COMSC-110 Chapter 14

Lists of unlimited size: Linked Lists

Valerie Colber, MBA, PMP, SCPM

Chapter 14

- This chapter is an opportunity to introduce concepts that you will run into if you go further in your study of computer science, and to provide an application of the programming features we studied in the preceding chapters.
- This chapter introduces Linked Lists.

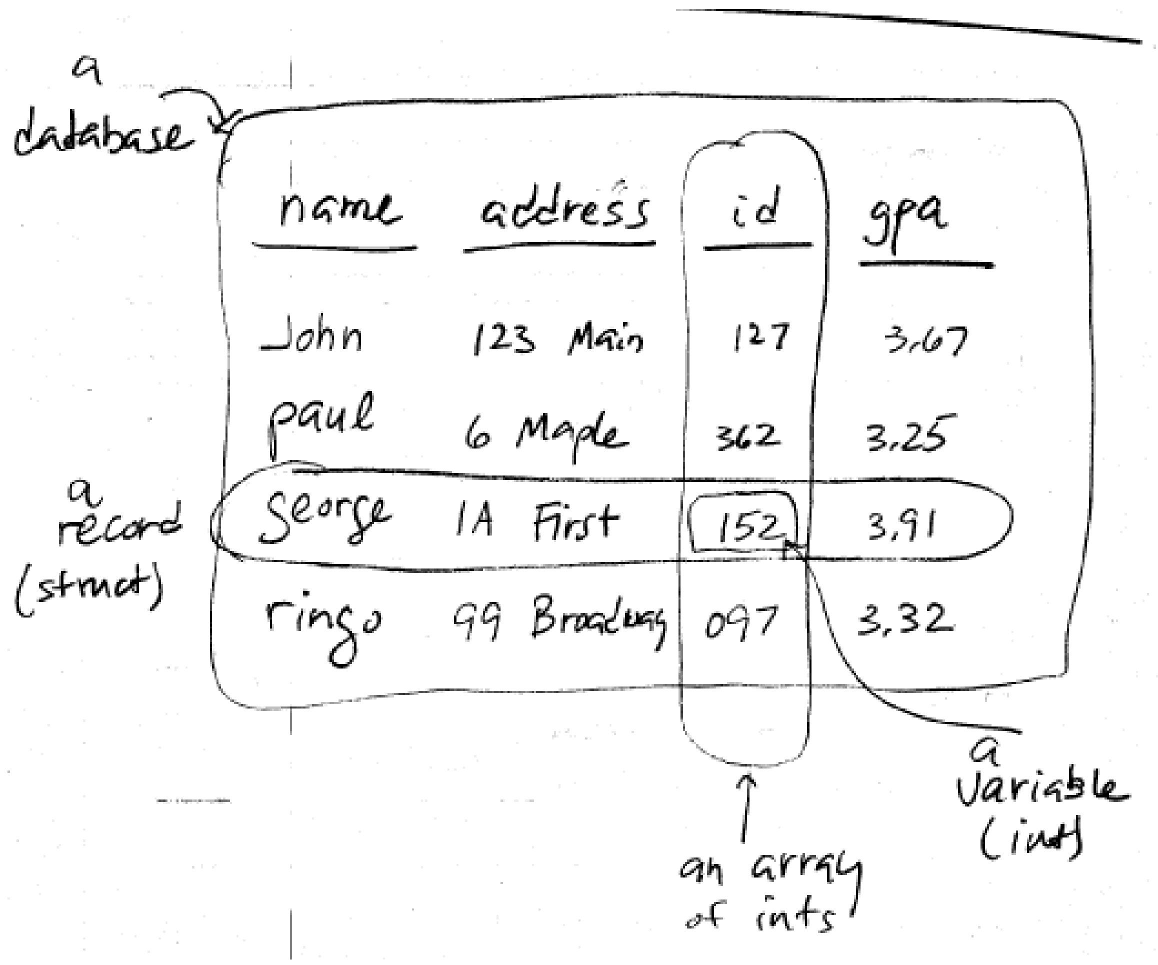
Disadvantages of Array-based Lists

- they have a limited capacity.
- if you do not use them to their capacity, you wasted memory.
- the code for inserting and deleting values at any place except for the end of the list is rather lengthy, and the computer processes they invoke are time-consuming.
- they are "*last* come, first served" -- what's known as a "stack" model.

Storing values in a program

- Literal values
- Variables: store one value
- Arrays: store multiple values of the *same data type*
- Structs: database records
- Array-Based Lists: arrays in which not all of the elements are used, but are available for use instead.
- Linked Lists: each value is separate, containing a "link" to the next value in the list.

Variables, arrays, records, and data bases



name	address	id	gpa
John	123 Main	127	3.67
paul	6 Maple	362	3.25
George	1A First	152	3.91
ringo	99 Broadway	097	3.32

List

- A *list* is a collection of variables -- usually struct variables, but they can be integers or floating points or text, too.
- The language allows the data types for these to be different from one another, even in the same collection!
- The concept is that of a "bag", which can contain lots of different items.
- *But for purposes of our study, we will consider collections of like data types only.*

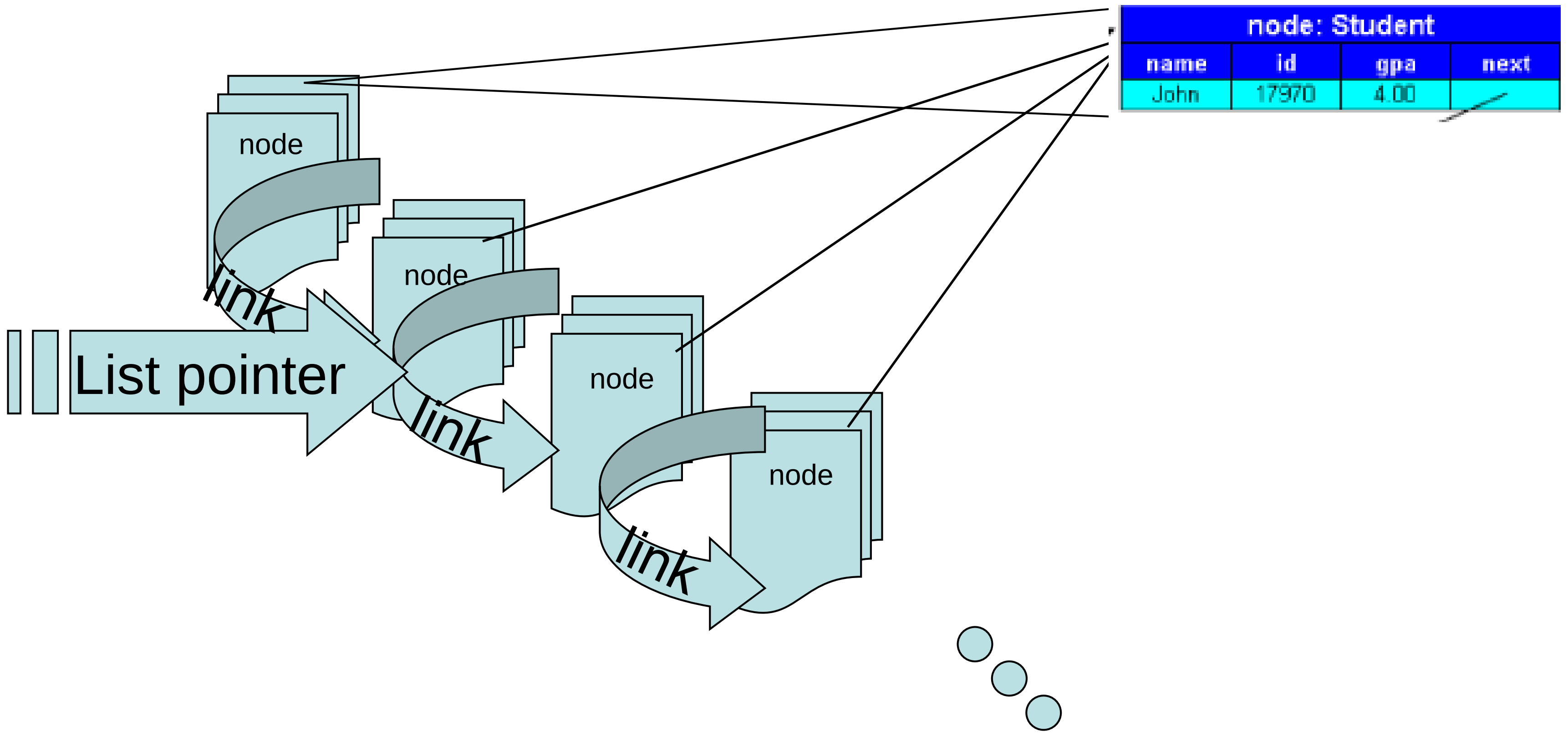
Lists

- A list can be empty -- that is, it can have zero values in it.
- Lists cans grow in size by adding values, and shrink by taking them away.
- With a list we could find lowest test score and delete it, or insert a score for a late assignment.
- By contrast, the arrays we studied were of *fixed*, nonzero size, and all elements had a value.
- So there are similarities and differences between arrays and lists.

Linked Lists

- Like arrays, but *no pre-declared storage* so no specified capacity.
- Capacity only limited by computer memory!
- Can even be empty, like array-based list.
- Can add to front *or* end of list.
- The topic of linked lists is worthy of months of study in a computer science class.

Visualizing a linked list



node: Student			
name	id	gpa	next
John	17970	4.00	

Linked lists use objects

- Linked lists have nodes
- Nodes are records using a struct definition except the last field is the field identifier next, that is the link to the memory address of the following node
- Linked lists use pointers to traverse the list
- Fields in a node use an arrow instead of a dot
- The last node in the list next field is 0, meaning it is pointing back to the 1st node in the list.

Advantages of Linked Lists

- Inserting values is simply a matter of breaking the link between two existing values, and creating links to the newly inserted value.
- Removing a value at any position in the link is even easier than inserting, by realigning the links of its adjacent values.

Disadvantages of Linked Lists

- One drawback of linked lists is that the require that the values be struct variables.
- If you want a list of integers or strings, you have to put the integer as a field in a struct, which may only have that one field.
- This may seem like a lot of overhead, but in most applications, linked lists are used with records anyway, so the struct already exists in these cases.

How to identify Array-based and Linked Lists

- Array-based lists are defined by two values: their container array and their size counter.
- Linked lists are identified by one value: a "link" to the first value in the list.

Linked List Struct Definitions

- The values contained in linked lists are struct variables.
- In order for a struct to be suitable for use in a linked list, it needs to contain a "link".
- The link is normally the *last* field in the struct definition, and it is usually named "next".

A struct Definition For Use In Linked Lists

```
struct Student
{
  string name;
  int id;
  float gpa;
  Student* next; // the "link"
}; // Student
```

each element in list "knows" the *next* element that follows

A struct Definition For Use In Linked Lists

- Note the ***** in the link field: **Student* next;**.
- It distinguishes between an actual record (without an asterisk, or "star"), and a link to another record (with).
- It is the same symbol we saw earlier in dynamically-sized arrays and arrays in function parameter lists.
- Variables declared with a struct containing a link are called *nodes*.

The Start pointer

- The identifier for a linked list is its *start pointer*.
- Here is how to declare an empty linked list:

  Student* start = 0;

- By convention, the identifier for a linked list is “start".
- But if you have more than one list in the same scope, they obviously cannot have the same identifier, so other names can of course be used.
- The star symbol makes **start** into a *pointer* -- a variable that stores the memory location of some other variable.

The Start pointer

- The start pointer "knows" where to find the first node in the list that it identifies.
- A value of zero indicates that the start pointer has no valid memory location stored in it, indicating that the list is *empty*.
- While it is not required, and some programmers and textbooks do not do so, linked lists should *always* start out empty.
- That's what the **0** does in the declaration statement for **start**.

Building a linked list

- Nodes can be added anywhere in the list -- at the front, at the end, or any place in between.
- In this course we will show how to add nodes to the front or end of a list, leaving other linked list features for more advanced courses.
- Nodes can be removed from the front of a list, so we will have two models:

1. a "first come, last served" *stack*, by adding nodes to the front.
2. a "first come, first served" *queue*, by adding nodes to the end.

- These should be sufficient for many simulation applications, including gaming.

List “nodes”

- Like an array "element", but contains a "next" link.
- *Not* referenced individually, as are elements.
- There is *no "index"*, as with arrays.
- Only processed via traversal.

Adding Nodes To The Front Of A Linked List -- The Stack Model

- Adding nodes to the front of a linked list simulates "stacks", which are based on "last in, first out", or LIFO.

```
// create a node
Student* s = new Student;
s->name =...; // fill its data fields
 ...
// add node to list (stack model)
s->next = start;
start = s;
```

Adding Nodes To The Front Of A Linked List -- The Stack Model

- The last two lines of code accomplish the linking of the node into the list.

Student* s = new Student;

- it's like an array declaration, but there are no square brackets and no size specification.
- 2-step process: create a *temporary* node and enter its data then *add* node.

Adding Nodes To The Front Of A Linked List -- The Stack Model

- So why didn't we create the node the same way we create records, like this?

Student s;

- It has to do with how memory works in C++, and the fact that records are temporary, while nodes are more permanent.
- You could actually code this with a record instead of a node, but it would lead to very nasty memory errors when the program runs.

Adding Nodes To The Front Of A Linked List -- The Stack Model

- We know **s.name** and **s.next**, but what's **s->name** and **s->next**?
- Here is a C++ rule: if **s** is a pointer, use the arrow instead of the dot!
- To type an arrow, it's "dash"-"greater than" -- a 2-character operator symbol.

Adding Nodes To The End Of A Linked List - The Queue Model

- Adding nodes to the end of a linked list simulates "queues", which are based on "first in, first out", or FIFO.

```
// create a node
Student* s = new Student;
s->name =...; // fill its data fields
...
// add node to END of the list (queue model)
Student* p, *prev;
for (p = start, prev = 0; p; prev = p, p = p->next);
s->next = p;
if (prev) prev->next = s;
else start = s;
```

Adding Nodes To The End Of A Linked List - The Queue Model

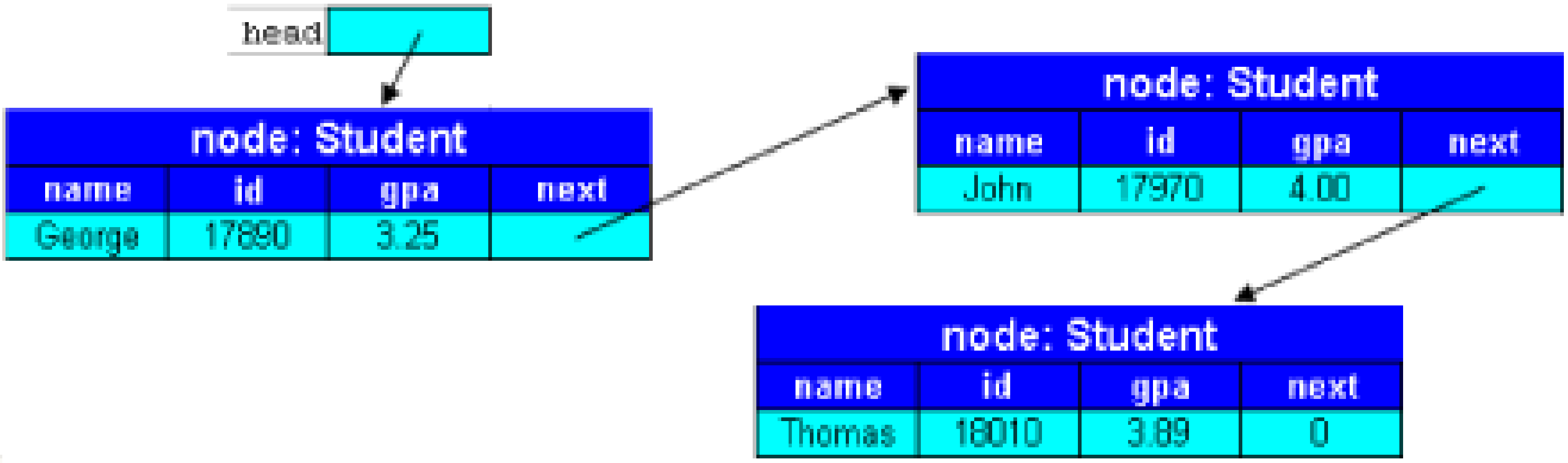
- the last five lines of code accomplish the linking of the node into the end of the list, and this code can simply be copied and pasted.
- All you would ever have to change is the identifier “start" in the 2 places it appears, and the identifier for the new node "s" in the 3 places it appears.
- If you repeat the code in the same scope, you would not need to repeat the first line -- just the remaining 4.
- no, that's not a typo -- there is a semicolon at the end of the **for**! The purpose of this loop is to move pointer p to the last node of the list and move pointer prev to the next to the last node of the list.

How start, next, nodes work

- **Student* start = 0;** does *not* create a node.
- There is no **new** keyword used, so nothing new is created here.
- Nodes have **name**, **id**, **gpa**, and **next** fields.
- **start** is just the **next** field alone, by another name.
- But **start** can refer to an otherwise existing record, if it is assigned a value other than **0** in a statement like

  start = s;
- In that case, **start->name** can be used to refer to the **name** field of whatever record **start** refers to at the time.

Visualizing a Linked List



In this example pointer start is called head and is pointing to the first node in the list or equals zero if the list is empty.

Traversing a Linked List

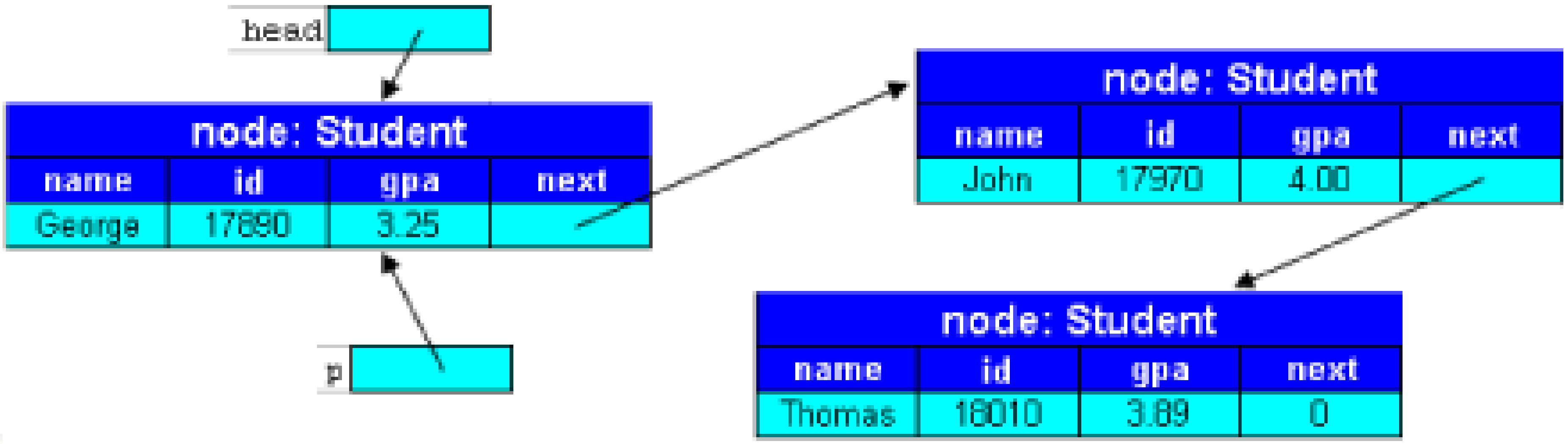
- We use a "for-loop" to traverse arrays.
- We also use a "for-loop" to traverse linked lists.
- This loop continues until p=0

```
Student* p;
for (p = start; p; p = p->next)
{
  ...
} // for
```

Traversing a Linked List

- Inside the loop, the fields are accessed with these expressions: **p->name**, **p->id**, and **p->gpa**.
- One of the tradeoffs between array-based lists and linked lists is that traversal is less efficient in linked lists.
- Also, since there are not square brackets to identify a specific node in the sequence, you would have to use traversal with an additional counter!

Traversing Linked Lists



Traversing Linked Lists

- **p** starts out referring to the same node that **start** refers to, as depicted above.
- This is accomplished in the code by the expression

  p = start.
- Then in the second cycle of the loop, its arrow moves so that it refers to the second node in the list, and so on.
- The moving of the arrow from one node to the next is accomplished in the code by the expression

  p = p->next.
- This continues until **p** finally reaches the last node and ends up with a value of **0**.

Getting the size of a Linked List

- Array-based lists carry an integer to track the size of the list. But not so with linked lists!
- While it is certainly possible to carry an integer for linked lists, too, it is typically not done.
- The reason is that it would be duplicated information, because the links determine the size.
- Managing duplicated information is always a challenge, and subject to error.
- The size-tracking integer in array-based lists is *not* duplicated information -- it is the only way to know how many elements are in the list.

Getting the size of a Linked List

- Traverse and count!

```
int size = 0;
Student* p;
for (p = start; p; p = p->next)
  size++;
```

Removing a node from the front of a list

- For simulations of stacks and queues, where nodes arrive at random times and are processed when possible, there needs to be a way to remove a node from a linked list and process it.

```
if (start) // make sure list is not already empty
{
  Student* next = start->next; // a stand-alone link
  delete start; // release memory referred to by start
  start = next; // start now refers to the next node
} // if
```

Removing all nodes

```
while (start)
{
  Student* next = start->next;
  delete start;
  start = next;
} // while
```

Linked List example

studentLinkedList.cpp

A Linked List example

- This is the studentList.cpp example rewritten using the stack model of a linked list.
- There is no sorting, because that would not make sense for either a linked list stack or queue.
- Besides, there are no library functions to sort a linked list!
- It also does not include removal of nodes.

studentLinkedList.cpp

- Note the loop at the end of the program that releases borrowed memory.
- Whenever the "new" keyword is used, extra memory is borrowed from the system for use with your program.
- If not returned, the system will not be able to reuse that memory, resulting in a "memory leak".
- So you should use the "delete" keyword before the program ends in order to avoid leaks.

Linked List of integers example

intLinkedList.cpp

inLinkedList.cpp

- You cannot have a linked list of integers or string, without first enclosing them in a struct.
- This is an example that reads integers from the keyboard, stores them in a linked list, and prints their average to the console display.

References

- INTRODUCTION TO PROGRAMMING USING C++ by Robert Burns