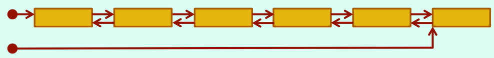
CS205

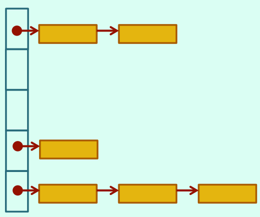
C / C++

Stéphane Faroult
faroult@sustc.edu.cn

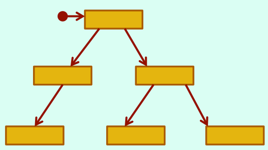
Wang Wei (Vivian) vivian2017@aliyun.com



We have seen last time lists, which include several variations such as a double link and, when the active part of the list is its end, a tail pointer to the last node.



We have also seen hash tables, which are arrays in which each slot is the start of a list. That list contains values for which the computation of the hash function gives the same slot number in the array.



And finally we have seen trees, which can be binary trees or more complicated structures, before discussing pointers on function.

Functions available in C

```c
#include <search.h>
```

BSD system

The little devil is the BSD logo, not a personal opinion.

```c
typedef struct entry {
        char    *key;
        void    *data;
       } ENTRY;
```

An influential Unix variant created at the University of Berkeley and called BSD brought to Unix and Linux "Collection" routines based on items contening a key and an undefined pointer to associated data.

Functions available in C

```c
#include <search.h>
```

BSD system

```c
int  hcreate(size_t nel);

void hdestroy(void);

ENTRY *hsearch(ENTRY item, ACTION action);
```

These ENTRY items can be used with hash tables.

There are also tree functions that take function pointers as parameters.

Functions available in C

```c
#include <search.h>
```

BSD system also tfind() and tdelete()

```c
void *tsearch(const void *key, void **rootp,
              int (*compar) (const void *key1,
                             const void *key2));

void twalk(const void *root,
           void (*action) (const void *node,
                VISIT order, int level));
```

Functions available in C

```c
#include <sys/types.h>
#include <db.h>
```

B-tree functions

There are also in Linux functions for managing B-trees that come from BSD systems as well.

GLib

The open-source GNU library, Glib, also contains many functions for managing data structures (the gcc compiler comes grom GNU, as well as the Gnome graphical environment frequently used on Linux systems)

It's often easier to write one's own custom functions than to use generic ones ...

Flickr: Mike Goren

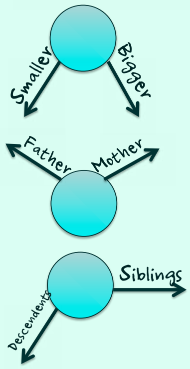
I don't recommend using these functions before you have a lot of practice. They are far harder to use than Java Collections, for instance. Because these functions have to be generic, they take as parameters function pointers, and void pointers (or pointers on void pointers ...). You must use a lot of "casts" to silence warnings, you may use the wrong pointer very easily, and debugging your program may become far more painful that if you had written a handful of functions tailored for your needs.

Comparing Data Structures

What to use, and when?

As already stated, data structures are the topic of Computer Science classes in their own right, and what we have seen here is to be understood as a practical but modest introduction to the topic. What is important with data structures, though, is to know when to use them (you don't need a tree for ten values), and for which purpose.

We have seen PATTERNS



Also keep in mind that even the binary tree is just a global pattern. For instance, instead of having "smaller" and "bigger" values, we could represent the full ascendency of one individual by having pointers to "father" and "mother", or a descending genealogical (or other) tree by having a left-side pointer to the first child, and the right-side pointer to siblings.

We have seen PATTERNS

Tons of variations are possible, as well as combining data structures.

There is nothing to prevent from adding to a tree node an additional head to a linked list, or having a node contain an array. It all depends on what your storage needs are, and finding clever ways to keep data organized and accessible.

WHAT SHOULD YOU REMEMBER ABOUT DATA STRUCTURES?

What is important is for what kind of problems you should use them.
You will notice that for one type of problem, several kinds of data structures can do the job. There is always a kind of problem for which they aren't appropriate at all.

Arrays Linked Lists Hash Tables Trees

If you need to keep track of when a piece of data was added to a structure, both arrays (higher index = more recent) and linked lists (at the tail = more recent) can be good choices, hash tables and trees are poor ones.

Time Dimension (LIFO/FIFO) Keep Ordered Search Add/ Remove

Arrays Linked Lists Hash Tables Trees

If keeping pieces of data ordered by something else than chronology is important, both linked lists and trees can do it. Arrays would require sorts, and hash tables just can't do it.

Time Dimension (LIFO/FIFO) Keep Ordered Search Add/ Remove

Arrays Linked Lists Hash Tables Trees

For finding information fast, both hash tables and trees are great. If data is static, a sorted array can be a good choice too because binary searches are efficient.

Time Dimension (LIFO/FIFO) Keep Ordered Search Add/ Remove

Arrays Linked Lists Hash Tables Trees

For adding and removing information dynamically, linked lists and hash tables and trees are great. For arrays, it depends: if you use the array as a stack (adding to and removing from higher indices) it can work; otherwise, no.

Time Dimension (LIFO/FIFO) Keep Ordered Search Add/ Remove

Reality-Check

Is it critical?

Enough data to make a difference?

Once again, don't over engineer. It's no use having a complicated data structure when the performance improvement over a plain array is barely noticeable, if noticeable at all. Simpler means fewer bugs.

Finally, there are also in-memory databases that can be a very nice alternative to very complicated data structures.

Think of in-memory databases too ...

Unfortunately **everything** cannot fit into **memory.**

This is why data usually needs to be persisted to files or databases. I hope we'll have time to come back to this topic towards the end of this course.

QUIZ 1

And now the answers to Quiz 1.

A condition such as:

```
if (0 = a) { ... }
```

a. Is true if a is equal to 0, false otherwise
b. Is true whatever the value of a is
c. Is false whatever the value of a is
d. Generates a compilation error

One single equal sign is interpreted as an assignment, and you cannot assign the value of a variable to a constant. Writing constant first in conditions may help avoid some errors.

If in a program you type `print()` instead of `printf()`:
a. It causes a compilation error
b. It causes a link error
c. It causes a runtime error

The compiler sees an unknown function print() and may issue a warning, but it will assume that it returns an int and fo on. It's when the linker looks for the code of the function in the standard library to add it to your program that you'll run into trouble.

If you read a 50-char string into a 20-char array called arr using the following instruction:

```
gets(arr);
```

a. You only read the first 20 characters of the 50-char string
b. You only read the first 19 characters of the 50-char string and a '\0' goes into the 20th position
c. You read everything and risk crashing your program

What is the value of sum after the following instructions?

```
sum = 0;
for (i=0; i<5; i++) {
   sum = sum + i;
   if (i < 3) continue;
}
```

a. 6
b. 10
c. 15
d. 3
e. 5

"continue" means "jump to the end of the loop". As there are no other instructions between "continue" and the end of the loop, it's completely useless, when the condition is false we do the same thing and just add 0+1+2+3+4

What is the output of the following code snippet?

```
int main() {
   int var;
   var = 5/2;
   printf("%d",var);
   return 0;
}
```

a. 2.5
b. 2.0
c. 3.0
d. None of the above

The division of two integers (5/2) yields an integer (2). We assign it to an integer, that we print (with %d) as an integer. So the program will show a plain "2".

What does the following snippet display:

```
char  s[5] = "abc";
char *p = s;
char  q;

q = 1 + *(p+1);
printf("%s\n", p);
```

a. bbc
b. abc
c. bc
d. cc
e. abd
f. acc

We make p the same address as s. Variable q stores the letter at the address that follows the letter pointed to by p (p points to 'a', so it's 'b') and adds 1 to it, so q stores value 'c' - but doesn't modify anything. Printing p is the same as printing s.

Not quite related to what we have seen (although there are formats in C for printing values in base 8, %o, or in base 16, %x) but basic knowledge. In base 10 the greatest digit in 9, in base 2 it's 1, so ...

The highest digit in any number system equals:
a. Zero
b. Base - 1
c. Base + 1

What is the best choice to print the value of variable x where:

```
int x = 123;
```

a. `printf("%d",x);`
b. `printf("%c",x);`
c. `printf("%f",x);`

%c would print the character that corresponds to code 123 (an opening curly bracket); %f would print x with a dot and 0s behind the dot. As x is an int, %d is the natural and best choice.

When you declare a variable as follows:

```
struct {
        char name[20];
        float value;
       } x;
```

what is reserved in memory is only a pointer.
a. True
b. False

This is a structure variable definition (not to be confused with a type definition, that starts with typedef). It reserves an array of 20 chars and a float. And it doesn't declare any pointer.

What can you say of the following statements:

```
int var = 10;
int *ptr = &(var + 1); //statement 1
int *ptr2 = &var;      //statement 2
&var = 40;             //statement 3
```

a. Statement 1 and 2 are wrong
b. Statement 2 and 3 are wrong
c. Statement 1 and 3 are wrong
d. All three statements are wrong
e. All three statements are correct

Statement 1 is wrong because var + 1 is a result and has no address. Statement 3 is wrong because & is used for returning the address, not setting it, and anyway changing the address of a variable makes no sense.