Structures
Pointers and Structures
Linked Lists

Abstract data types

- Abstraction = model
  - present characteristics, model, design
  - not the concrete data or objects
- Example: design of a database
  - tables, fields, properties
- Example: many math definitions
  - matrix = a table of numbers, etc
  - vectorial space = a set with algebraic operators and properties
- Abstractions very useful for humans when building "logic"

Combined data = structure

- in C++ we can create a new "user" type
- `class person {` *//this is the new defined type*
  - `int ID;` *// these are members*
  - `int age;`
  - `char name[25];`
  - `int phone;`
  - `char* address;`
- `}`
- `person x;` *//declare variable* `x` *of type* `person`
  - x contains combined data: ID, age, name, etc
  - think of it like a "box" variable, or "record"
  - how much memory x is allocated?

ID
AGE
NAME
PHONE
ADDRESS

Structure Members

- `person x,y;` //declares two struct variables, same type
- x.age is an integer variable for record x
  - x.age is independent of y.age
  - x.age independent x.ID, etc

Struct variables

- What can we do with a struct/record variable?
- Answer : everything that we do with normal variables.
  - declare
  - initialize
  - assign
  - point to
  - address of
  - array of
  - etc

Struct variables

```
person x ={21, 34, "Virgil", 1234567};
```

- declares x of type person
- initializes x.ID=21, x.age=34, x.name="Virgil", x.phone=1234567
- x.address not initialized - WHY ?

Struct variables

- Assignments work !
- `person x, y;`
- .....
- `x=y;` //valid: all members of y are copied on x
  - BE CAREFUL ABOUT POINTER MEMBERS!
  - copy pointer/address VS copy the content(value) of the pointer
  - x=y copies the pointer (address), not the value
  - deep copy :
    - allocate x.pointer separately,
    - copy *(y.pointer) into *(x.pointer)

Array of struct variables

- `person A[10];` //declares an array of 10 struct objects
- A[0] = first object/variable, A[1]= second variable
- A[0].ID = member ID of first object
- most array operations work like before

Struct object as function parameter

```
int myfunction (person x) { //regular parameter
  cout << x.ID;
return 0;}
```

```
int myfunction (person &x) { //reference parameter
  cout << x.ID;
  x.ID=25; //modifies the original call variable -WHY ?
return 0;}
```

```
int myfunction (person* x) { //pointer parameter
  cout << (*x).ID;
return 0;}
```

Pointers to Struct Objects

- `person *p;` p=memory location of a `person` object
- *p = the "value", or the struct object stored
- (*p). ID = the ID member variable of object *p
- p->ID = the ID member variable of object pointed by p
  - same as (*p).ID

Dereferencing member variables

Table 11-3

Expression	Description
`s->m`	`s` is a structure pointer and `m` is a member. This expression accesses the `m` member of the structure pointed to by `s`.
`*a.p`	`a` is a structure variable and `p`, a pointer, is a member. This expression dereferences the value pointed to by `p`.
`(*s).m`	`s` is a structure pointer and `m` is a member. The * operator dereferences `s`, causing the expression to access the `m` member of the structure pointed to by `s`. This expression is the same as `s->m`.
`*s->p`	`s` is a structure pointer and `p`, a pointer, is a member of the structure pointed to by `s`. This expression accesses the value pointed to by `p`. (The -> operator dereferences `s` and the * operator dereferences `p`.)
`*(*s).p`	`s` is a structure pointer and `p`, a pointer, is a member of the structure pointed to by `s`. This expression accesses the value pointed to by `p`. `(*s)` dereferences `s` and the outermost * operator dereferences `p`. The expression `*s->p` is equivalent.

Array of struct objects

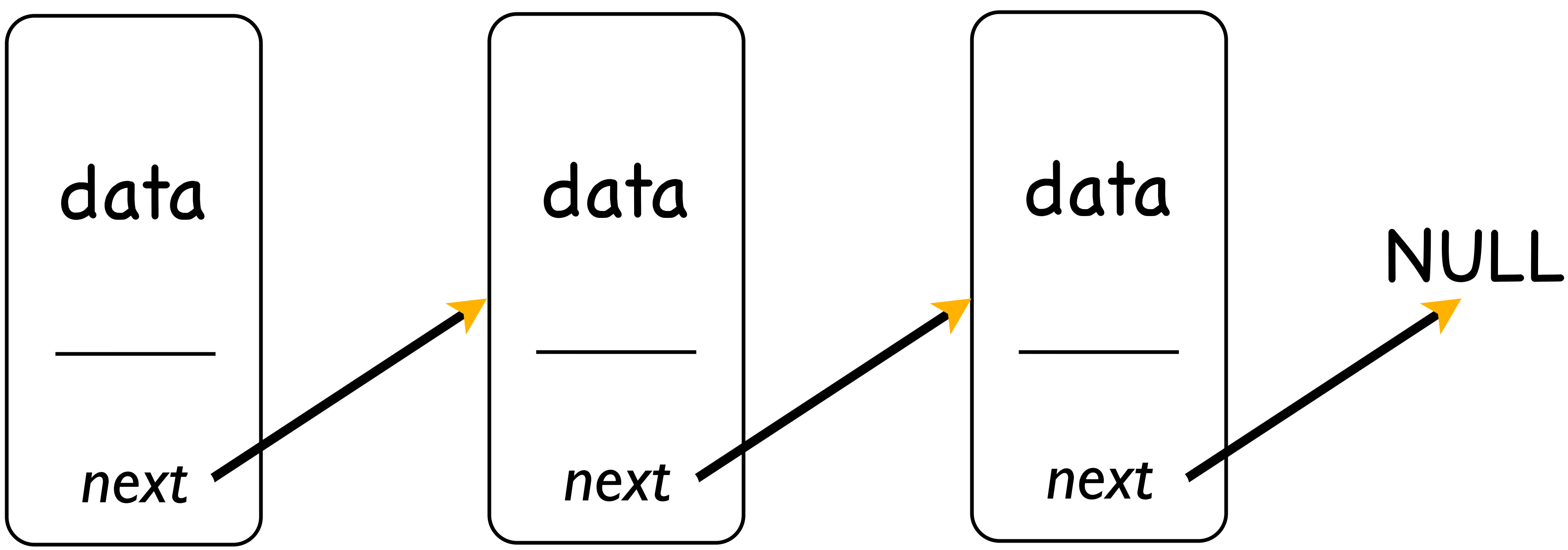
- `person *p = new person[20];` //declares a pointer, allocates dynamically space for 20 `person` objects
  - (same as) `person p[20];` //but this is static

- `person* p[20] ;` //static array of 20 pointers

Linked Lists

Link List Philosophy

- List objects: contain data, and the link to the next list object



- how do we implement this in C++ ?

Linked List

```
class listobject{
•   char* word;
•   int count;
•   double testscore;
•   char[30] name;

  listobject* next;

};
```

//data section

//link to next object

have to "know" the first list object, to have a way to get to it

Traversing a list looking for "value"

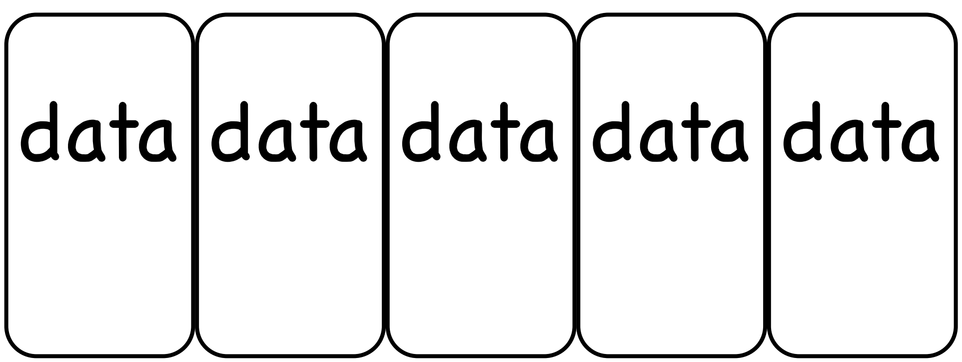
- case 1: list does not exist
  - create the first object, return it
- case 2: list exists, but doesnt have an object with data="value"
  - create a new object, append it to the list, return it
- case 3: list has an object with data="value"
  - return that object

Traversing a list

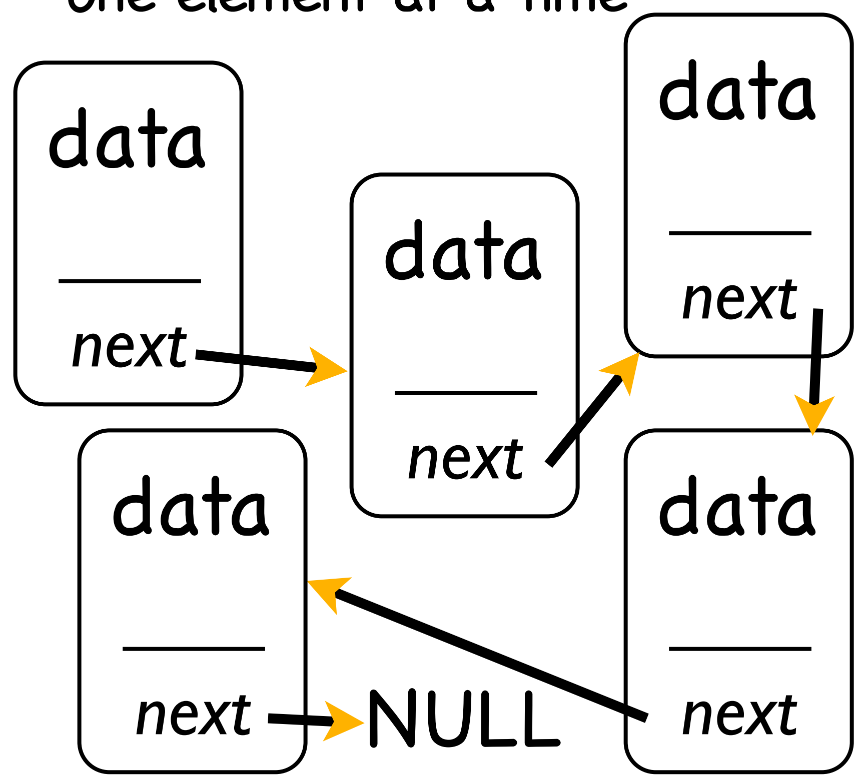
- listobject* GiveMeTheElement (value)
  - listobject* t = <my_list_head>
  - if t==0 CASE 1 *//create the first object of a new list*
  - while (t->data != value){ *//looking for "value" object*
    - if (t->next==NULL) CASE 2 *//create a new object of existing list*
    - t = t->next *//keep looking*
  - }
  - CASE 3 *//found the "value" object*
- }

Arrays vs Lists

- Arrays are a contiguous block of memory
  - no need for "next"-WHY?
- Arrays allow for direct access to n^{th} element A[n]
- Arrays have to be allocated at once

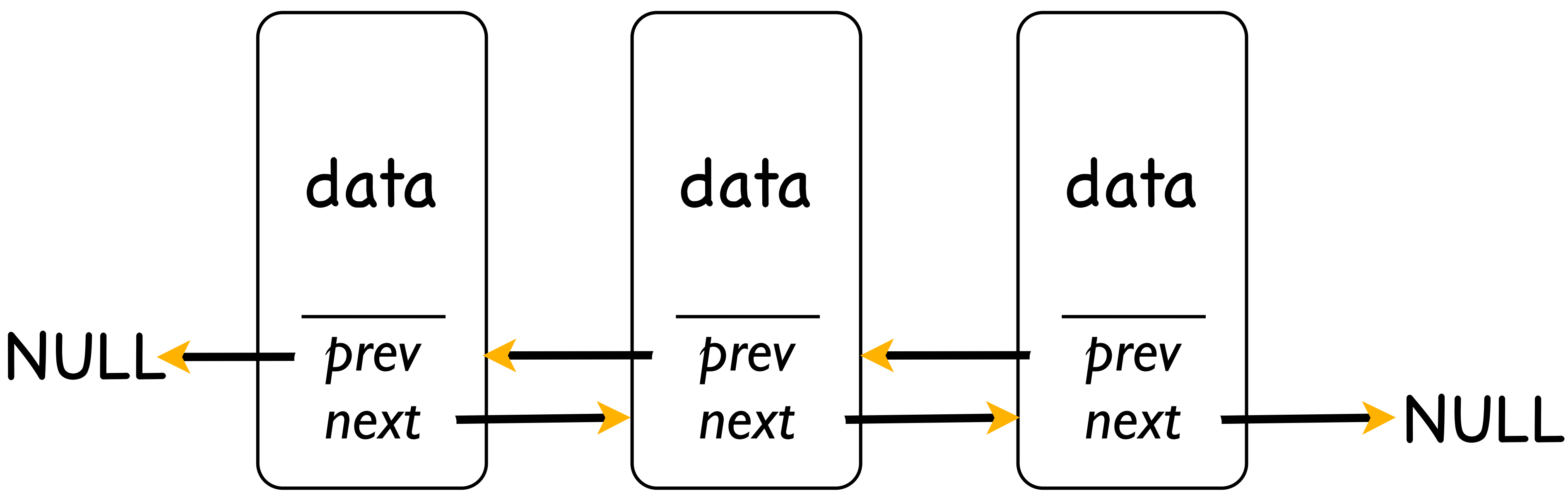


- Lists are sparse locations in memory
- Lists have to be traversed from beginning in order to access an element
- Lists are allocated "as we go" one element at a time



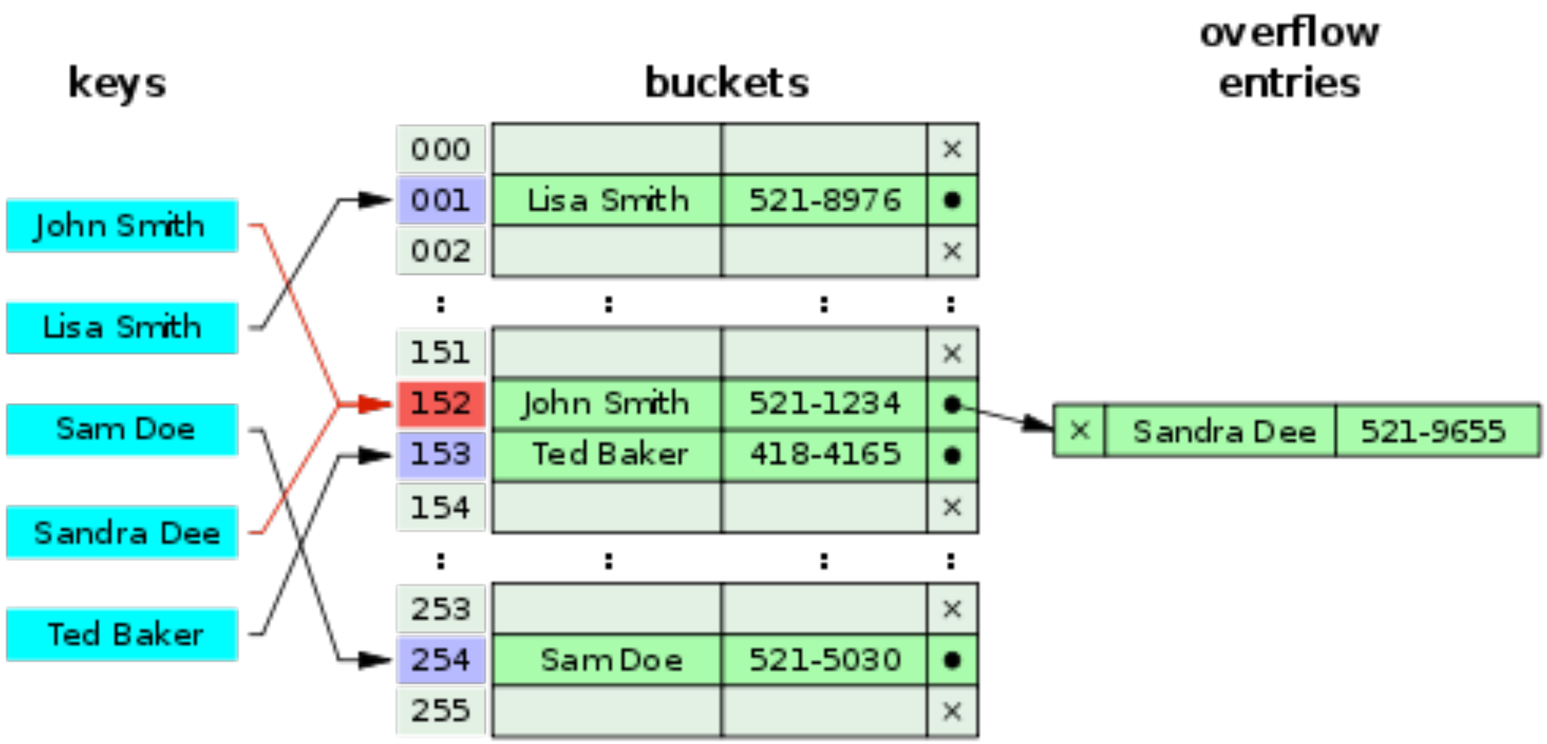
Double-linked Lists

- Use two link pointers : `prev`, and `next`
- Thus we can traverse the list in any direction



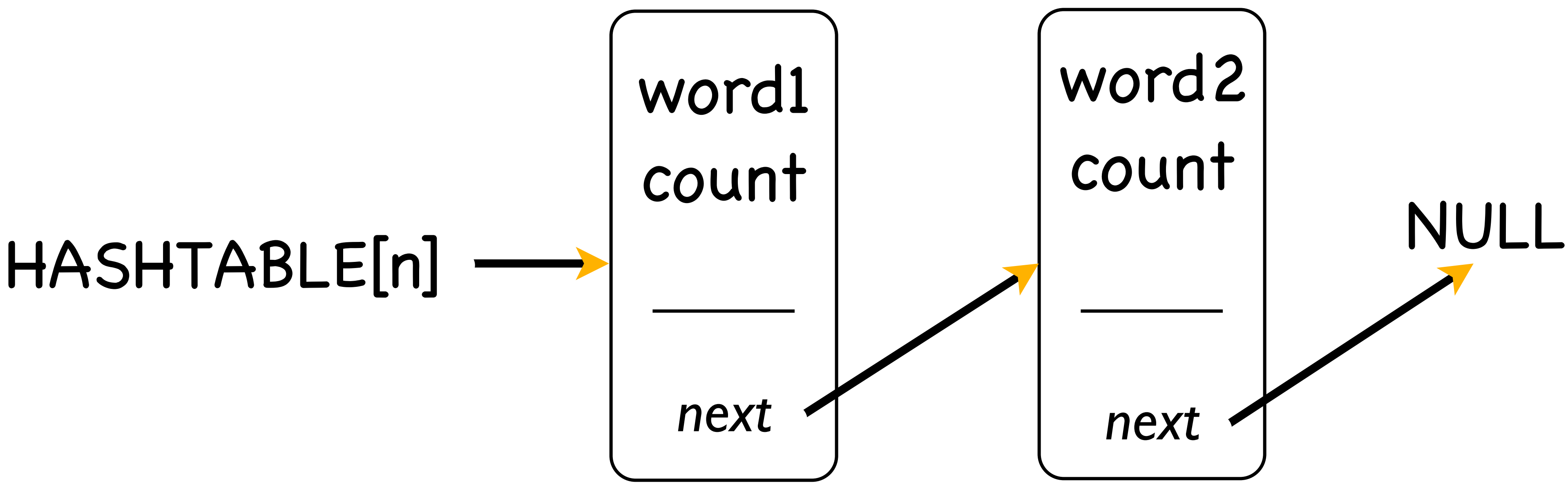
Hash Tables - Collisions

- when several keys (words) map to the same key (index)
- have to store the actual keys in a list
  - list head stored at the HASHTABLE index
- key -> index -> list_head -> search for that key



Hashing

- for each hash value, create a linked list of all strings that hash to that value
- if hfunction (word1) = hfunction(word2) =n
- then HASHTABLE[n] stores the head of a list containing objects (word1, count1) and (word2, count2)



Hashing with linked lists

- HASHTABLE[n] = listhead of a list with all words that hash-map to n
- when accessing an object "word"
  - first get the hash value n = hash-map("word")
  - then traverse the list starting at HASHTABLE[n] looking for the the object that has "word"
  - once found, do something with it : for the HW, increase the word count.