void ArrayTest() {

int scores[100];

// operate on the elements of the scores array...

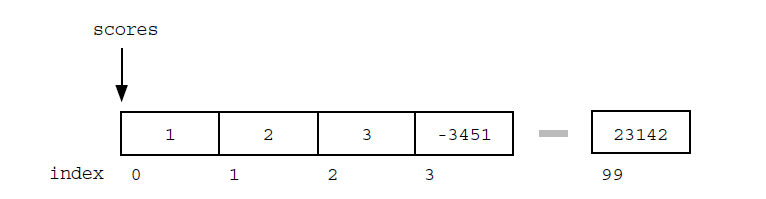
scores[0] = 1;

scores[1] = 2;

scores[2] = 3;

}

The key point is that the entire array is allocated as one block of memory.



Once the array is set up, access to any element is convenient and fast with the [ ] operator. Array access with expressions such as scores[i] is almost always implemented using fast address arithmetic: the address of an element is computed as an offset from the start of the array which only requires one multiplication and one addition.

Some disadvantages of arrays:

* The size of the array is fixed. Most often this size is specified at compile time with a simple declaration such as in the example above . With a little extra effort, the size of the array can be deferred until the array is created at runtime, but after that it remains fixed. You can go to the trouble of dynamically allocating an array in the heap and then dynamically resizing it with realloc(), but that requires more effort.
* Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

**Linked Lists**

An array allocates memory for all its elements lumped together as one block of memory. In contrast, a linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets is overall structure by using pointers to connect all its nodes together like the links in a chain. Think of it like a train. The programmer always stores the first node of the list. This would be the engine of the train. The pointer is the connector between cars of the train. Every time the train adds a car, it uses the connectors to add a new car. This is like a programmer using the keyword new to create a pointer to a new struct or class. 

Each of the big blocks is a struct (or class) that has a pointer to another one. Remember that the pointer only stores the memory location of something, it is not that thing, so the arrow goes to the next one. At the end, there is nothing for the pointer to point to, so it does not point to anything, it should be a null pointer or a dummy node to prevent it from accidentally pointing to a totally arbitrary and random location in memory (which is very bad). 

Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field, which is a pointer used to link one node to the next node.

Each node is allocated on the heap with a either malloc( ) or new. The node memory continues to exist until it is explicitly de-allocated using free( ) or delete.

A linked list is a dynamic data structures - grow and shrink during execution

**Node and Pointer (Self-referential)**

*• Node* The type for the nodes which will make up the body of the list.

These are allocated on the heap. Each node contains a single client data element and a pointer to the next node in the list.

struct node {

int data;

struct node\* next;

};

struct node \*ptr

//or

struct node {

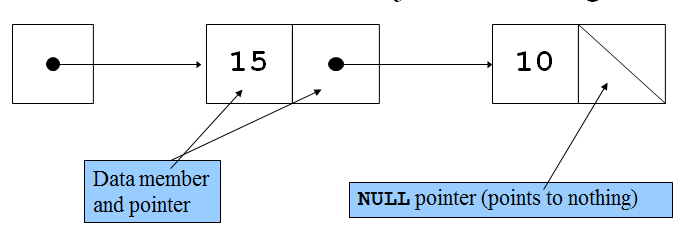
int data;

struct node\* next;

};

typedef struct node node;

node \*ptr;

*• Node Pointer* The type for pointers to nodes. 

* A linked list is a dynamic data structure
* The address of the first node is called the head or first. The address of the last node is called the tail.
* The first (or head) pointer of a linked list is always fixed, pointing to the first node in the list.
* **Linear** **relationship** Each element except the first has a unique predecessor, and each element except the last has a unique successor.
* To traverse a linked list, the program must use a pointer different than the head pointer of the list, initialized to the first node in the list.
* A single linked list is traversed in only one direction
* The search on a linked list is sequential
* The length of a linked list is the number of nodes in the list
* Four common operations associated with lists: Insertion, deletion, retrieval and traversal
* Item insertion and deletion from a linked list do not require data movement; only the pointers are adjusted.

**Example 1 (Creating a single node)**

struct node {

int data;

node \*next;

};

int main()

{

node \*head; //This will be the unchanging first node

head = new node; //Now head points to a node struct

head->next = NULL; //The node head points to has its next pointer

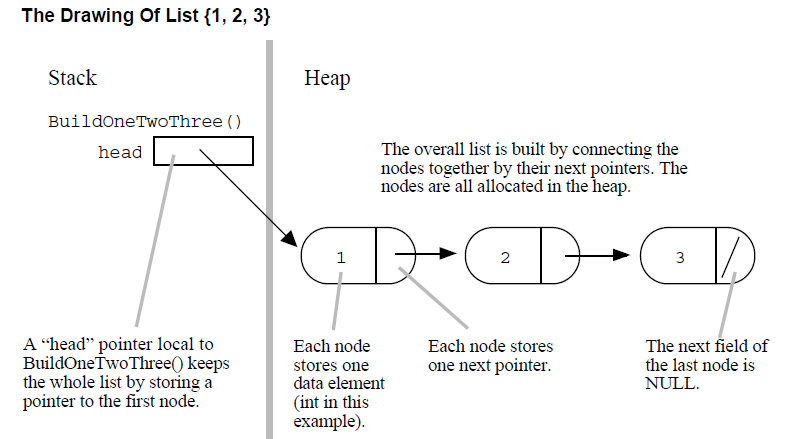
//set equal to a null pointer

head->data = 5; //By using the -> operator, you can modify the node

//a pointer (head in this case) points to.

}

**Example 2 (Creating multiple nodes)**



The beginning of the linked list is stored in a"head" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its .next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the head and following the .next pointers. Operations towards the front of the list are fast while operations which access node farther down the list take longer the further they are from the front. This "linear" cost to access a node is fundamentally more costly then the constant time [ ] access provided by arrays. In this

respect, linked lists are definitely less efficient than arrays.

When working on linked list code, it's a good habit to remember to check the empty list case to verify that it works too. Sometimes the empty list case works the same as all the

cases, but sometimes, it requires some special case code.

/\*

Build the list {1, 2, 3} in the heap.

Returns the first pointer to the caller.

\*/

struct node {

int data;

node \*next;

};

struct node\* build() {

struct node\* first = NULL;

struct node\* second = NULL;

struct node\* third = NULL;

// allocate 3 nodes in the heap

first = (node \*)malloc(sizeof(struct node));

second = (node \*)malloc(sizeof(struct node));

third = (node \*)malloc(sizeof(struct node));

first->data = 1; // setup first node

first->next = second; // note: pointer assignment rule

second->data = 2; // setup second node

second->next = third;

third->data = 3; // setup third link

third->next = NULL;

return first ;

}

int main()

{

node \*head=build();

}

**Example 3 (Adding a node to the beginning)**

General mechanism to build lists. The best solution will be an independent function which

adds a single new node to any list. We can then call that function as many times as we

want to build up any list. The 3 steps are...

*1) Allocate* Allocate the new node in the heap and set its .data to whatever needs to be stored.

struct node\* newNode;

newNode = (node \*)malloc(sizeof(struct node));

newNode->data = data\_client\_wants\_stored;

*2) Link Next* Set the .next pointer of the new node to point to the current first node of the list.

newNode->next = head;

*3) Link Head* Change the head pointer to point to the new node, so it is now the first node in the list.

head = newNode;

struct node {

int data;

node \*next;

};

struct node\* build() {

struct node\* head;

head = (node \*)malloc(sizeof(struct node)); // allocate on the heap

head->data = 2; // setup first node

head->next = (node \*)malloc(sizeof(struct node));

head->next->data = 3; // setup second node

head->next->next = NULL;

return (head);

}

void main() {

struct node\* head = build ();

struct node\* newNode;

newNode= (node \*)malloc(sizeof(struct node)); // allocate

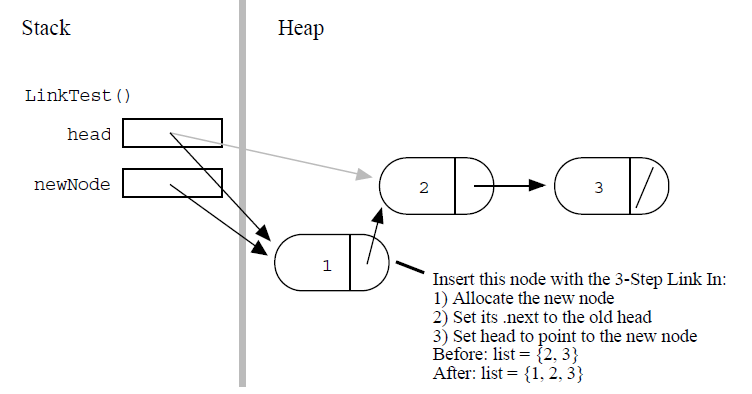
newNode->data = 1;

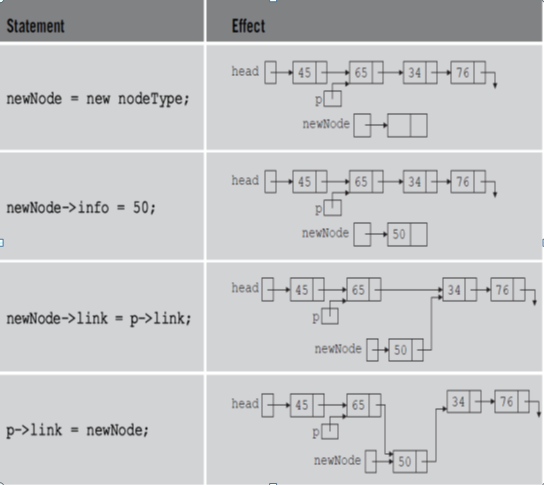
newNode->next = head; // link next

head = newNode; // link head

// now head points to the list {1, 2, 3}

}





**Example 4 (Traversing and Finding the number of elements in a linked list)**

/\*

Given a linked list head pointer, compute and return the number of nodes in the list. BuildOneTwoThree() creates a linked list with 3 elements. Code is not included

\*/

int Length(struct node\* head) {

struct node\* current = head;

int count = 0;

while (current != NULL) {

count++;

cout<< current ->data;

current = current->next;

}

return count;

}

Int main() {

struct node\* myList = BuildOneTwoThree();

int len = Length(myList); // results in len == 3

}

1) The local pointer, current in this case, starts by pointing to the same node as the head pointer with current = head;. When the function exits, current is automatically deallocated since it is just an ordinary local, but the nodes in the heap remain.

2) The while loop tests for the end of the list with (current != NULL). This test smoothly catches the empty list case — current will be NULL on the first iteration and the while loop will just exit before the first iteration.

3) At the bottom of the while loop, current = current->next; advances the local pointer to the next node in the list. When there are no more links, this sets the pointer to NULL. If you have some linked list code which goes into an infinite loop, often the problem is that step (3) has been forgotten.

Some questions to consider:

Q: What if at the end of Length() we said head = NULL

Would that mess up the myList variable in the caller?

Q: What if what was passed in contained no elements, does Length() handle that case properly?

**Example 5 (Error with linked list: local variable)**

The key is that the line head = newNode; changes the head local in wrongAdd() but not

the head back in main().

struct node {

int data;

node \*next;

};

struct node\* build() {

struct node\* head;

head = malloc(sizeof(struct node)); // allocate on the heap

head->data = 2; // setup first node

head->next = malloc(sizeof(struct node)); // allocate on the heap

head->next->data = 3; // setup second node

head->next->next = NULL;

return (head);

**}**

//The change to head is not passed back

void wrongAdd (struct node\* head, int data) {

struct node\* newNode = (node \*)malloc(sizeof(struct node));

newNode->data = data;

newNode->next = head;

head = newNode; // NO this line does not work!

}

int main() {

struct node\* head = build();

wrongAdd(head, 1); // try to add 1 to the front -- doesn't work

cout<<head->data;

}

**Example 6 (Corrected version of pervious example)**

//same as before

struct node {

int data;

node \*next;

};

//same as before

struct node\* build() {

struct node\* head;

head = malloc(sizeof(struct node)); // allocate on the heap

head->data = 2; // setup first node

head->next = malloc(sizeof(struct node)); // allocate on the heap

head->next->data = 3; // setup second node

head->next->next = NULL;

return (head);

**}**

//modified: Returns head

struct node\* rightAdd (struct node\* head, int data) {

struct node\* newNode = (node \*)malloc(sizeof(struct node));

newNode->data = data;

newNode->next = head;

head = newNode;

return head

}

//modified: catches head after rightAdd call

int main() {

struct node\* head = build();

head= rightAdd (head, 1);

cout<<head->data;

}

**Example 7 (Correcting the previous example again using \*\*)**

• Design the function to take a pointer to the head pointer. This is the standard technique in C — pass a pointer to the "value of interest" that needs to be changed. To change a struct node\*, pass a struct node\*\*.

• Use '&' in the caller to compute and pass a pointer to the value of interest.

• Use '\*' on the parameter in the callee function to access and change the value of interest.

/\*

Takes a list and a data value. Creates a new link with the given data and pushes it onto the front of the list. The list is not passed in by its head pointer. Instead the list is passed in as a "reference" pointer to the head pointer -- this allows us to modify the caller's memory.

\*/

struct node {

int data;

node \*next;

};

struct node\* build() {

struct node\* head;

head =(node \*) malloc(sizeof(struct node)); // allocate on the heap

head->data = 2; // setup first node

head->next = (node \*)malloc(sizeof(struct node));

head->next->data = 3; // setup second node

head->next->next = NULL;

return (head);

**}**

void add(struct node\*\* headRef, int data) {

struct node\* newNode = (node \*)malloc(sizeof(struct node));

newNode->data = data;

newNode->next = \*headRef;

\*headRef = newNode;

}

void main() {

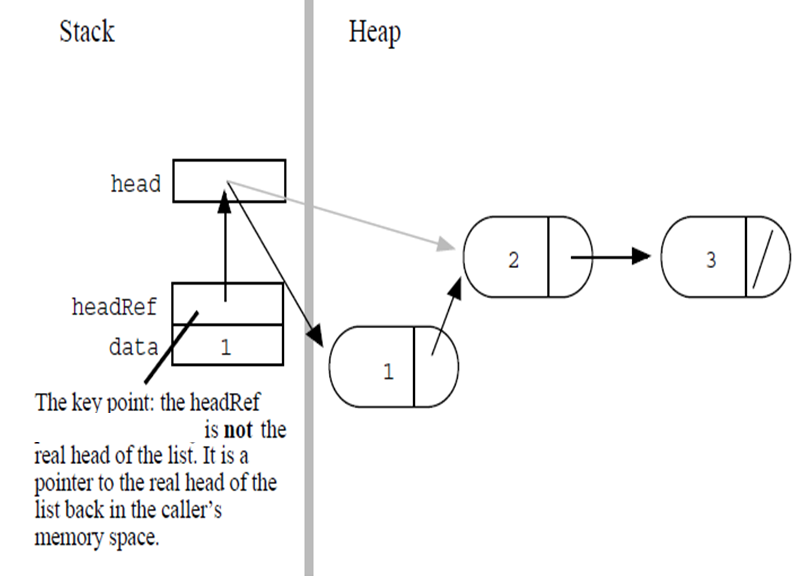
struct node\* head = build();

add (&head, 1); // note the &

add (&head, 13);

// head is now the list {13, 1, 2, 3}

}



**Example 8 (Adding to the end)**

Think back to the train. Let’s imagine a conductor who can only enter the train through the engine, and can walk through the train down the line as long as the connector connects to another car. This is how the program will traverse the linked list. The conductor will be a pointer to node, and it will first point to root, and then, if the root's pointer to the next node is pointing to something, the "conductor" (not a technical term) will be set to point to the next node. In this fashion, the list can be traversed. Now, as long as there is a pointer to something, the traversal will continue. Once it reaches a null pointer, meaning there are no more nodes (train cars) then it will be at the end of the list, and a new node can subsequently be added if so desired.

Adding a node at the tail of a list most often involves locating the last node in the list, and then changing its next field from NULL to point to the new node, such as the tail variable.

struct node {

int data;

node \*next;

};

struct node\* build() {

struct node\* head;

head = (node \*)malloc(sizeof(struct node)); // allocate on the heap

head->data = 2;

head->next = (node \*)malloc(sizeof(struct node));

head->next->data = 3; // setup second node

head->next->next = NULL;

return (head);

}

void addToEnd (struct node \* current, int data){

struct node\* newNode;

while ( current->next != NULL)

current = current->next;

newNode = new node; // Sets it to actually point to something

newNode->data = data;

newNode->next = NULL;

current->next=newNode;

}

int main (){

struct node \* head=build();

if (head) //Make sure head is valid

addToEnd(head, 5);

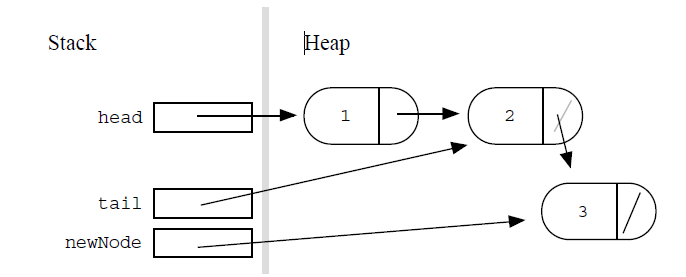
while ( head != NULL ) {

cout<< head->data;

head=head->next;

}

}



**Example 9 (without dummy node: Adding to the end)**

struct node {

int data;

node \*next;

};

typedef struct node node;

node\* add(node\*& end, int data) {

node\* newNode = (node \*)malloc(sizeof(struct node));

newNode->data = data;

newNode->next = NULL;

end = newNode;

return (end);

}

node\* buildWithOutDummy () {

node\* head, \*current;

head=current=NULL;

for (int i=1; i<4; i++) {

if (!head)

current=add(head, i);

else

current=add(current->next,i);

}

return(head);

}

int main(){

node\* head, \*current;

current=head==buildWithOutDummy();

while (current!= NULL ) {

cout<< current ->data;

current = current ->next;

}

}

**Example 10 (Building a linked list with a dummy node: Adding to the end)**

Another solution is to use a temporary dummy node at the head of the list during the

computation. The trick is that with the dummy, every node appear to be added after the

.next field of a node. That way the code for the first node is the same as for the other

nodes. The tail pointer plays the same role as in the previous example. The difference is

that it now also handles the first node.

struct node {

int data;

node \*next;

};

struct node\* add(struct node\*& tailNext, int data) {

struct node\* newNode = (node \*)malloc(sizeof(struct node));

newNode->data = data;

newNode->next = NULL;

tailNext = newNode;

return tailNext;

}

//Note how dummy is declared

struct node\* buildWithDummy () {

struct node dummy; // Dummy node is temporarily the first node

dummy.next = NULL;

struct node\* tail = &dummy; // Start the tail at the dummy.

// Build the list on dummy.next (aka tail->next)

for (int i=1; i<4; i++) {

//the content of tail->next is always NULL. But since we are

//passing by reference, we care about what will be placed in

//the location once inside the function.

//At first tail->next is going to point to dummy.next. In

//the next iterations it will contain the address of other nodes

//as created in the add function.

tail=add(tail->next, i);

}

// The address for the beginning of the list is here

return(dummy.next);

}

int main(){

struct node \*current, \*head;

current=head =buildWithDummy();

while ( current != NULL ) {

cout<< current->data;

current=current->next;

}

}

**Example 11 (Building a linked list with a dummy node without pass by alias)**

struct node {

int data;

node \*next;

};

struct node\* add(struct node\*\* tailNext, int data) {

struct node\* newNode = (node \*)malloc(sizeof(struct node));

newNode->data = data;

newNode->next = NULL;

\*tailNext = newNode;

return \*tailNext;

}

//Note how dummy is declared

struct node\* buildWithDummy () {

struct node dummy; // Dummy node is temporarily the first node

dummy.next = NULL;

struct node\* tail = &dummy; // Start the tail at the dummy.

// Build the list on dummy.next (aka tail->next)

for (int i=1; i<4; i++) {

//the content of tail->next is always NULL. But since we are

//passing by reference, we care about what will be placed in

//the location once inside the function.

//At first tail->next is going to point to dummy.next. In

//the next iterations it will contain the address of other nodes

//as created in the add function.

tail=add(&tail->next, i);

}

// The address for the beginning of the list is here

return(dummy.next);

}

int main(){

struct node \*current, \*head;

current=head=buildWithDummy();

while ( current != NULL ) {

cout<<current->data;

current=current->next;

}

}

Some linked list implementations keep the dummy node as a permanent part of the list.

For this "permanent dummy" strategy, the empty list is not represented by a NULL

pointer. Instead, every list has a dummy node at its head. Algorithms skip over the

dummy node for all operations.

**Example 12 (Making a copy of a linked list)**

Consider a CopyList() function that takes a list and returns a complete copy of that list.

One pointer can iterate over the original list in the usual way. Two other pointers can

keep track of the new list: one head pointer, and one tail pointer which always points to

the last node in the new list. The first node is done as a special case, and then the tail

pointer is used in the standard way for the others...

struct node\* copyList(struct node\* head) {

struct node\* current = head; // used to iterate over the original list

struct node\* newList = NULL; // head of the new list

struct node\* tail = NULL;

while (current != NULL) {

if (!newListHead) {

newList =(node \*) malloc(sizeof(struct node));

newList ->data = current->data;

newList ->next = NULL;

tail = newList;

} else {

tail->next = (node \*)malloc(sizeof(struct node));

tail=tail->next

tail->data = current->data;

tail->next = NULL;

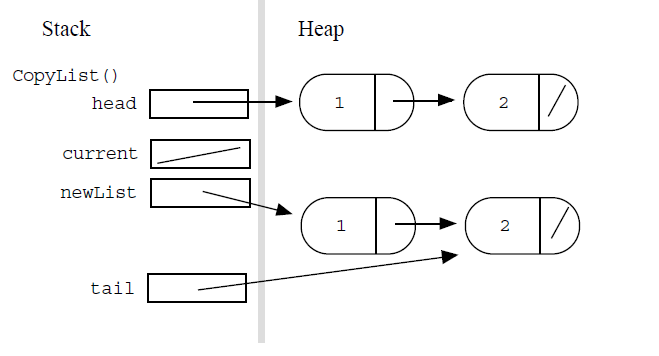
}

current = current->next;

}

return(newList);

}



**Example 13 (Another version of making a copy of a linked list)**

The above implementation is a little unsatisfying because the 3-step-link-in is repeated —

once for the first node and once for all the other nodes. Write a CopyList2() which uses

add() to take care of allocating and inserting the new nodes, and so avoids repeating that

code.

void add(struct node\*& current, int data) {

current = (node \*)malloc(sizeof(struct node));

current ->data = data;

current ->next = NULL

}

// Variant of CopyList() that uses add()

struct node\* copyList(struct node\* head) {

struct node\* current = head; // used to iterate over the original list

struct node\* newList = NULL; // head of the new list

struct node\* tail = NULL;

while (current != NULL) {

if (!newList) {

add(newList, current->data);

tail = newList;

} else {

add(tail->next, current->data);

tail=tail->next;

}

current = current->next;

}

return(newList);

}

**Additional NOTES**

Watch for the following when deleting:

1. An attempt to remove a node from an empty list
2. Deleting the only node from a one node linked list. Both head and tail are set to null
3. Removing the first node requires updating the head.
4. Removing the last node requires updating of the tail.

When deleting, consider, deleting the first node and also any other node.

There are a many variations on the basic linked list which have individual advantages

over the basic linked list.

*• Dynamic Array* Instead of using a linked list, elements may be stored in an array block allocated in the heap. It is possible to grow and shrink the size of the block as needed with calls to the system function realloc(). Managing a heap block in this way is a fairly complex, but can

have excellent efficiency for storage and iteration., especially because modern memory systems are tuned for the access of contiguous areas of memory. In contrast, linked list can actually be a little inefficient, since they tend to iterate through memory areas that are not adjacent.

*• Dummy Header* Forbid the case where the head pointer is NULL. Instead, choose as a representation of the empty list a single "dummy" node whose .data field is unused. The advantage of this technique is that the pointer-to-pointer (reference parameter) case does not come up for operations such as Push(). Also, some of the iterations are now a little simpler since they can always assume the existence of the dummy header node. The disadvantage is that allocating an "empty" list now requires allocating (and wasting) memory. Some of the algorithms have an ugliness to them since they have to realize that the dummy node "doesn't count."

Mainly the dummy header is for programmers to avoid the ugly reference parameter issues in functions such as Push().

*• Tail Pointer* The list is not represented by a single head pointer. Instead the list is represented by a head pointer which points to the first node and a tail pointer which points to the last node. The tail pointer allows operations at the end of the list such as adding an end element or appending two lists to work efficiently.

• *Head struct* A variant better than the dummy header is to have a special "header" struct (a different type from the node type) which contains a head pointer, a tail pointer, and possibly a length to make many operations more efficient. Many of the reference parameter problems go

away since most functions can deal with pointers to the head struct (whether it is heap allocated or not).

**Example 14: ERROR -- Removing from a list**

void deleteNode(int item, node \*&head){

node \*temp, \*curr=head;

//Make sure that there is list

if (head!=NULL){

//if the first item is a match

if ( (head->a==item){

temp=head; //get ready to take out the head

head=head->next; //head is updated

}else{

curr=curr->next;

//Start looking from the second item forward

//Make sure that there is a second item

//Move curr as long as it is not found

while (curr!=NULL && curr->a!=item)

//Notice that we are always looking ahead of the

//link that we are in

curr=curr->next;

//If item has been found then curr should

//not be null because the found item is ahead of

//where curr is

if (curr!=NULL){

//curr of the place that needs to be deleted

temp=curr;

//???????????????????????????????????????????

//error: How do we connect the previous node to

//one beyond the node that we want to delete

//??? previous curr->next= curr->next; ???

}

}

//Only if item was found should we delete

if(temp!=NULL)

delete temp;

}

}

**Example 15: Correcting previous version -- Removing from a list**

void deleteNode(int item, node \*&head){

node \*temp, \*curr=head;

//Make sure that there is list

if (head!=NULL){

//if the first item is a match

if ( (head->a==item){

temp=head; //get ready to take out the head

head=head->next; //head is updated

}else{

//Start looking from the second item forward

//Make sure that there is a second item

//Move curr as long as it is not found

while (curr->next!=NULL && curr->next->a!=item)

//Notice that we are always looking ahead of the

//link that we are in

curr=curr->next;

//If item has been found then curr->next should

//not be null because the found item is ahead of

//where curr is

if (curr->next!=NULL){

//curr of the place that needs to be deleted

temp=curr->next;

//curr is where we are pointing to

//curr->next contains address of node to be deleted

//curr->next needs to skip to curr->next->next

curr->next=curr->next->next;

}

}

//Only if item was found should we delete

if(temp!=NULL)

delete temp;

}

}

**Example 17: Removing from the list with a temp node**

struct node \* deleteAlternate(int item, node \*head){

node \*temp, \*curr=new node;

curr->next=head;

//Start looking from the second item forward

//Make sure that there is a second item

//Move curr as long as it is not found

while (curr->next!=NULL && curr->next->a!=item){

//Notice that we are always looking ahead of the

//link that we are in

cout<<curr->next->a;

curr=curr->next;

}

//If item has been found then curr next should

//not be null because the found item is ahead of

//where curr is

if (curr->next!=NULL){

//curr of the place that needs to be deleted

temp=curr->next;

if (temp==head)

head=curr->next->next;

else

curr->next=curr->next->next;

delete temp;

}

return (head) ;

}