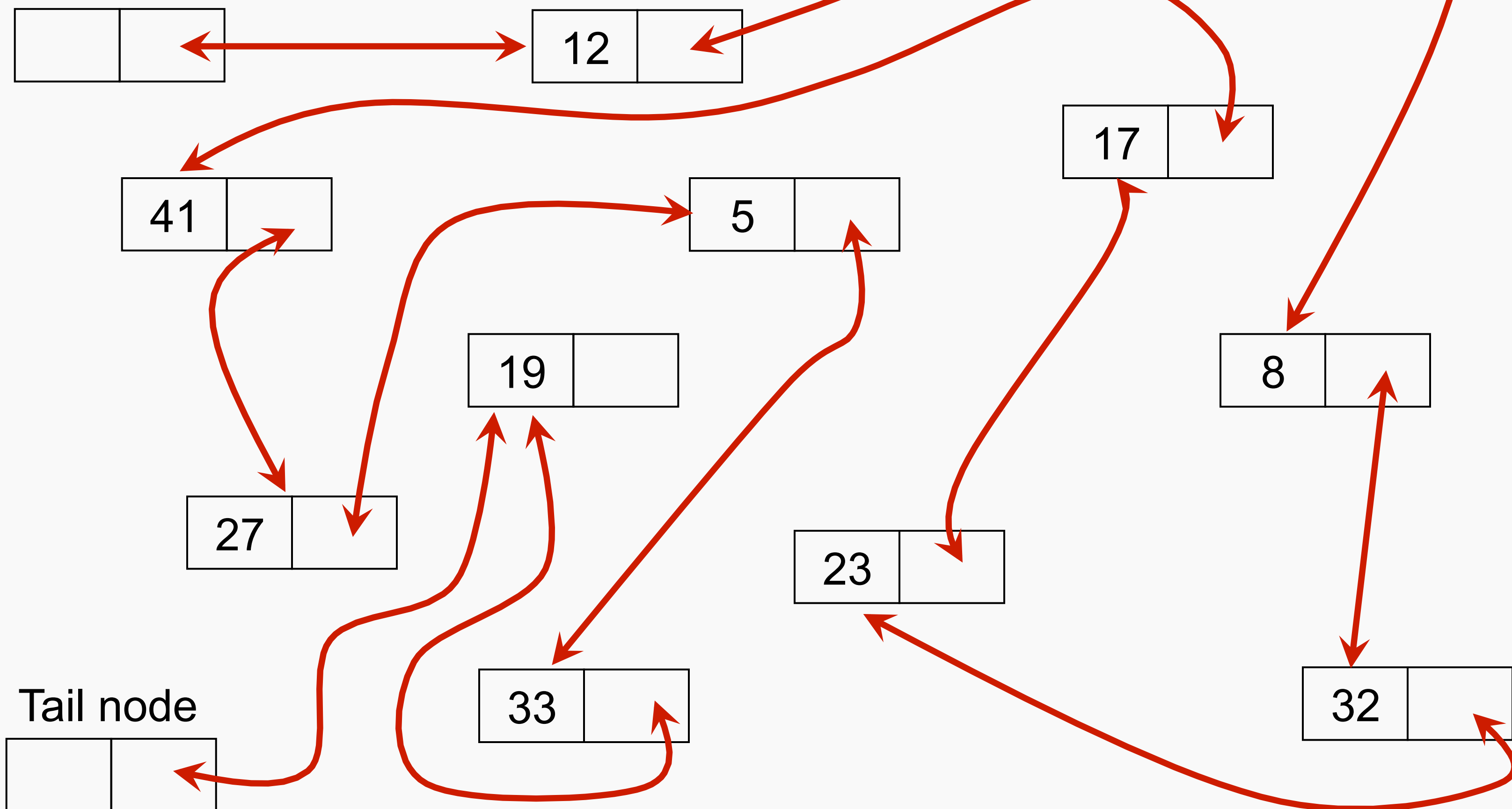
Linked Lists

A linked list is a *data structure* that uses a "chain" of node objects, connected by pointers, to organize a collection of user data values.

Here's a fairly typical conceptual view of a doubly-linked list:

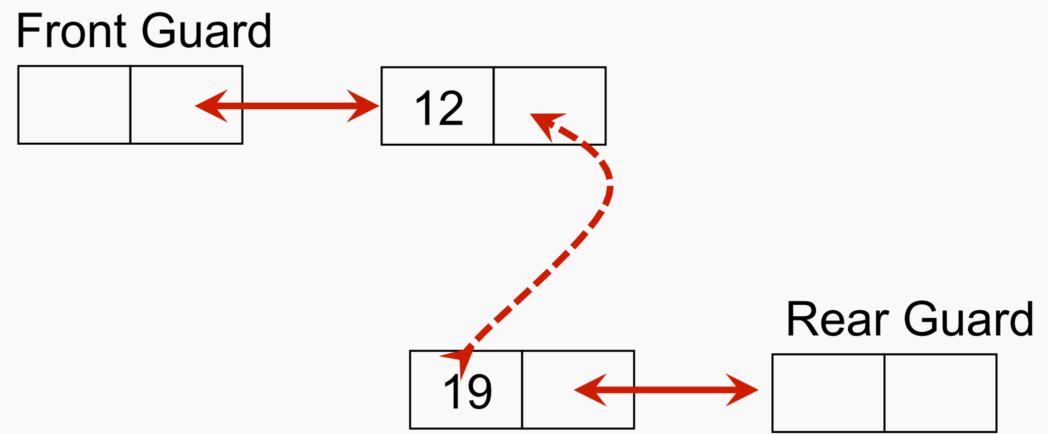


©2005-2015 McQuain

Structural Considerations

The use of "guard" nodes at the front and rear of a list eliminate any "special cases" when implementing insertion/deletion operations.

This way, every "data" node will lie between two nodes.



The common alternative is to simply have pointers to the first and last data nodes, probably stored in a list object. That leads to special cases when operating at the front or rear of the list.

©2005-2015 McQuain

Minimal Linked List Interface

A linked list implementation will typically provide at least:

- initialization function to set up basic structure for an empty list
- insert functions to add new element to the list; at front, at rear, at user-selected position, ordered insertion
- remove function to remove element from the list
- find function to determine whether a given element occurs in the list
- clear function to restore the list to an empty state

In C we would organize this as a pair of `struct` types (list and node) and a collection of associated functions.

 ©2005-2015 McQuain

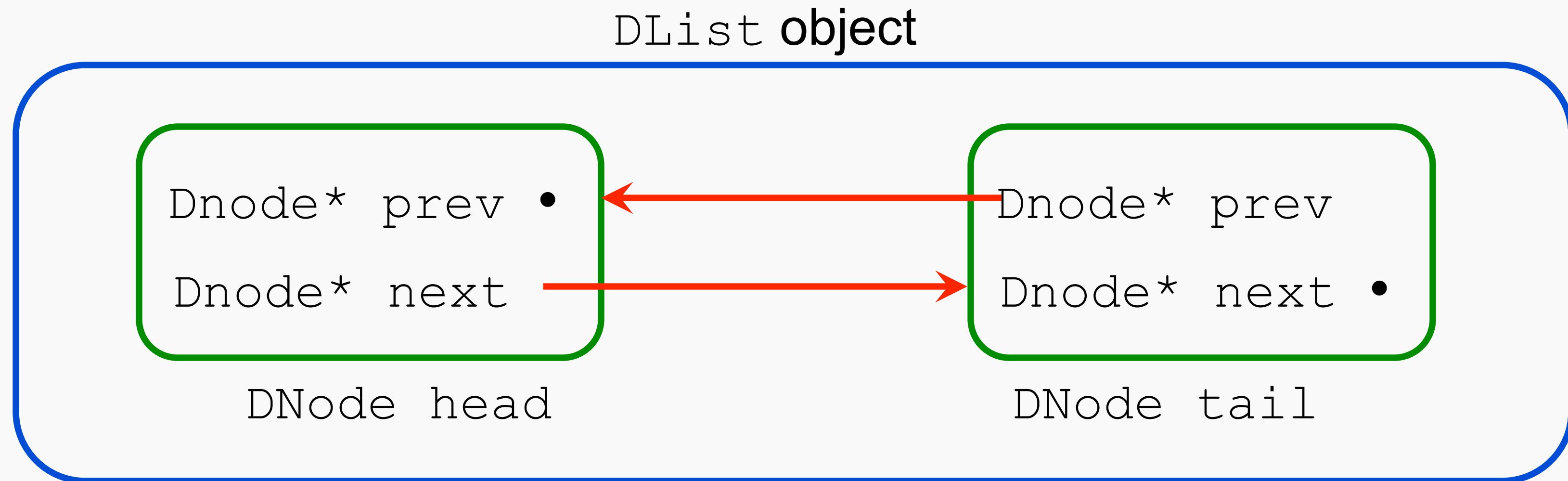
Generic Node and List


```
#ifndef DLIST_H
#define DLIST_H

// List node:
struct _DNode {

   struct _DNode *prev;      // points toward front of list
   struct _DNode *next;      // points toward tail of list
};

// List object:
struct _DList {

   struct _DNode head;       // front guard node for list
   struct _DNode tail;       // rear guard node for list
};

typedef struct _DNode DNode;
typedef struct _DList DList;

#endif
```


©2005-2015 McQuain

DList Initialization

An empty DList will be constructed as shown below:



This eliminates special cases, because every data node will always be between two other nodes.

We could also make head.prev point to tail and tail.next point to head, which would eliminate NULL pointers and allow the list to be used in a circular fashion.

©2005-2015 McQuain

Wrapping the Node in the Payload

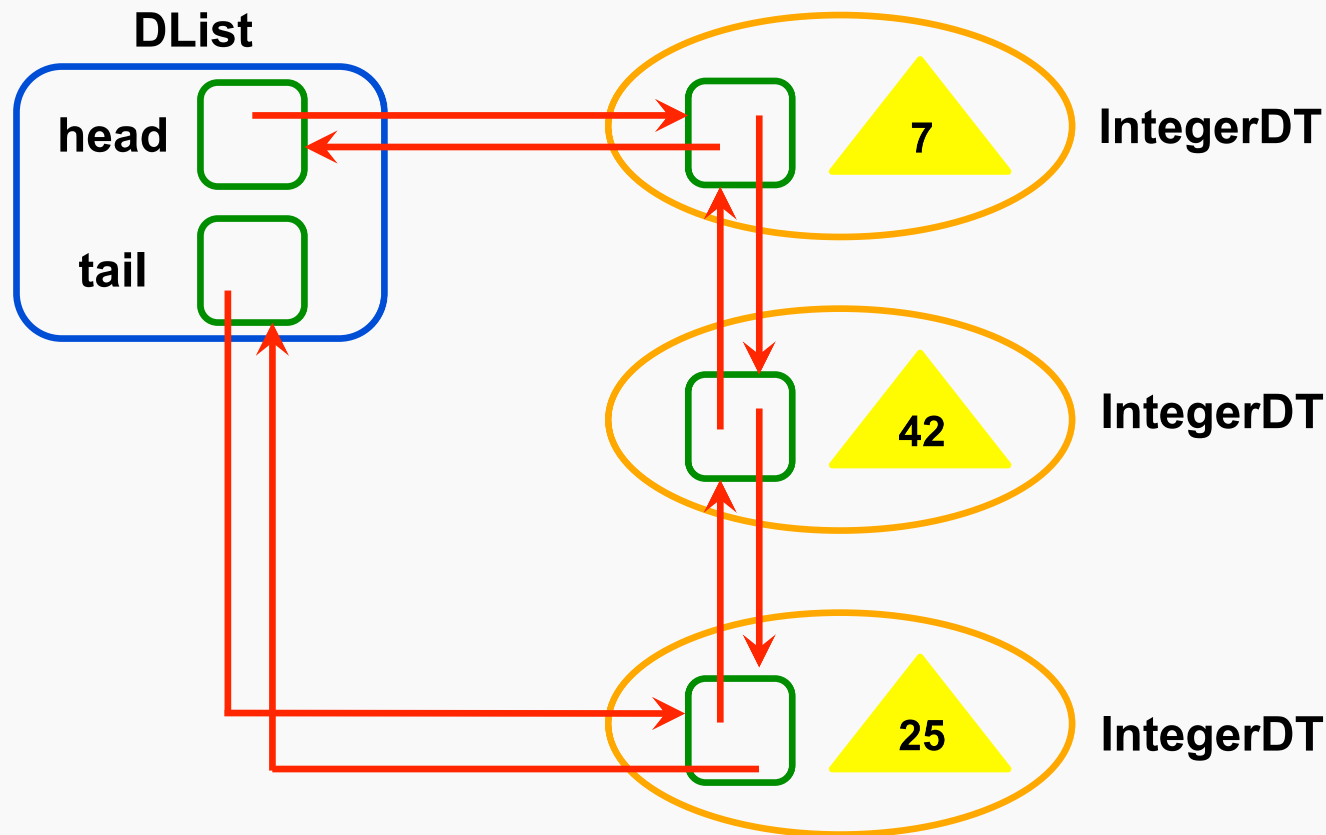
We may use a single `DList` of `DNode` objects with any user data type, without sacrificing type-checking.

We merely have to create a "duct tape" object to attach a data object to a node:

```
#ifndef INTEGERDT_H
#define INTEGERDT_H

#include "DList.h"

struct _IntegerDT {      // "duct tape" attaches data object to DNode
   int   payload;
   DNode node;
};

typedef struct _IntegerDT IntegerDT;

void IntegerDT_Init(IntegerDT* const pLE, const int* const I);

#endif
```


©2005-2015 McQuain

Example of "duct-taped" List Structure



The `DList` only "knows about" two `DNode` objects.

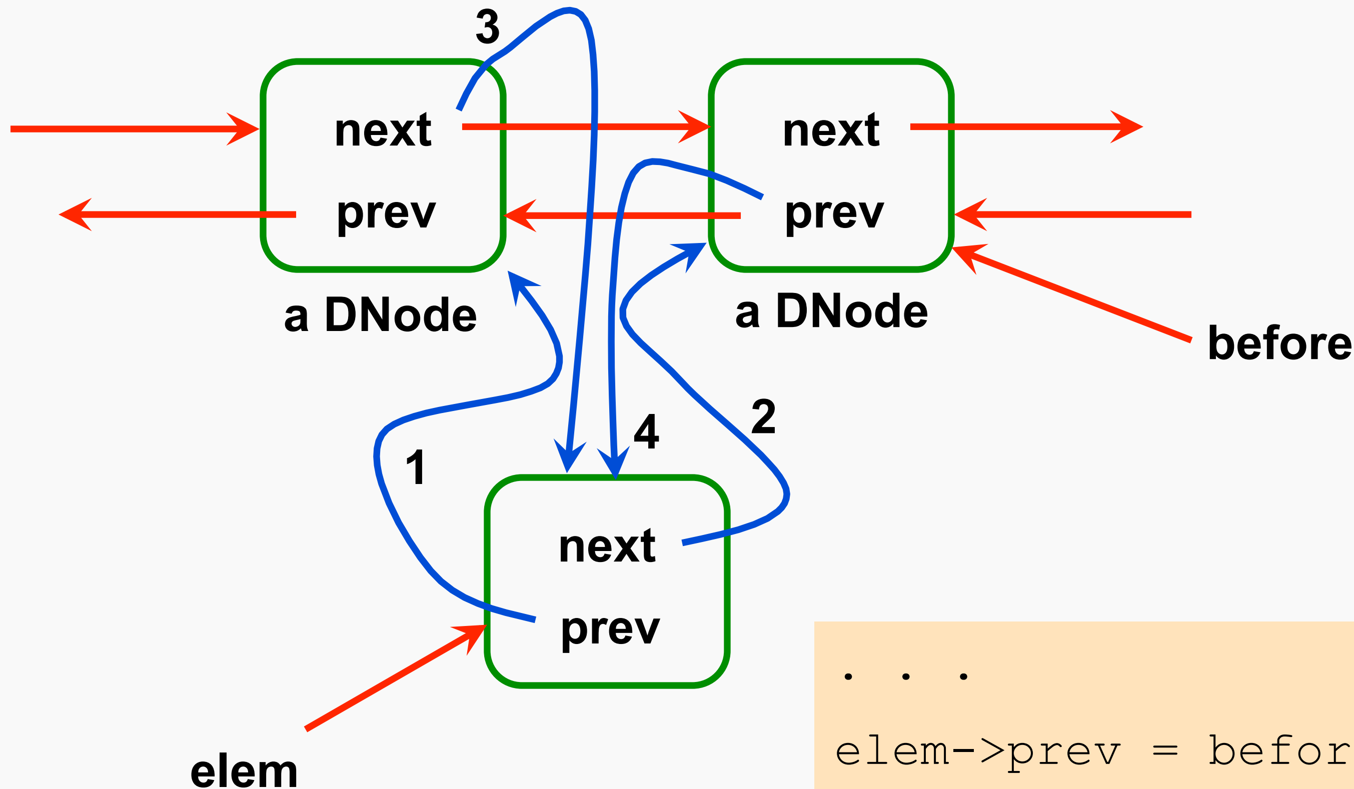
Each `DNode` object only "knows about" one or two other `DNode` objects.

The `DList` and `Dnode` objects "know" nothing of `IntegerDT` objects.

©2005-2015 McQuain

Inserting a DNode

We want to insert the node on the bottom between the other two nodes:




```
. . .
elem->prev = before->prev;  // 1
elem->next = before;        // 2
before->prev->next = elem;  // 3
before->prev = elem;        // 4
```


©2005-2015 McQuain

Inserting a DNode

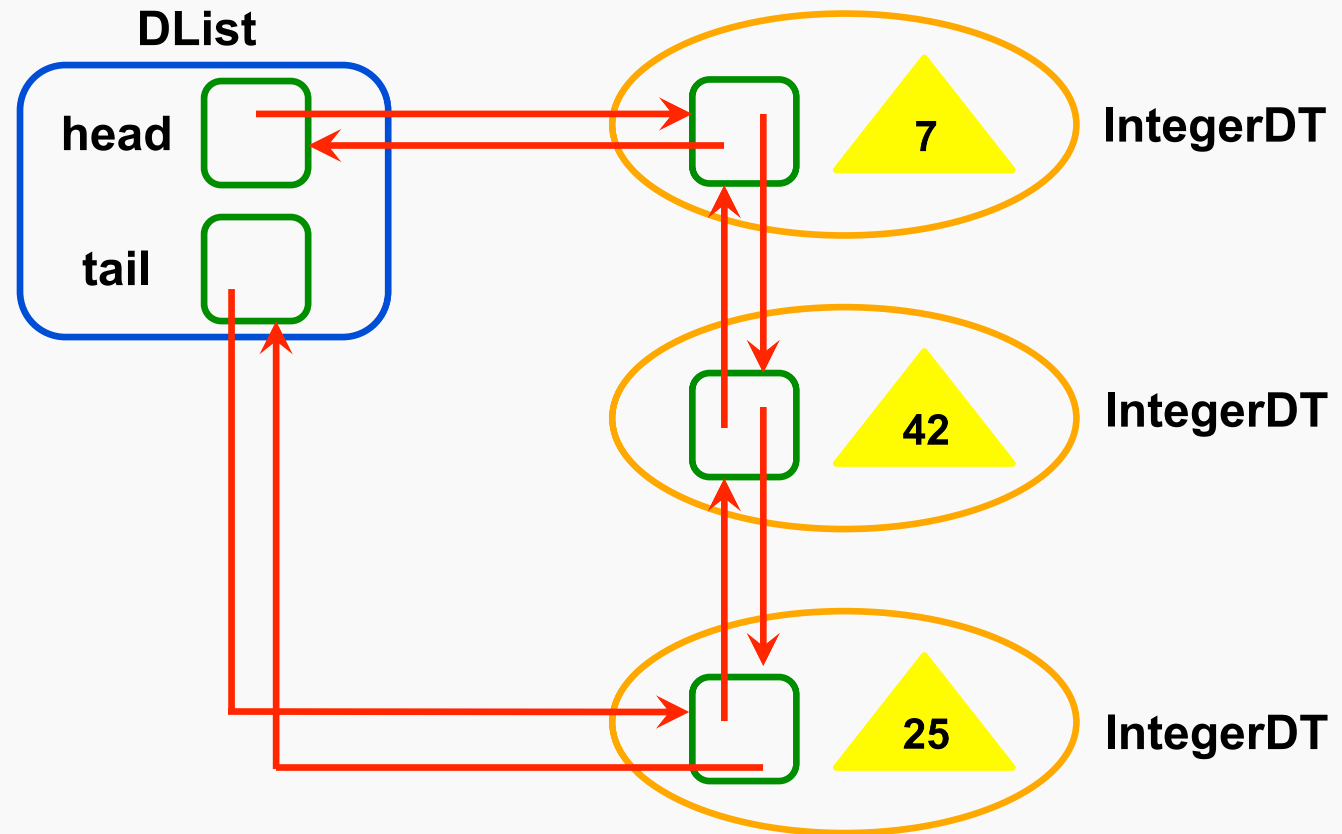
The `DList` only "knows about" two `DNode` objects.

```
/* Inserts elem as the predecessor of before, which may be
   either an interior element or a tail.
*/
void DList_Insert (DNode* const before, DNode* const elem)
{

  assert (is_interior (before) || is_tail (before));
  assert (elem != NULL);

  elem->prev = before->prev;
  elem->next = before;
  before->prev->next = elem;
  before->prev = elem;
}
```


©2005-2015 McQuain

Searching

Clearly, we need to be able to search a list for a data value that matches some search criterion.



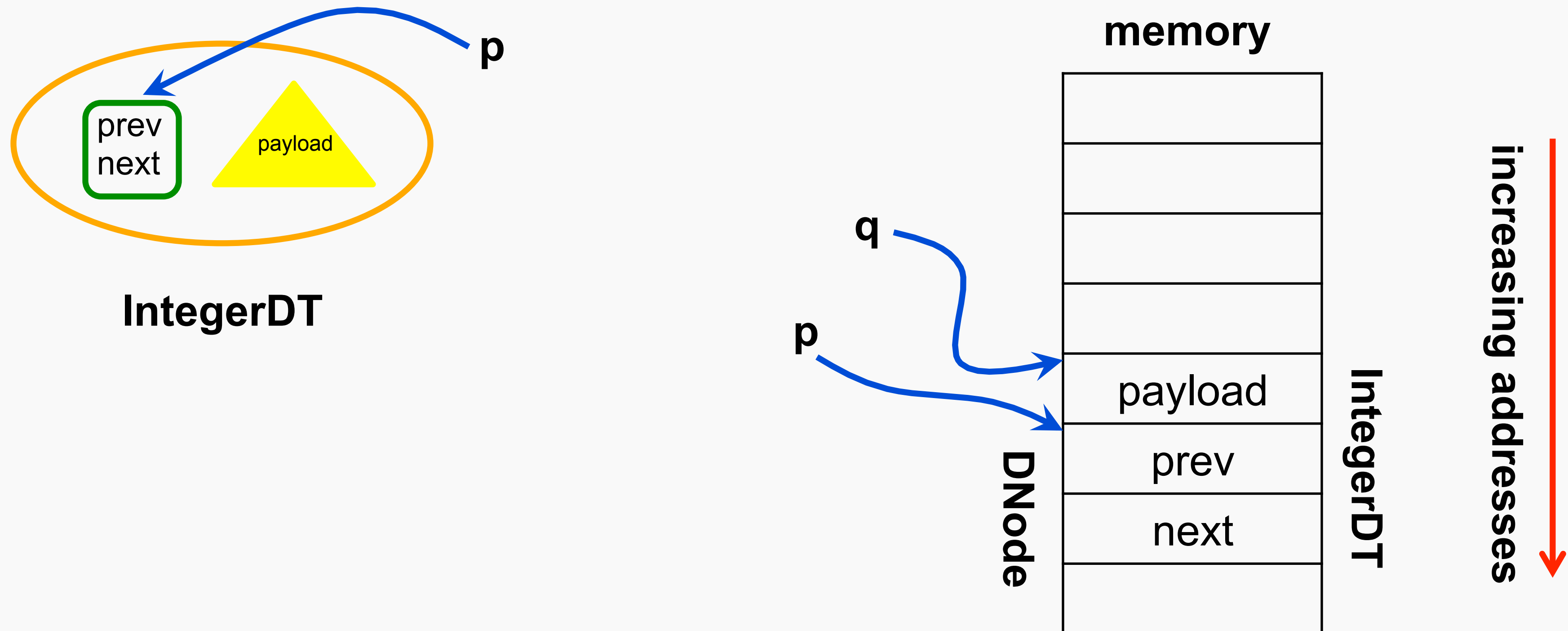
But we must follow the list pointers, which tie the `DNode` objects together…

… so how are we going to access the user data objects?

©2005-2015 McQuain

Accessing the "duct tape"

Suppose the IntegerDT object is laid out in memory as shown:



We want a pointer q that points to the IntegerDT object that contains the Dnode that p points to.

Then it appears we can set the value for q by subtracting 4 from p…
… but that logic depends on the specific memory layout shown above.

 ©2005-2015 McQuain

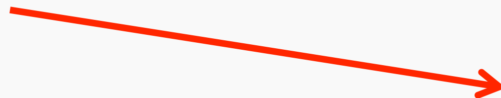
offsetof() to the Rescue!

The Standard Library includes a relevant C macro:

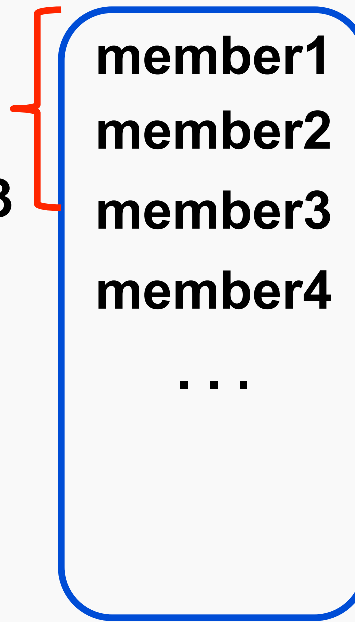
`offsetof(`*`type, member-designator`***`)`**

expands to an integer constant expression that has type `size_t`, the value of which is the offset in bytes, to the structure member (designated by *member-designator*), from the beginning of its structure (designated by *type*).

`offsetof(F, member3)`



offset of member3

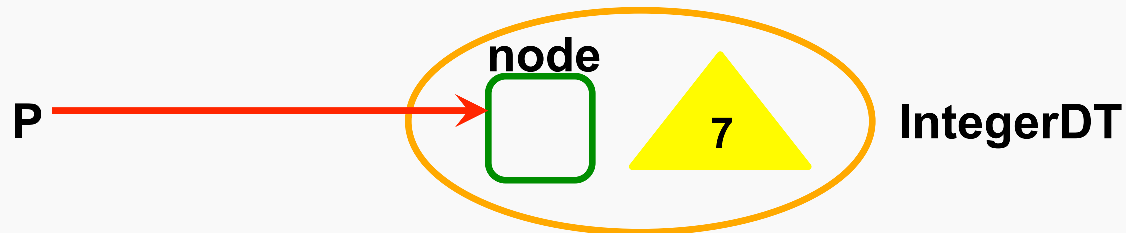



```
struct F {
   member1;
   member2;
   member3;
   member4;
};
```


©2005-2015 McQuain

So…

Let's say that we have a pointer `P` to a `DNode`, which is embedded within one of the `IntegerDT` objects seen earlier, and is also part of a `DList`.



Then, the address of the `IntegerDT` object would (almost) be given by:

```
P - offsetof(IntegerDT, node)
```

We just need to throw in a couple of typecasts:

```
(IntegerDT*) ( (uint8_t*)(P) - offsetof(IntegerDT, node) )
```


©2005-2015 McQuain

DList_Entry()

This is just begging to be turned into a C preprocessor macro:

```
/* Converts pointer to a DNode NODE into a pointer to the
   structure that DNode is embedded inside.

   Supply the name of the outer structure STRUCT and the
   member name MEMBER of the DNode.
*/

#define DList_Entry(NODE, STRUCT, MEMBER)      \
        ((STRUCT *) ((uint8_t *) (NODE) -      \
                           offsetof (STRUCT, MEMBER)))
```


©2005-2015 McQuain

Aside: Macro Translation

When the preprocessor sees code whose pattern matches the macro "interface", it replaces that code with code generated from the macro "body":

```
. . .
IntegerDT *p = DList_Entry(e, IntegerDT, node);
. . .
```

```
#define DList_Entry(NODE, STRUCT, MEMBER)       \
        ((STRUCT *) ((uint8_t *) (NODE) -       \
                      offsetof (STRUCT, MEMBER)))
```

```
. . .
IntegerDT *p = ((IntegerDT*)
          ((uint8_t*) (e - offsetof(IntegerDT, node)) );
. . .
```


©2005-2015 McQuain

Traversing the DList

```
void traverseList(DList* pL) {

   DNode* e = DList_Head(pL);

   while ( (e = DList_Next(e)) != DList_End(pL)) {

      // Get pointer to the "duct-tape" object from
      //    the pointer to the DList element:
      IntegerDT *p = DList_Entry(e, IntegerDT, node);

      // Get value of payload within "duct-tape" object:
      int userData = p->payload;

      // do stuff with current user data element
   }
}
```

©2005-2015 McQuain

More DList Functions

Here are some ideas for `DList` interface functions:

```
// Set up an empty list:
void DList_Init(DList* pList);

// Insert node elem in front of node before:
void DList_Insert(DNode* pBefore, DNode* pElem);

// Remove node elem:
DNode* DList_Remove(DNode* pElem);

// Is list empty?
bool DList_Empty(DList* pList);

// Restore list to empty state:
void Dlist_Clear(Dlist* pList);
. . .
```


©2005-2015 McQuain

More DList Functions

```
. . .
// Get pointer to first/last data node in list:
DNode* DList_Begin(DList* pList);
DNode* DList_End(DList* pList);

// Get pointer to successor/predecessor of node:
DNode* DList_Next(DNode* pElem);
DNode* DList_Prev(DNode* pElem);

// Get pointer to head/tail of list:
DNode* DList_Head(DList* pList);
DNode* DList_Tail(DList* pList);
. . .
```

 ©2005-2015 McQuain

More DList Functions

```
. . .
// Insert elem at front/rear of list:
void DList_PushFront(DList* pList, DNode* pElem);
void DList_PushBack(DList* pList, DNode* pElem);

// Remove elem from front/rear of list:
DNode* DList_PopFront(DList* pList);
DNode* DList_PopBack(DList* pList);
```


©2005-2015 McQuain