Dynamic Data & Linked Lists

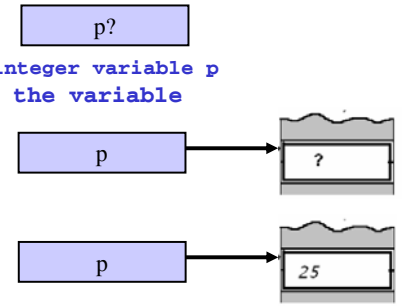
Lecture 5

- In this lecture we study the use of *pointers* to create **dynamic data structures**.
- Dynamic data structures grow and shrink during program execution, so you only use up as much memory as you need.
- The simplest and most common such data structure is a ***linked list***. We'll see how linked lists are implemented and used.
- We'll also look at the idea of ***abstract data types***, data structures whose implementation details are hidden.

Dynamic variables

- We have seen how pointers are used to pass parameters in functions. However, the most important reason for having pointers is their use in dynamic variables.
- Dynamic variables are created and destroyed while the program is running.
- Static variables (also called automatic variables) are created during compile time according to the declaration part of the program.

```
int *p;

// create a new dynamic integer variable p
// leave p to point to the variable

p = new int;


*p = 25;
```



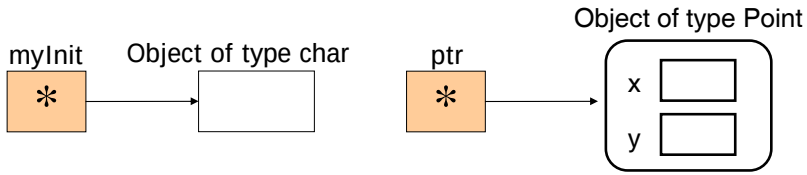
More examples of dynamic variables

- Here is a dynamic array:

```
int* myArray = new int[arraySize];
```

- We can use the `new` operator to dynamically allocate an object of any type, i.e.,

```
char* myInit = new char;

Point* ptr = new Point;
```

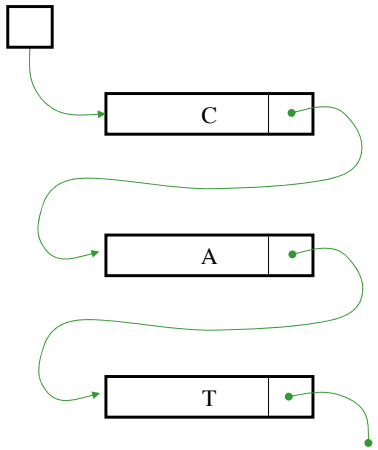


Abstract Data Types

- A linked list is also an example of an *abstract data type* (ADT). Abstract data types are an important part of modular programming.
- The idea is that the underlying implementation of the data structure is **invisible** to the procedures that use it. All they see are a number of predefined functions for manipulating it.

Linked Lists

- Here's a common example of a dynamic data structure: a ***linked list***. This is a list of characters, spelling the word "CAT".
- Assuming it's not empty, a linked list is a *pointer* to a block of memory, comprising two items: the **first element** of the list, and a pointer to the **rest of the list**.
- In other words, a linked list is an element followed by a linked list. So linked lists are ***recursively*** defined structures, in other words they are defined in terms of themselves.
- Finally, there's a special linked list: the **empty list**. This is usually represented by a special pointer, the ***null*** pointer. The end of the list is indicated by the null pointer.

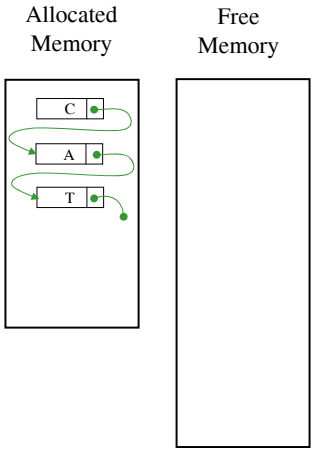


Uses of Linked Lists

- Linked lists are absolutely everywhere in almost all large application programs.
- If things are created and destroyed during a program's execution, there is usually a dynamic data structure involved — a linked list or something very similar.
- Some obvious examples include,
  - The set of open windows on your computer's desktop.
  - The set of files inside a folder.
  - The text being typed into a window, which would typically be stored as a list of blocks of characters.
- But there are many more hidden examples — temporary internal structures created during a program's execution.

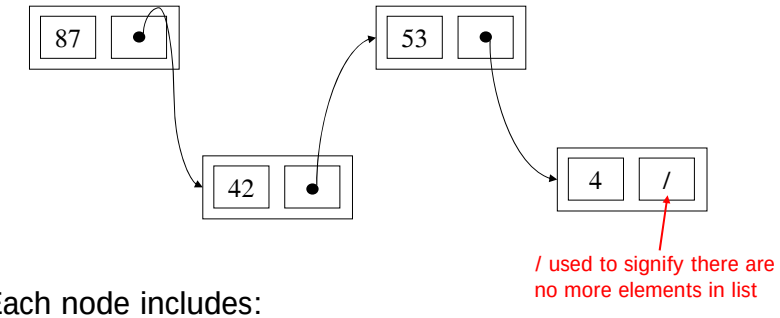
The Heap

- At run-time, every program maintains an area of memory known as the ***heap***. Dynamic data structures, such as linked lists, inhabit the heap.
- The heap comprises an area of ***allocated*** memory and an area of ***free*** memory.
- To grow a data structure — to add an element to a list, for example — the program grabs a piece of free memory, and allocates it to the new element.
- When a data structure shrinks — when an element is deleted from a list, for example — the memory allocated to that element becomes free again.
- In this way, the memory used up at any time while a program is running is exactly the amount the program needs, and no more.



Another example of a linked list

- This depicts a linked list holding a sequence of numbers: 87, 42, 53, 4.

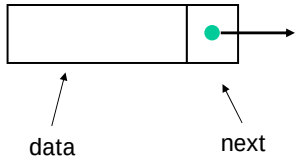


- Each node includes:
  - → A container for individual data structure
  - → A link to next node in the chain

Defining linked list in C++

- Structure of a node in a linked list implemented as class in C++
  - data member (or members) used to hold data contents of node
  - a pointer to next member in list, i.e. data member contains variable of type *Node**
- Example:

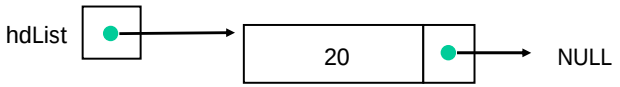
```
class Node {
   public:
        int data;
        Node* next;
};
```



Linked Lists In C++

- Let's see how to create a linked list with ONE integer. The steps are:
  1. Declare the data type for a node
  2. Declare a pointer to the head of the linked list
  3. Create the first node as a dynamic variable using the new operator
  4. Fill in the data for the node
  5. Make the next pointer pointing to null, denoting the end of the list

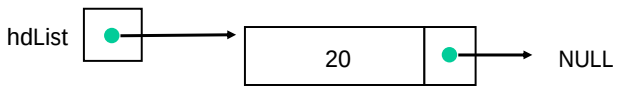
```
class Node {
   public:
     int data;
     Node* next;
};

Node* hdList;
     ........
hdList = new Node;
hdList->data = 20;
hdList->next = NULL;
```



A better version

- Better to use **typedef** to make the data type more abstract and hence easier to change in the future.
- In this case, the integer data element is defined a type named **Item**.
- We also define a type named **NodePtr** as pointer to a Node.

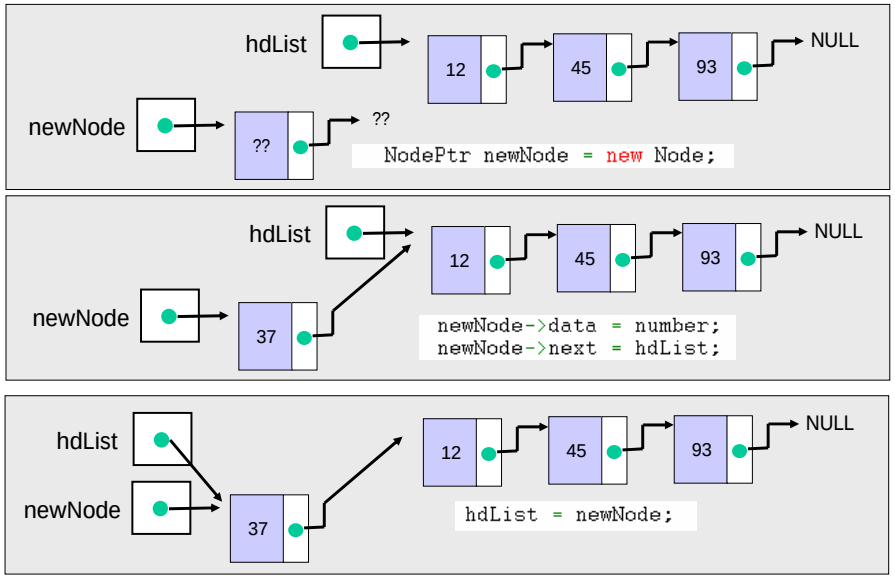
```
typedef int Item;

class Node {
   public:
     Item data;
     Node* next;
};

typedef Node* NodePtr;

NodePtr hdList = new Node;
     ......
hdList->data = 20;
hdList->next = NULL;
```



Lists as an Abstract Data Type

- Linked lists, as well as being an example of a dynamic data structure, are also an example of an ***abstract data type***.
- An abstract data type (ADT) is one whose underlying implementation is invisible to the procedures that use it. Instead, they manipulate the data structure via a number of predefined functions and routines, known as ***access routines***.
- Access routines come in two main varieties: those for constructing data structures, and those for accessing their parts.
- Let us first consider the access routine that adds an item to the list.

How to add to head of list?



Access routines to build the list

- **addToList** takes two parameters: the number to add, and a pointer pointing to the head of the list.
- On exit, **hdList** will be pointing to a new node. Therefore it must be passed to the function by reference, hence the '**&**' in front of **hdList**.

```
void addToList (Item    number,
                NodePtr &hdList) {
    NodePtr newNode = new Node;
    newNode->data = number;
    newNode->next = hdList;
    hdList = newNode;
}
```

- Is the new item added to the beginning (i.e. head) or the end (i.e. tail) of the linked list?

Build a list of szList items

- The function **buildList** will create a linked list of **szList** nodes and return a pointer **hdList** that points to the head of the list.
- The items are generated randomly in the routine **getItem()**.
- **firstTime** is used to detect when **getItem()** is first called and **randomize()** is invoked for the random number generator.

```
#define DEBUG FALSE
bool firstTime = TRUE;

Item getItem () {
    int number;
    if (firstTime == TRUE) {
        randomize();
        firstTime = FALSE;
        }
    number = rand();
    if (DEBUG==TRUE)
        cout << number << endl;
    return Item(number);
}

void buildList (int     szList,
                NodePtr &hdList) {
    int i;
    for (i = 0; i < szList; i++)
        addToList(getItem(), hdList);
}
```

- If **DEBUG** is true, the random numbers are displayed on the console as they are generated.

Access routine to read the data

- **getFromList()** extracts one item from the head of the list and than return the its value. In addition, it also return a new pointer pointing to rest of the list NOT include the returned item.

```
Item getFromList (NodePtr &hdList) {
    int number;
    number = hdList->data;
    hdList = hdList->next;
    return Item(number);
}

void printAll (NodePtr hdList) {
    cout << "\nList of numbers are:\n";
    while (hdList != NULL) {
        cout << getFromList(hdList) << endl;
    }
}
```

- **printAll()** displays on the console ALL the items stored in the entire linked list. It terminates when the next pointer is pointing to NULL (i.e. empty).
- Why is it better NOT to combine these two routines into one?
- In what order is the list printed?

Putting these together

- Here is the very clear and clean main program.
- However, this program has a memory leak, i.e. it create dynamic memory to store 4 items, and these are never returned even after the items are taken from the list.

```
int main()
{
    buildList (4,hdList);
    printAll (hdList);
    getchar();
}
```

- To fix this problem, **`getFromList()`** is modified to delete the node immediate after it is extracted from the list.

```
Item getFromList (NodePtr &hdList) {
    int number;
    NodePtr nowPtr;
    nowPtr = hdList;
    number = nowPtr->data;
    hdList = nowPtr->next;
    delete nowPtr;
    return Item(number);
}
```


Maintaining the Heap

- To ensure that space on the heap is not wasted, redundant structures should be taken apart, and the space they occupy made available again.
- In C++, the delete operator is used to return unwanted memory back to the heap.
- As an alternative to the previous slide here's a function that destroys the list that we have created.

```
void destroyList (NodePtr hdList) {
    NodePtr nowPtr;
    nowPtr = hdList;
    while (hdList != NULL) {
        hdList = hdList->next;
        delete nowPtr;
    }
}
```


What to do before the next lecture?

- Complete Exercise C (non-assessed) – See separate sheet
- Read first part of Chapter 15.1 of Savitch