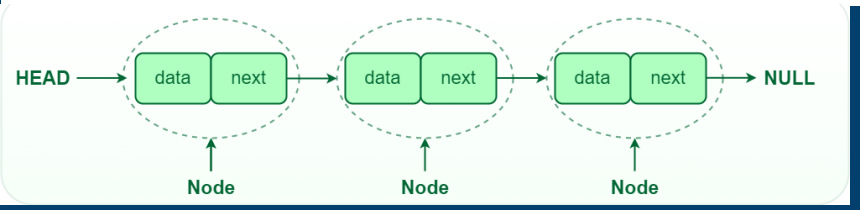
ensia The National School of Artificial Intelligence
المدرسة الوطنية العليا للذكاء الاصطناعي



Data Structure & Algorithms 1

CHAPTER 6: DYNAMIC MEMORY ALLOCATION (POINTER ARRAY – LINKED LIST)

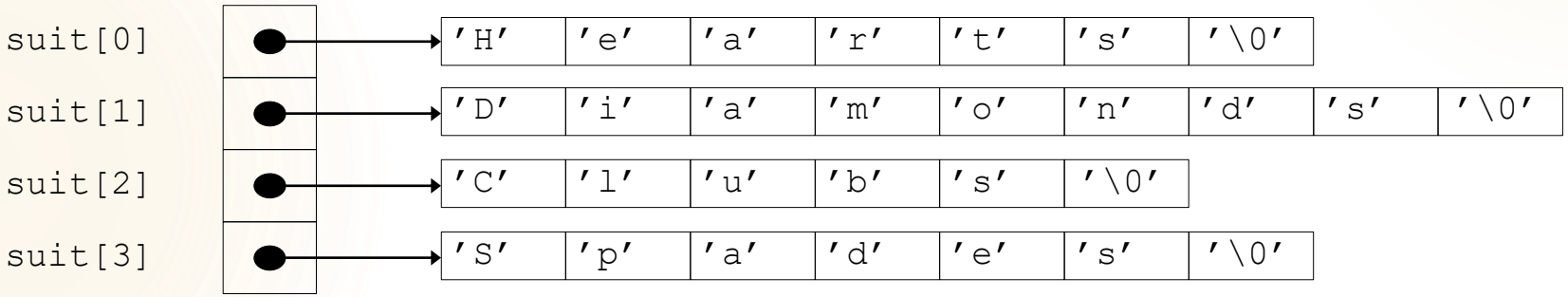
Sep – Dec 2023

Array of Pointers

- Arrays can contain pointers
  - Commonly used to store array of strings

```
char *suit[ 4 ] = {"Hearts", "Diamonds", "Clubs", "Spades" };
```

  - Each element of **suit** points to **char *** (a string)
  - Array does not store strings, only pointers to strings

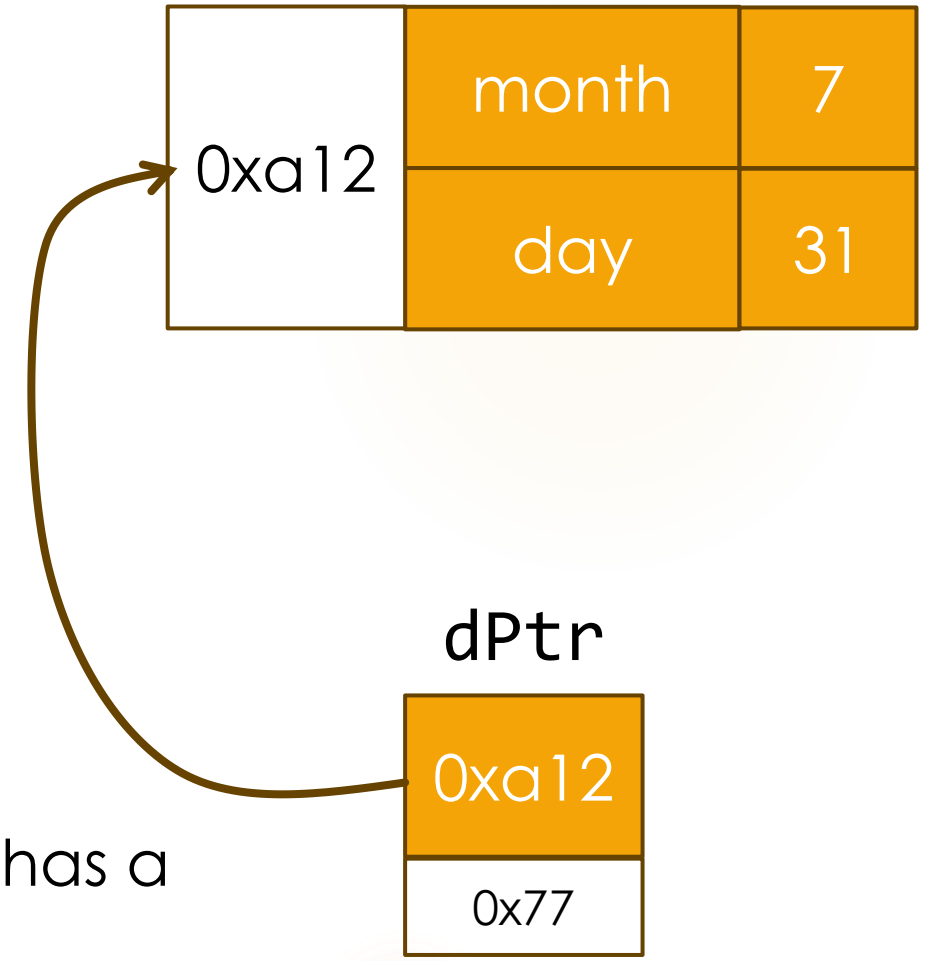


  - **suit** array has fixed size, but strings can be of any size

Pointers and Structs

- Pointers can point to a struct or class instance as well as to a regular variable.
- One way to do this would be to dereference and then use dot notation:

```
Date d;
d.month = 7;
Date* dPtr = &d;
cout << (*dPtr).month << endl;
```

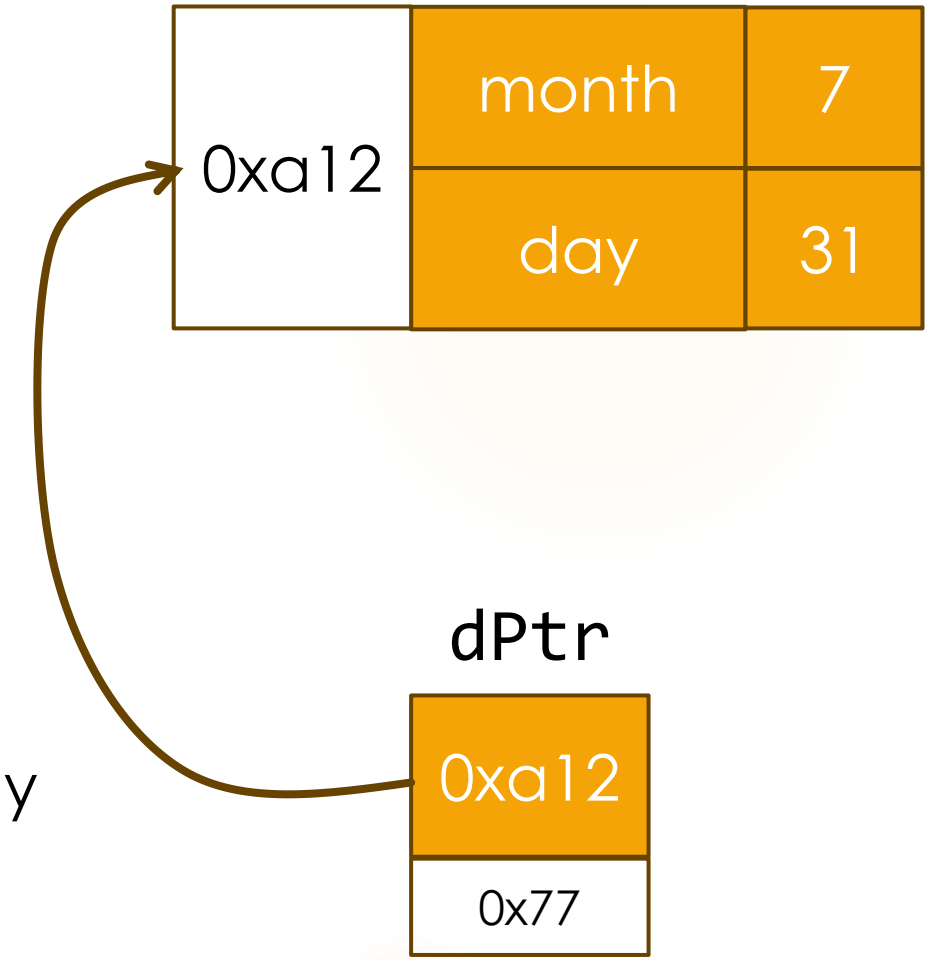


But, this notation is cumbersome, and the parenthesis are necessary because the "dot" has a higher precedence than the *.

Pointers and Structs

- So, we have a different, and more intuitive syntax, called the "arrow" syntax, **->**

```
Date d;
d.month = 7;
Date* dPtr = &d;
cout << dPtr->month   << endl;
```

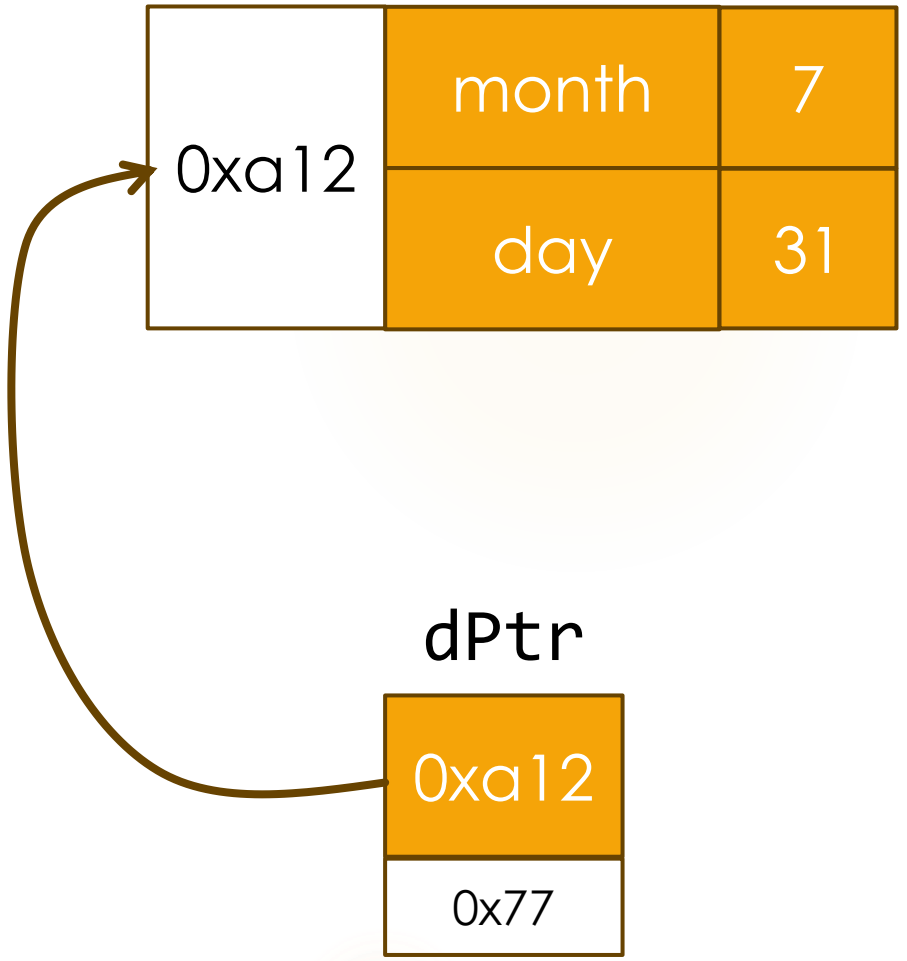


We will use the arrow syntax almost exclusively when using structs.

Pointers and Structs

- The arrow syntax can be used to set a value in a struct via a pointer, as well

```
Date d;
d.month = 7;
Date* dPtr = &d;
cout << dPtr->month << endl;
dPtr->day = 1;
cout <<dPrtp->day << endl; // 1
```



Dynamic memory allocation

So far in this course, all variables we have seen have been local variables that we have defined inside functions. Sometimes, we have had to pass in object references to functions that modify those objects. For instance, take a look at the following code:

```
struct Vector {
  int data[100];
  int size;
};
void squares(Vector &vec, int numSquares) {
  for (int i = 0; i < numSquares; i++) {
    vec.data[i] = i * i;
  }
}
```

This function requires the calling function to create a struct to use inside the function. This isn't necessarily bad, but could we do it a different way? In other words, could we create the **Vector** inside the function and just pass it back?

Dynamic memory allocation

Could we create the Vector inside the function and just pass it back?

```
Vector squares(int numSquares) {
  Vector vec;
  for (int i = 0; i < numSquares; i++) {
    vec.data[i] = i * i;
  }
  return vec;
}
```

- Does this work?
  - Could we create the Vector inside the function and just pass it back? It actually does, but there is an issue:
    - you have to make a copy of the Vector, which is inefficient (though...these days if the creator of the Vector class is clever, we won't have to make a copy). Remember, we would rather not pass around large objects

Dynamic memory allocation

Okay…maybe we can do this?

```
Vector &squares(int numSquares) {
  Vector vec;
  for (int i = 0; i < numSquares; i++) {
    vec.data[i] = i * i;
  }
  return vec;
}
```

- Does this work?
  - No :(This is actually really bad. Why?
    - The scope of **vec** is only the function, and you are not allowed to pass back a reference to a variable that goes out of scope.

Dynamic memory allocation

Well, how about with pointers? Can we do this?

```
Vector *squares(int numSquares) {
  Vector vec;
  for (int i = 0; i < numSquares; i++) {
    vec.data[i] = i * i;
  }
  return &vec;
}
```

- Does this work?
  - No :(This is also really bad. Why? Same as before: the scope of **vec** is only the function, and you are not allowed to pass back a pointer to a variable that goes out of scope. When the function ends, the variable is destroyed, and your program will almost certainly **crash**.

Dynamic memory allocation

What do we want here? What's the big deal?

- What we really want is really two things:

1. a way to reserve a section of memory so that it remains available to us throughout our entire program, or until we want to destroy it (give it back to the operating system)
2. a way to reserve any amount of memory we want at the time we need it.

- You might think that global variables are what we want, but that would be incorrect.
  - Global variables can be accessed by any function in our program, and that isn't what we want. Also, global variables have a fixed size at compile time, and that isn't what we want, either

Dynamic memory allocation: new

C++ allows you to request memory from the operating system using the keyword **new**. This memory comes from the "heap" whereas variables you simply declare come from the "stack."

- Variables on the stack have a scope based on the function they are declared in.
- Memory from the heap is allocated to your program from the time you request the memory until the time you tell the operating system you no longer need it, or until your program ends.
- To request memory from the heap, we use the following syntax:

```
type *variable = new type;    // allocate one element or
type *variable = new type[n]; // allocate n elements
```

Dynamic memory allocation: new

Examples:

```
int *anInteger = new int; // create one integer on the heap
int *tenInts = new int[10];   // create 10 integers on the heap
```

- The second example (**tenInts**) is very powerful:
  - the memory you are given is an array guaranteed by the operating system to be contiguous. So, that's how we allocate an array of items dynamically!
- Notice that **new** returns a pointer to the type you request
  - this is important! This is why we need to learn about pointers — in order to dynamically allocate memory, you have to use a pointer.

Dynamic memory allocation: delete

- The memory you request is yours until the end of the program, if you need it that long.
- You can pass around the pointer you get back as much as you'd like, and you have access to that memory through that pointer in any function you pass the pointer to.
- But, what if you are done using that memory? Let's say you create an array of 10 ints, use them for some task, and then are done with the memory?
- In this case, you **delete** the memory, giving it back to the Operating System

Dynamic memory allocation: delete

```
int *tenInts = new int[10]; // create 10 integers on the heap
for (int i=0; i < 10; i++) {
     tenInts[i] = i + 1;
}
someFunction(tenInts);
// done using tenInts
delete [] tenInts; // the [] is necessary for an array
```

Dynamic memory allocation: delete

delete is sometimes confusing. Take a look at the following function:

```
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];
    for (int i=0; i < length; i++) {
       multiple[i] = origArray[i] * 2; // double each value
    }
    printArray(multiple, length); // prints each value doubled
    delete [] multiple; // give back the memory
    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
       multiple[i*2] = origArray[i] * 2; // double each value
       multiple[i*2+1] = origArray[i] * 3; // triple the value
    }
    printArray(multiple, length * 2);
    delete [] multiple; // clean up
}
```

Dynamic memory allocation: delete

delete is sometimes confusing. Take a look at the following function:

```
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];
    for (int i=0; i < length; i++) {
       multiple[i] = origArray[i] * 2; // double each value
    }
    printArray(multiple, length); // prints each value doubled
    delete [] multiple; // give back the memory
    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
       multiple[i*2] = origArray[i] * 2; // double each value
       multiple[i*2+1] = origArray[i] * 3; // triple the value
    }
    printArray(multiple, length * 2);
    delete [] multiple; // clean up
}
```

First: notice that we delete multiple, and then use it again!

- Is that allowed??

It is! **delete** does not delete any variables! Instead, it follows the pointer and returns the memory to the OS!

- However, you are not allowed to use the memory after you have **deleted** it.
- This does not preclude you from re-using the pointer itself

Dynamic memory allocation: delete

What does this print out, for an **origArray = {1, 5, 7}?**

```
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];
    for (int i=0; i < length; i++) {
       multiple[i] = origArray[i] * 2; // double each value
    }
    printArray(multiple, length); // prints each value doubled
    delete [] multiple; // give back the memory
    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
       multiple[i*2] = origArray[i] * 2; // double each value
       multiple[i*2+1] = origArray[i] * 3; // triple the value
    }
    printArray(multiple, length * 2);
    delete [] multiple; // clean up
}
```

```
void printArray(int *array, int length) {
  cout << "[";
  for (int i=0; i < length; i++) {
    cout << array[i];
    if (i < length-1) {
      cout << ", ";
    }
  }
  cout << "]" << endl;
}
```

Output:

[2, 10, 14]
[2, 3, 10, 15, 14, 21

Dynamic memory allocation: delete

- The memory you request is yours until the end of the program, if you need it that long.
- You can pass around the pointer you get back as much as you'd like, and you have access to that memory through that pointer in any function you pass the pointer to.
- The standard library classes, **string**, uses dynamic memory.

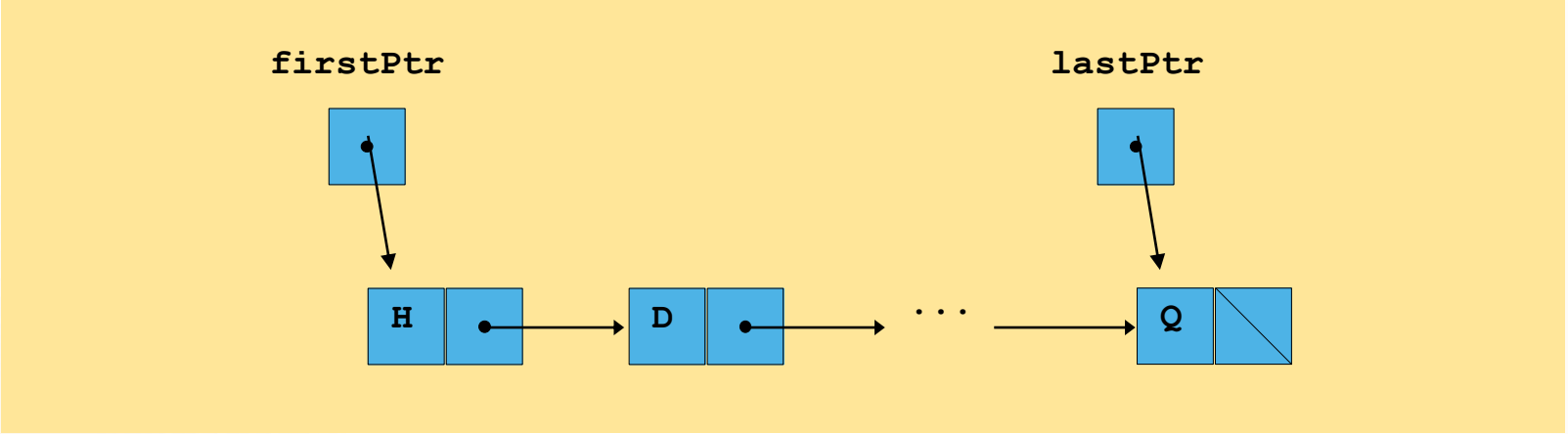
Dynamic memory allocation: Recap

- **`new`**: used to request heap memory that lasts for the rest of your program, or until you don't need it anymore.
- **`delete`**: used to return memory to the operating system.
- If you use new to request memory, you should **`delete`** it somewhere in your program.
- You are **not allowed** to use memory that has been **`delete`**d.
- deleting memory does not somehow "delete" the pointer variable -- it goes to the location in memory pointed to, and tells the operating system that we are done with it

Linked Lists

- Collection of self-referential struct object (nodes) connected by pointers (links)
- Accessed using pointer to first node of list (**head**)
  - Subsequent nodes accessed using the links in each node
- Link in last node is null (zero)
  - Indicates end of list
- Data stored dynamically
  - Nodes created as necessary
  - Node can have data of any type, including objects of other structs
- Linked Lists, Stacks and Queues are linear data structures.
- Trees are non-linear data structures

Linked Lists




```
struct Node {
   int  data;
   Node *  next;
};
```

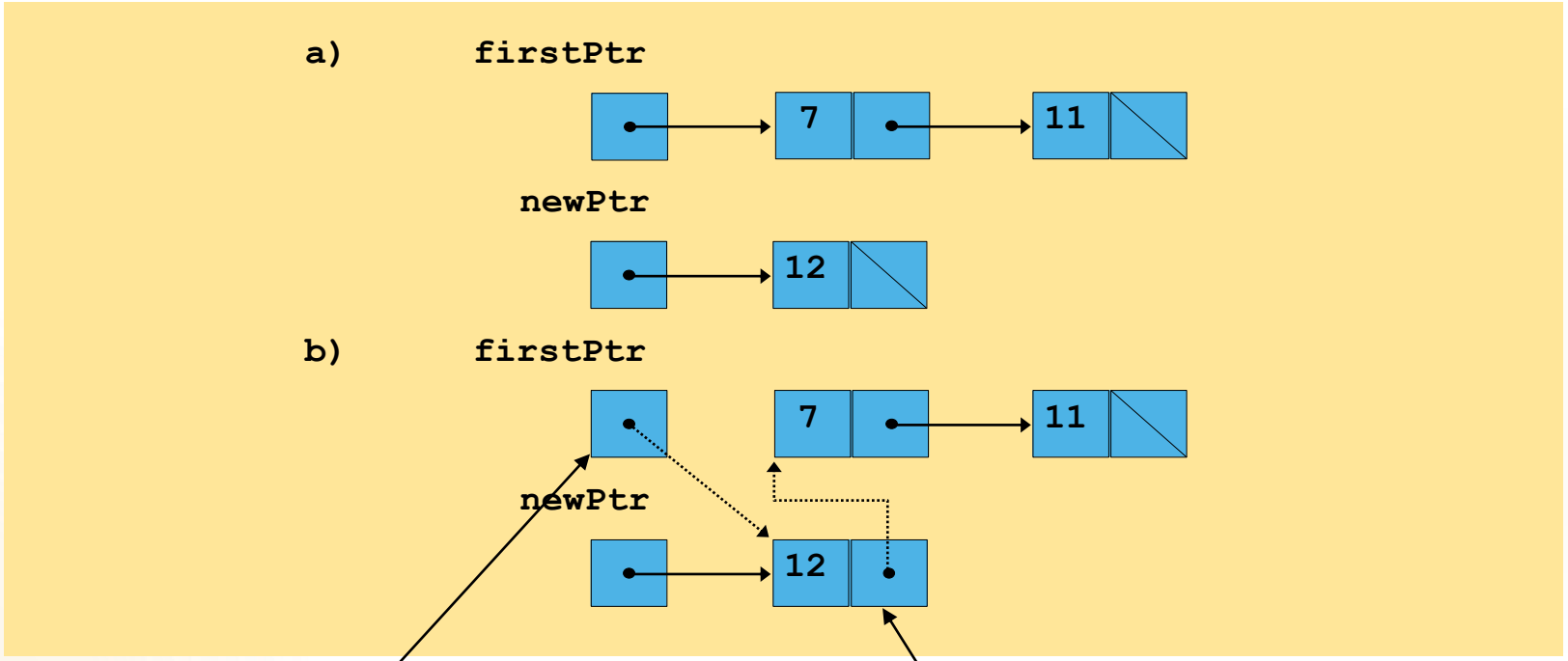
Linked Lists

- **Linked lists vs. arrays**
  - Arrays can become full
    - Allocating "extra" space in array wasteful, may never be used
    - Linked lists can grow/shrink as needed
    - Linked lists only become full when system runs out of memory
  - Linked lists can be maintained in sorted order
    - Insert element at proper position
    - Existing elements do not need to be moved (unlike arrays)
  - Arrays provide direct access to elements (unlike linked lists)

Linked Lists

- Selected linked list operations
  - Insert node at front
  - Insert node at back
  - Remove node from front
  - Remove node from back
- In following illustrations
  - List has **`firstPtr`** and **`lastPtr`**
  - (a) is before, (b) is after

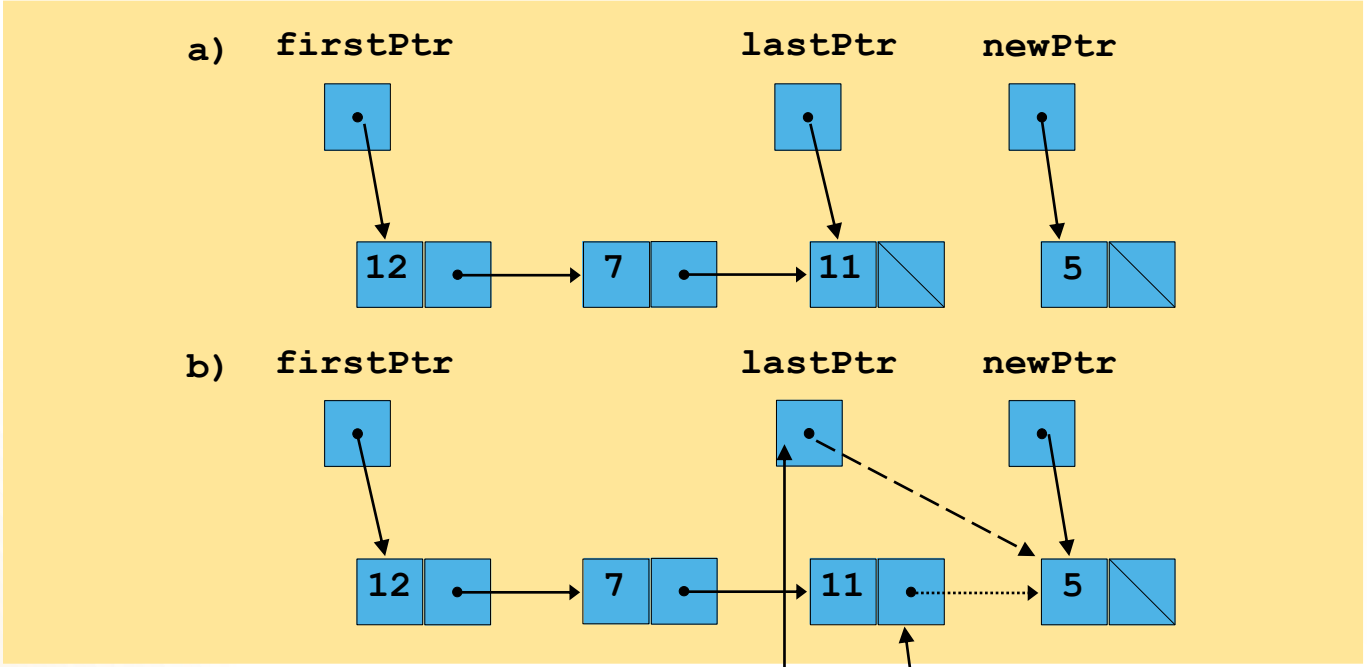
Insert at front




```
newPtr->next = firstPtr
```

```
firstPtr = newPtr
```
If list empty, then
```
firstPtr = lastPtr = newPtr
```

Insert at back



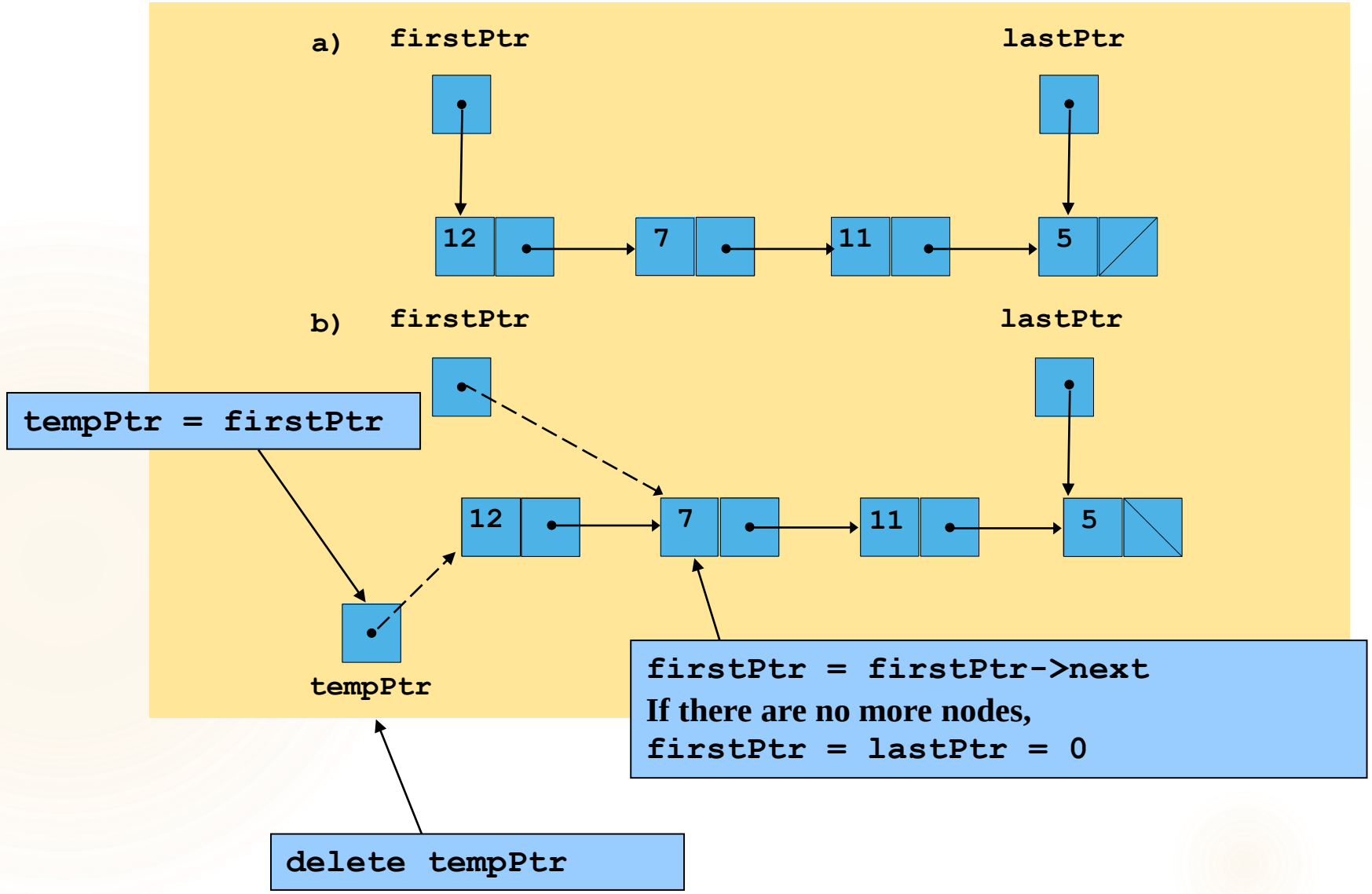

```
lastPtr->next = newPtr
```

```
lastPtr = newPtr
```

If list empty, then

```
firstPtr = lastPtr =
newPtr
```

Remove from front



Remove from back