching the list.
3. If the flag is set to false then set the previous pointer to point to the current node. If the flag is set to true then don't change the value of flag (the previous pointer tracks the node just prior to the node which first meets the search criteria).
4. Move on to the next node (by setting the current pointer to the current node's next pointer).
5. Continue step 2 – 4 until the end of the list is reached (current node is NIL).

Searching The List (3)

Example:

```
procedure search (    tamjClientList          : ClientList;
                      desiredName             : NameArray;
                  var isFound                 : boolean;
                  var previousFirst           : ListPointer );
var
   currentNode : ListPointer;
begin
   currentNode := tamjClientList;
   previousFirst := NIL;
   isFound := False;
```

Searching The List (4)

```
while (currentNode <> NIL) do
  begin
     if (desiredName = currentNode^.data.lastName) then
     begin
        writeln('Found contact':20);
        writeln('First name :':20, currentNode^.data.firstName);
        writeln('Last name :':20, currentNode^.data.lastName);
        writeln('Income $':20, currentNode^.data.income:0:2);
        writeln('Email :':20, currentNode^.data.email);
        writeln;
        if (isFound = False) then
             isFound := True;
     end; (* if-then *)
```

Searching The List (5)

```
        if (isFound = False) then
            previousFirst := currentNode;
        currentNode := currentNode^.nextPointer;
    end; (* while *)
  if (isFound = False) then
    writeln('Contact not found in list');
  end; (* search *)
```

Deleting A Node From The List

Algorithm:

Steps

1. Search the list (by calling the search procedure) to determine if there exists a node that matches the necessary criteria for deletion.
2. Check the flag to determine if the search was successful or not. If the flag is false then there is no matching node in the list. Stop. There is no matching node to delete.
3. Check to see if the node to be deleted is the first node in the list or not by determining if the previous node is NIL.
4. If the node to be deleted is the first node then have a temporary pointer point to the first element and make the front of the list the second element.
5. If the node to be deleted is not the first node then have a temporary pointer point to the node to be deleted. Set the next pointer of the previous node point to the node after the node to be deleted (bypassing this node)
6. For steps 4 & 5 free up the memory allocated by the node to be deleted.

Deleting A Node From The List (2)

Example:

```
procedure erase (var tamjClientList : ClientList );
var
   desiredName    : NameArray;
   previousFirst  : ListPointer;
   temp           : ListPointer;
   isFound        : boolean;
begin
   write('Enter last name of client to delete: ');
   readln(desiredName);
   search (tamjClientList, desiredName, isFound, previousFirst);
   if (isFound = True) then
   begin
      writeln('Deleting first instance of ', desiredName);
      if (previousFirst = NIL) then
```

Deleting A Node From The List (3)

```
begin
    writeln('Deleting first instance of ', desiredName);
    if (previousFirst = NIL) then
    begin
      temp := tamjClientList;
      tamjClientList := tamjClientList^.nextPointer;
    end (* if-then *)
    else
    begin
      temp := previousFirst^.nextPointer;
      previousFirst^.nextPointer := temp^.nextPointer;
    end; (* else *)
    dispose(temp);
  end; (* if-then *)
end; (* Procedure *)
```

Summary

You should now know how to perform some common link list operations:

1) Declaration of a link list
2) Creation of a new list
3) Traversal of the list
4) Adding a new node to the list
5) Deleting an existing node from the list
6) Searching the list for a node that meets some criteria