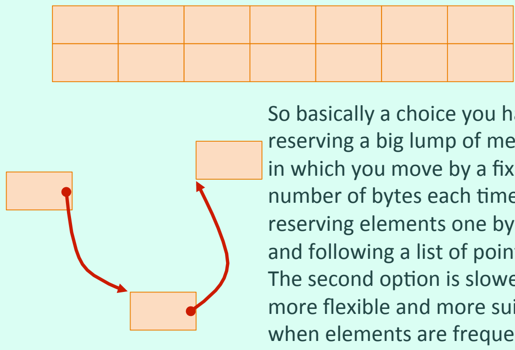
CS205

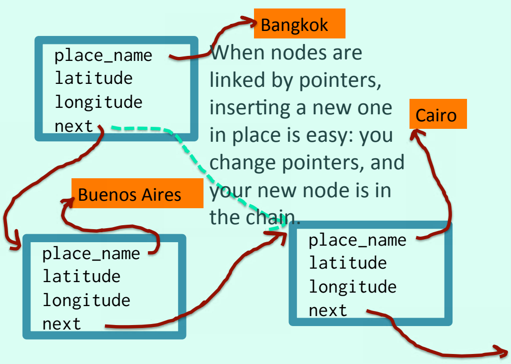
C / C++

Stéphane Faroult
faroult@sustc.edu.cn

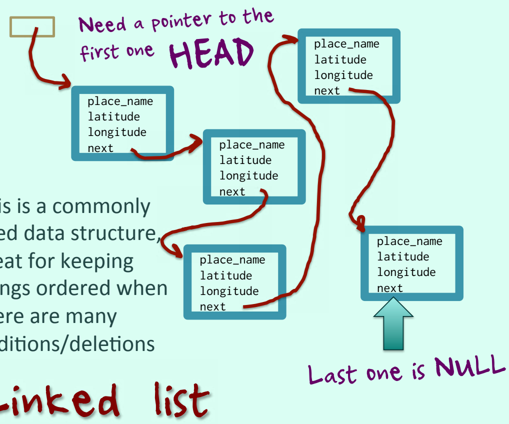
Wang Wei (Vivian) vivian2017@aliyun.com



So basically a choice you have is reserving a big lump of memory in which you move by a fixed number of bytes each time, or reserving elements one by one and following a list of pointers. The second option is slower but more flexible and more suitable when elements are frequently inserted/deleted



When nodes are linked by pointers, inserting a new one in place is easy: you change pointers, and your new node is in the chain.



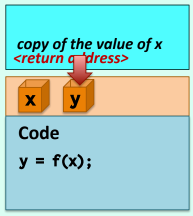
This is a commonly used data structure, great for keeping things ordered when there are many additions/deletions

Linked list

Maintaining a list in order requires changing pointers; and if you have functions for adding a new node, or removing a node, then you must be careful on how you handle pointers in your functions.

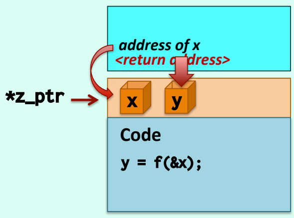
One very important thing when working with pointers - using them in functions.

When you call a simple `int` function that takes an `int` parameter, when you call the function with the parameter the value of the parameter is copied on the stack. The function picks it there, and assigns the value to a local variable called (here) z. The function won't be able to change the value of x.

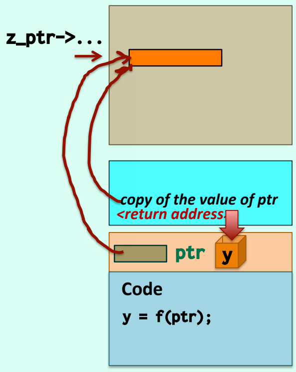



```
int f(int z) {
  ...
}
```

If you want a function to be able to modify a value (`scanf()` is an example), you must pass to the function the address of the variable. The function will get a copy of the address, and by dereferencing it will be able to change the original variable.



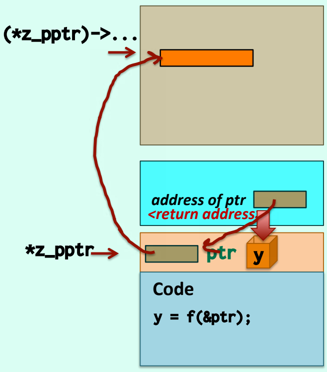

```
int f(int *z_ptr) {
  ...
}
```



When you pass an address, it can also be a copy of an address in the heap returned by `malloc()` (the kind of pointer we have when dealing with nodes).

```
int f(XYZ *z_ptr) {
  ...
}
```

XYZ is the name of a struct



Now, what if we want to CHANGE pointers? We must pass their address, a pointer to a pointer (headache inducing). Dereferencing once gives the pointer, dereferencing twice gives the value(s).

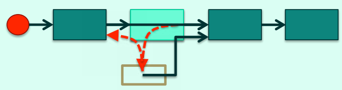
```
int f(XYZ **z_pptr) {
  ...
}
```

If you want to change a THINGAMAGIG, the function must take a THINGAMAGIG * as argument (and dereference it).

If you want to change a THINGAMAGIG *, the function must take a THINGAMAGIG ** as argument (and dereference it).

Easy to add a node at the right place

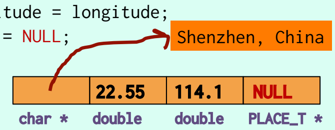
Easy to remove a node



Removing a node means saving to a pointer variable the address of the next, identifying the preceding node, freeing the node, then copying the address of the node to the previous one in order not to break the chain.

```
PLACE_T  *new_place(char  *place_name, "Shenzhen, China"
                    double latitude,    22.55
                    double longitude) { 114.1
   PLACE_T *p = NULL;

   if (place_name) {
      if ((p = (PLACE_T *)malloc(sizeof(PLACE_T)))
                != NULL) {
         p->place_name = strdup(place_name);
         p->latitude = latitude;
         p->longitude = longitude;
         p->next = NULL;
      }
   }
   return p;
}
```



Inserting a node

In a list list head needed

PLACE_T *

First node ever? Need to modify

Smallest node so far ? the head

Inserting a node

In a list list head needed

The head is a pointer, we need to modify this pointer, so we must pass its address and deal with a pointer on a pointer.

PLACE_T *

we need a POINTER to that

Inserting a node

In a list pointer to list head needed

~~**PLACE_T ***~~

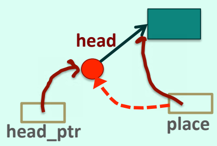
PLACE_T **

```c
void insert_place(PLACE_T **head_ptr, PLACE_T *place) {
    PLACE_T  *p;
    PLACE_T **prev_ptr = NULL;

    if (head_ptr != NULL) {
       if ((p = *head_ptr) == NULL) {
          *head_ptr = place;
       }
```

Slightly paranoid test

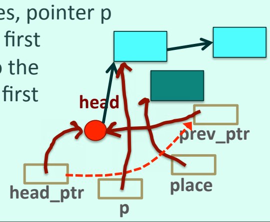
If initially the head pointer is null, we just set it to the address of the node we want to add to the list.




```c
void insert_place(PLACE_T **head_ptr, PLACE_T *place) {
    PLACE_T  *p;
    PLACE_T **prev_ptr = NULL;

    if (head_ptr != NULL) {
       if ((p = *head_ptr) == NULL) {
          *head_ptr = place;
       } else {
          prev_ptr = head_ptr;
```

If the list already contains nodes, pointer p is initially made to point to the first node and we set the pointer to the pointer (still following?) to the first node to be the same as the head pointer.

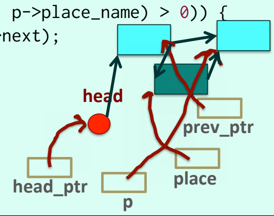



```c
void insert_place(PLACE_T **head_ptr, PLACE_T *place) {
    PLACE_T  *p;
    PLACE_T **prev_ptr = NULL;

    if (head_ptr != NULL) {
       if ((p = *head_ptr) == NULL) {
          *head_ptr = place;
       } else {
          prev_ptr = head_ptr;
          while (p && (strcmp(place->place_name,
                               p->place_name) > 0)) {
             prev_ptr = &(p->next);
             p = p->next;
          }
          place->next = p;
          *prev_ptr = place;
       }
    }
}
```

Loop as long as the end of the list isn't reached and the name is greater

Then set pointers.



Or we can go recursive

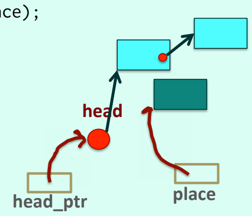
List operations can also be written in a recursive way. For recursion we need a trivial case (the empty list) and a "smaller" problem related to the current one. If you consider that in a node the "next" pointer can be considered like the head of a smaller list, then you have your smaller problem with fewer and fewer nodes in the list until you reach the "null" pointer at the end of the original list, which can be seen as an empty list.

```
void insert_place(PLACE_T **head_ptr, PLACE_T *place) {

    if (head_ptr != NULL) {
       if (*head_ptr == NULL) {
          *head_ptr = place;
       } else {
          if (strcmp(place->place_name,
                     (*head_ptr)->place_name) > 0)) {
             insert_place(&((*head_ptr)->next),
                          place);
          }
```

Trivial case (same as before)

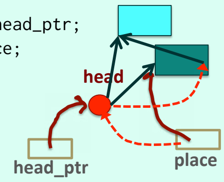
Insert into the "next" list if bigger than the first node




```
void insert_place(PLACE_T **head_ptr, PLACE_T *place) {

    if (head_ptr != NULL) {
       if (*head_ptr == NULL) {
          *head_ptr = place;
       } else {
          if (strcmp(place->place_name,
                     (*head_ptr)->place_name) > 0)) {
             insert_place(&((*head_ptr)->next),
                          place);
          } else {
             place->next = *head_ptr;
             *head_ptr = place;
          }
       }
    }
}
```

Otherwise the place is found. Change pointers.



Displaying the list in order is also very easily done by recursion: if the list is not empty, print the first node, then print the smaller list that follows.

```
void show_places(PLACE_T *p) {
    if (p) {
       printf("%s (%lf, %lf)\n", p->place_name,
                                 p->latitude,
                                 p->longitude);
       show_places(p->next);
    }
}
```

Recursive call!

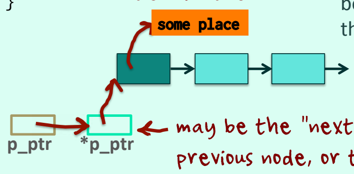
Obviously when the list is empty there is nothing to do ...

Similar story when you release memory: you first free the list that follows, then deal with the current one, then set the pointer to the freed node to null.

```
void delete_places(PLACE_T **p_ptr) {
    if ((p_ptr != NULL) && (*p_ptr != NULL)) {
       delete_places(&((*p_ptr)->next));
       free((*p_ptr)->place_name);
       free(*p_ptr);
       *p_ptr = NULL;
    }
}
```

Recursive call!

We are modifying pointers here (hence **) and the head will be reset to NULL by the function.



may be the "next" field of the previous node, or the head

Different ways to use a list

Ordered by value

Appended

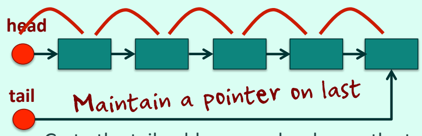
Lists are much used because they are rather versatile. They are great for maintaining in order a dynamic collection ("dynamic" means many additions/deletions) but also for keeping things say in order of arrival; same thing as a line of people waiting for their turn at the bank or to pay at the cashier. Every new arrival is appended at the end of the list.

Different ways to use a list

Ordered by value

Appended

In such a case hopping from node to node from the head to the last one is often seen as a waste of time, and a second pointer is kept to the last node in the list.



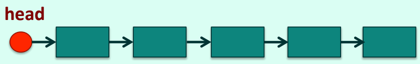
Go to the tail, add your node, change the tail pointer.

Which strategy are we going to apply to remove nodes from the list?

LIFO

Last In First Out

Different strategies can be applied for processing elements in a list. One is known as LIFO (Last In, First Out)

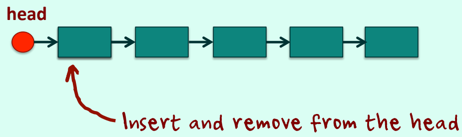


This strategy is useful when the list represents dependencies. Think of prerequisites for courses: I want to take XYZ301. But it demands ABC201, which wants DEV101 - I'll take this one first.

Which strategy are we going to apply to remove nodes from the list?

Last In First Out

In such a case you can insert the nodes at the head of the list, then remove them from the head to process them. "So the last will be first, and the first last."

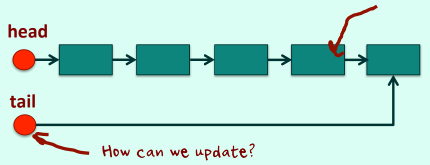


Which strategy are we going to apply to remove nodes from the list?

Last In
First Out

You could also think of adding at the end, but then when removing the node changing pointers would be difficult.

How can we set to NULL?



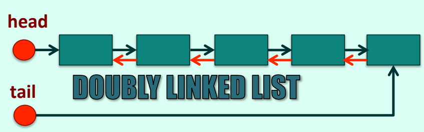
How can we update?

Which strategy are we going to apply to remove nodes from the list?

Last In
First Out

```
typedef struct place {
         char   *place_name;
         double latitude;
         double longitude;
         struct place *next;
         struct place *prev;
       } PLACE_T;
```

Unless you also keep a pointer to the previous node, which is useful in some applications.

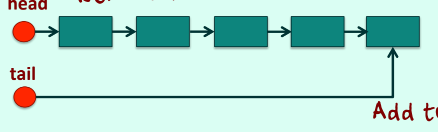


Which strategy are we going to apply to remove nodes from the list?

FIFO
FIrst In
First Out

The other famous strategy is known as FIFO (or "first come, first served"). Newcomers are added to the tail, nodes to process are taken from the head.

Remove from the head



Add to the tail

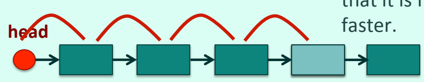
FUN variations

These are the basic strategies. You can imagine other things, such as having different priorities ("urgent" node to process), or making a list self-order itself.

Managing priorities

Self-managing lists

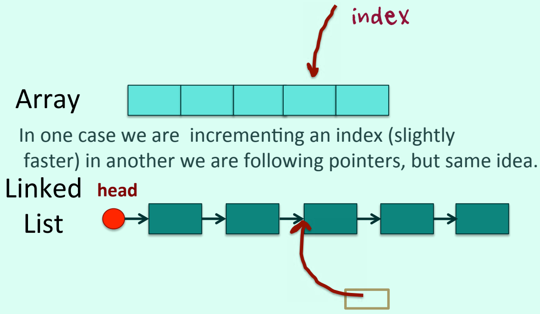
If a node is often looked for, you move it forward so that it is found faster.



We have compared lists to arrays and lists are better when you want to keep in a given order nodes that are often inserted and deleted. Arrays are good for data that doesn't change.

What about searches?

One way to search is to move up the list or the array until the value is the one we are looking for or is bigger (which means that the value we are looking for is absent)



But we can do better with arrays

BINARY SEARCH

Find New York

0	Aberdeen
1	Boston
2	Chihuahua
3	Edinburgh
4	Istanbul
5	Liverpool
6	London
7	New York
8	Reykjavik
9	Rio de Janeiro
10	Shanghai
11	Tokyo

12 elements

Middle = index 5

Search 6 to 11

Search this

If you know the number of elements in your array, you can check the middle element, then decide which half of the array to search.

Find New York

0	Aberdeen
1	Boston
2	Chihuahua
3	Edinburgh
4	Istanbul
5	Liverpool
6	London
7	New York
8	Reykjavik
9	Rio de Janeiro
10	Shanghai
11	Tokyo

12 elements

Middle = index 5

Search 6 to 11

New middle = index 8

} Search this

By narrowing each time the number of elements to search, you focus very fast on the "area" where what you are looking for should be.

Find New York

0	Aberdeen
1	Boston
2	Chihuahua
3	Edinburgh
4	Istanbul
5	Liverpool
6	London
7	New York
8	Reykjavik
9	Rio de Janeiro
10	Shanghai
11	Tokyo

12 elements

Middle = index 5

Search 6 to 11

New middle = index 8

Search 6 to 7

New middle = index 6

Search 7 to 7

Benefits

So you see that we should try to do far better for searches in a data structure using pointers, because here arrays are really good.

Linear search in an ordered list of N — N/2 comparisons (average)

Binary search in an ordered list of N — 1 among N

1 among N/2

1 among N/4

...

1 among $N/2^n$

with $N \sim 2^n$

What happens when we double N?

Lists are good for maintaining data that changes, not so good for searches.

Arrays are not good for keeping in order data that changes, but good for searches.

Nothing prevents from combining data structures, and try to get some advantages of both.

For instance:

Hash table

A combination of array and linked list

Idea:

Make storage location
dependent on value.

"If this is this value, then it must go here".
Think of a bookstore - history book => here

Ordering is relative.

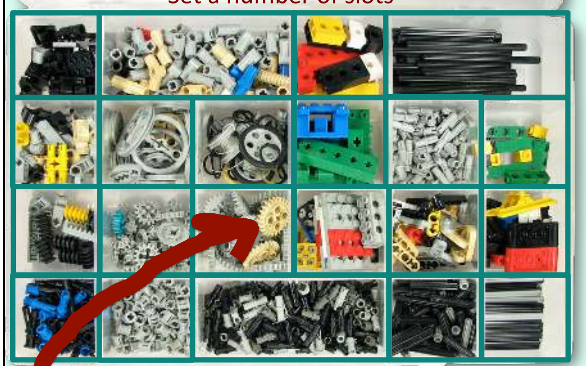
History books may be ordered, but not
history books vs novels or travel books.

Make location absolute.

You know where shelves are.

Flickr: Windell Oskay

Set a number of slots



Find a function (hash function) to associate a
slot with an item

```
int hash_func(char *str, int slots) {
    int hashval = 0;
    int i = 0;

    // Not necessarily a good hash function
    if (str) {
       while (str[i] != '\0') {
          hashval += (int)str[i];
          i++;
       }
       hashval %= slots;
    }
    return hashval;
}
```

This is an example of hash function; if I have an array with "slots" positions and am trying to store strings such as `str` into it, I can compute a number by adding the codes of the letters in `str`, and take the modulo to get a number between 0 and slots - 1, the computed position.

You can compute far better hash functions by using functions used in cryptography (it means "hidden writing" in Greek) that transform anything into a short code, and two almost identical values into two very different codes (these functions are used for security features - you should never store passwords, but their hash value, and check that the hash value of what was supplied matches the stored hash value)

Better functions: **libssl**

MD5
```
#include <openssl/md5.h>
```

SHA1
```
#include <openssl/sha.h>
```

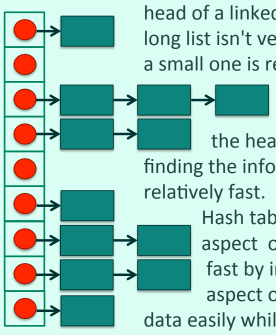
Take hash value % number of slots in the array

PROBLEM

Different items can hash into the same value.

With a function such as the preceding bad example (adding letter values), the result will be the same for anagrams, such as "eager" and "agree". As soon as we have more pieces of data than slots, whatever the function, we'll have Conflicts, pieces of data landing at the same slot

- Make each slot a linked list of items hashing to the same value.



In a hash table, each slot will be the head of a linked list. If searching a long list isn't very efficient, searching a small one is reasonably quick. Computations will give us the slot where the head of the list is, then finding the information will be relatively fast.

Hash tables combine one good aspect of arrays (locating data fast by index) with one good aspect of linked lists (adding data easily while keeping everything organized).

Finding the good function

Correctly sizing the array

In practice, finding the right size and the right function that minimizes conflicts may require some testing.

Access time depends on length of each list.

The same is true for "simple" linked lists and linked list in hash tables: if many values hash to the same slot and if some lists are very long, speed to find the information will suffer.

Binary Search

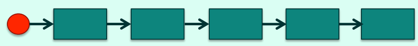
0	Aberdeen
1	Boston
2	Chihuahua
3	Edinburgh
4	Istanbul
5	Liverpool
6	London
7	New York
8	Reykjavik
9	Rio de Janeiro
10	Shanghai
11	Tokyo

You have seen that the time to search a list depends on the number of items in the list, and that time may also vary in a poorly built hash-table. Contrast this with the binary search, in which it takes always more or less the same time to find data, even if you double the number of items to search.

TREES

There is one family of data structures that is very much used because it allows a search similar to the binary search in an array, which is the family of "trees".

```c
typedef struct place {
           char   *place_name;
           double latitude;
           double longitude;
           struct place *next;
        } PLACE_T;
```



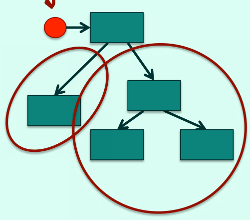
In a linked list each node contains a pointer to the next element. If the list is used as an ordered list, it will be a pointer to the node containing the next greater value.

```c
typedef struct place {
           char   *place_name;
           double latitude;
           double longitude;
           struct place *left;
           struct place *right;
        } PLACE_T;
```

In a (binary) tree each node contains TWO pointers to other nodes, each one the beginning of a subtree. The subtree to the left may contain all smaller values, and the subtree to the right all greater values. The first node in the tree no longer contains the smallest value.

```c
typedef struct place {
           char   *place_name;
           double latitude;
           double longitude;
           struct place *left;   <— Smaller nodes
           struct place *right;  <— Bigger nodes
        } PLACE_T;
```

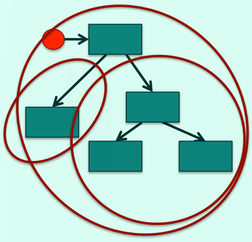
root pointer



The first node (the one at the top) of the tree is no longer called "head" but "root".

```c
typedef struct place {
           char   *place_name;
           double latitude;
           double longitude;
           struct place *left;
           struct place *right;
        } PLACE_T;
```

As the subtrees are just like the whole tree, only smaller, trees are data structures made for recursion.

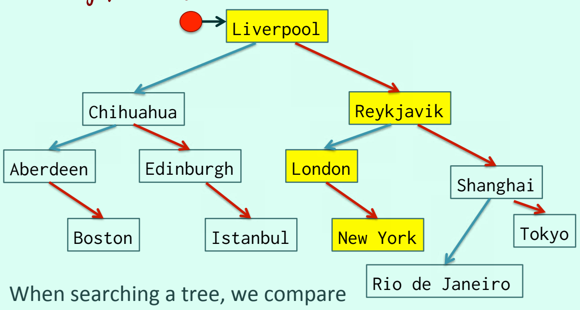


Ready for recursion?

```
Aberdeen
Boston
Chihuahua
Edinburgh
Istanbul
Liverpool
London
New York
Reykjavik
Rio de Janeiro
Shanghai
Tokyo
```

When performing a binary search in an array we were first looking at the middle value - not the smallest one; This middle value could very well be the one at the top of a tree.

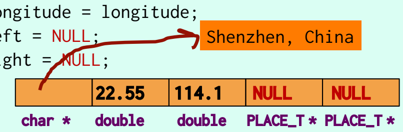
Looking for "New York"



When searching a tree, we compare the value that we are looking for to the value in the current node. If they are different, if the current node value is bigger we search to its left, otherwise to its right.

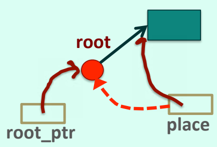
```
PLACE_T  *new_place(char  *place_name,   "Shenzhen, China"
                    double latitude,      22.55
                    double longitude) {   114.1
   PLACE_T *p = NULL;
   if (place_name) {
      if ((p = (PLACE_T *)malloc(sizeof(PLACE_T)))
               != NULL) {
         p->place_name = strdup(place_name);
         p->latitude = latitude;
         p->longitude = longitude;
         p->left = NULL;
         p->right = NULL;
      }
   }
   return p;
}
```

Creating a tree node is similar to creating a list node, with two pointers



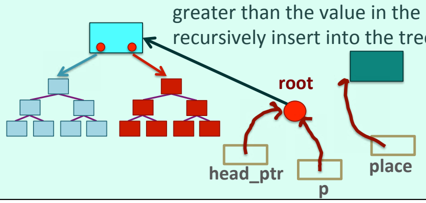

```
void insert_place(PLACE_T **root_ptr, PLACE_T *place) {
    PLACE_T *p = NULL;
    if (root_ptr != NULL) {
       if ((p = *root_ptr) == NULL) {
          *root_ptr = place;
       }
```

As with a list, inserting a node into an empty tree just means making the root pointer take the value of the node address. We may need to modify the root pointer, therefore we must use its address and get a pointer to a pointer (**)




```
void insert_place(PLACE_T **root_ptr, PLACE_T *place) {
    PLACE_T *p = NULL;
    if (root_ptr != NULL) {
       if ((p = *root_ptr) == NULL) {
          *root_ptr = place;
       } else {
          if (strcmp(place->place_name,
                     p->place_name) > 0) {
             insert_place(&(p->right), place);
          }
```

If the tree isn't empty, if the current value is greater than the value in the node, we recursively insert into the tree to the right.



```c
void insert_place(PLACE_T **root_ptr, PLACE_T *place) {
    PLACE_T *p = NULL;
    if (root_ptr != NULL) {
       if ((p = *root_ptr) == NULL) {
          *root_ptr = place;
       } else {
          if (strcmp(place->place_name,
                     p->place_name) > 0) {
             insert_place(&(p->right), place);
          } else {
             if (strcmp(place->place_name,
                        p->place_name) < 0) {
                insert_place(&(p->left), place);
             }
          }
       }
    }
}
```

Otherwise we recursively insert into the tree to the left, until we find an empty subtree and make it point to our new node.

```c
void show_places(PLACE_T *p) {
    if (p) {
       show_places(p->left);
       printf("%s (%lf, %lf)\n", p->place_name,
                                 p->latitude,
                                 p->longitude);
       show_places(p->right);
    }
}
```

Recursive calls!

To print the values in the tree, we can first print all the smaller values than the current node value, then the current node value, then all the greater values: everything will be printed in order.

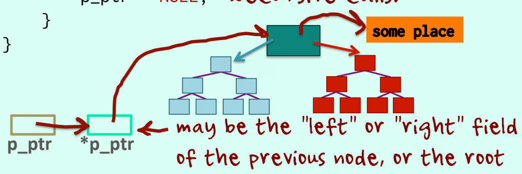
We can :
Display what is smaller, the node, what is greater.
Or display the node, what is smaller, what is greater.
Or display what is smaller, what is greater, then the node.

And of course display what is greater before what is smaller in all cases.

For freeing memory, we recursively free the two subtrees, then free memory allocated for the current node. The node pointer is modified, so we must use its address.

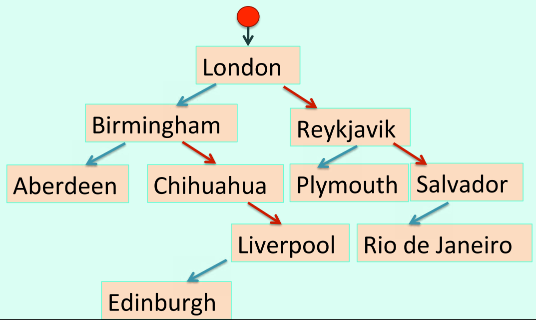
```c
void delete_places(PLACE_T **p_ptr) {
    if ((p_ptr != NULL) && (*p_ptr != NULL)) {
       delete_places(&((*p_ptr)->right));
       delete_places(&(*p_ptr)->left));
       free((*p_ptr)->place_name);
       free(*p_ptr);
       *p_ptr = NULL;
    }
}
```

Recursive calls!



There is one basic issue with trees.

If we insert nodes in more or less random order, we will get a nicely balanced tree that will be fast to search.



What if I were reading data from this file?

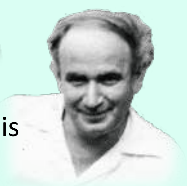
Aberdeen
Birmingham
Chihuahua
Edinburgh
Liverpool
London
Plymouth
Reykjavik
Rio de Janeiro
Salvador

The tree would degenerate into a linked list, slower to search.

AVL tree

Two russian mathematicians, Adelson-Velsky and Landis (initials: AVL), found in the early 60s a very clever way to keep trees balanced.

Adelson-Velsky



Landis

Balance Factor

They introduced a technical piece of information which is the difference in height between subtrees.

(Height of left subtree) – (Heigh of right subtree)

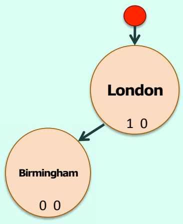
-1
0
1

OK

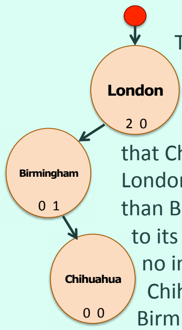
If adding one node equilibrates or just brings an imbalance of one level, there is no problem.

else rotate

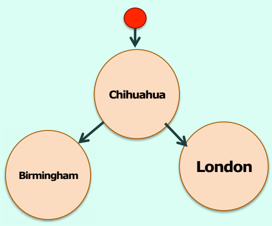
If the new node brings an imbalance of two levels, then you must "rotate" the tree.



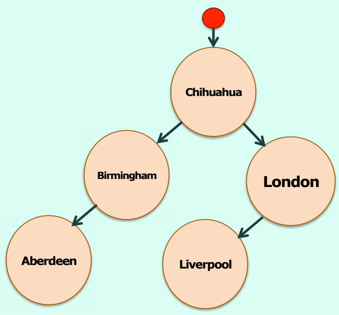
When adding "Birmingham", there is no problem, for node "London" the left subtree has a height of 1, there is nothing to the right so the height is 0 and the difference is one.



The addition of Chihuaha brings an imbalance problem. To rebalance the tree, we may notice that Chihuaha is "smaller" than London, so to its left, but "bigger" than Birmingham, so to its right. There would be no imbalance problem if Chihuahua, being between Birmingham and London, were the root.



So we can move up Chihuahua to the root, change pointers, and rebuild the tree as a balanced tree.

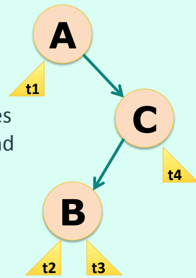


Then we will insert new nodes until we have a new imbalance problem.

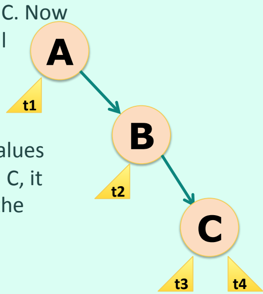
Only very few cases of rotations

By studying trees, Adelson-Velsky and Landis discovered that in fact a very limited number of "rotations" (besides, symmetrical pairs of rotations) could take care of all cases of imbalance and keep trees nicely balanced as data is inserted into them.

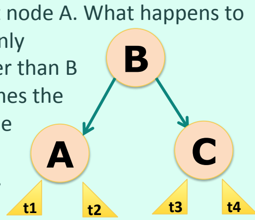
This is just one example of how rebalancing can be performed in a complex tree.
Notice that subtree t2 contains values greater than A but smaller than B, and subtree t3 values greater than B but smaller than C.



The first step consists in moving B one level up, between nodes A and C. Now the subtree to the right of B will be the one starting at C. What can we do with subtree t3, which originally was to the right of B? As it only contains values greater than B but smaller than C, it goes to the left of C, replacing the pointer to B itself.



Then we rotate the tree, by making B become the root, and its left subtree will start at node A. What happens to its original left subtree t2? It only contains values that are smaller than B but greater than A, so it becomes the right subtree of A, replacing the pointer to B itself, and we end up with a lovely balanced tree.



We have rotated nodes in a counter-clockwise move, the symmetrical (clock-wise) rotation can also happen. You have to be very careful with your pointers, but it works fine.

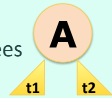
A tree isn't necessary binary

You have tons of variations on trees. Some trees contain several values per node (and more than two pointers), to try to limit height.

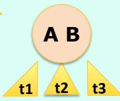
Example : 2-3-4 tree

Allow:

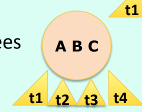
One value per node, two subtrees



Two values per node, three subtrees



Three values per node, four subtrees



When nodes are full, you split them.

Another extremely famous case of non-binary tree is the B-tree, which was designed for recording the location of values in file records and finding very fast where this value or that value is located.

Variants of the B-tree algorithm are today used in almost all database software.

B-Tree

This algorithm was invented by two researchers at Boeing in the early 1970s.

BOEING early 1970s

Rudolf Bayer
(born 1939)

Edward McCreight
(born 1946)

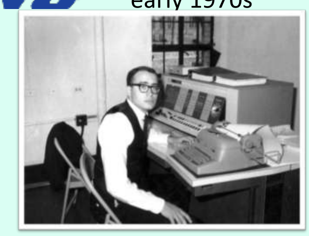
Balanced?

B-Tree

At that time they looked more like these pictures (the picture of McCreight is a college picture from about 1965). They never wanted to say what B was standing for.

?
BOEING early 1970s

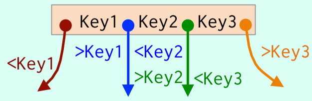
?
Rudolf Bayer
(born 1939)



Edward McCreight
(born 1946)

B-Tree

Several keys per node, N max

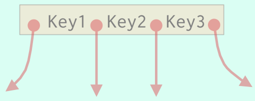


In a B-Tree node you have at most N key values. You also have one more pointer than you have keys, with each pointer being the root of a tree that contains values between the values of two keys in the node.

B-Tree

There is also a requirement about not wasting too much space in nodes.

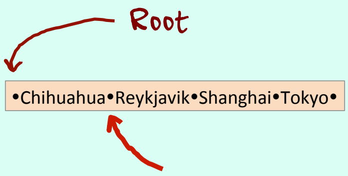
Several keys per node, N max



Keep every node except the root **at least** half-full

Liverpool

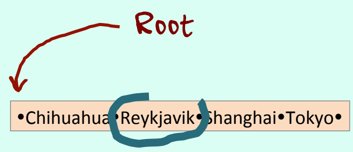
Let's look at an example. We already have the maximum number of keys in our first node and want to insert a new key.



Liverpool would go there if there were room

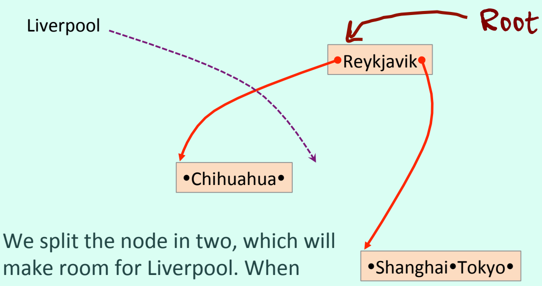
Max: 4 keys per node *(Min: 2 keys per node)*

Liverpool



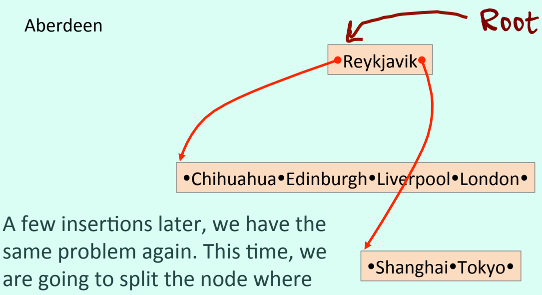
In that case we select the value that would be in the middle if there were room for Liverpool.

Max: 4 keys per node *(Min: 2 keys per node)*



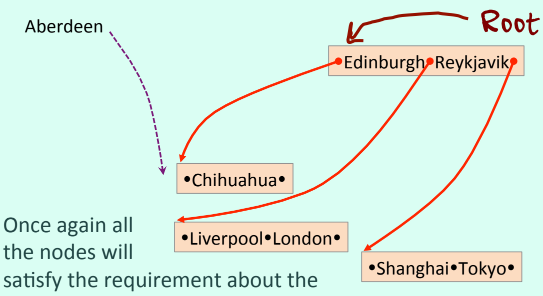
We split the node in two, which will make room for Liverpool. When Liverpool is inserted, every node (except the root) contains the minimum number of keys, and we have room for inserting new keys.

Max: 4 keys per node *(Min: 2 keys per node)*



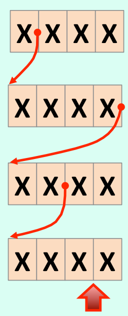
A few insertions later, we have the same problem again. This time, we are going to split the node where Aberdeen should go (and where Edinburgh would be in the middle). The node is split and Edinburgh is pushed up into the root, where there is room.

Max: 4 keys per node *(Min: 2 keys per node)*



Once again all the nodes will satisfy the requirement about the minimum number of keys per node.

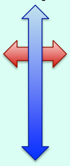
Max: 4 keys per node *(Min: 2 keys per node)*



If one day when we insert a key in the node we find that the node above (where the middle key is pushed) is full too, we will split recursively until the moment when we find a node with room above - or, in the worst case, we create a new root as we did when inserting Liverpool.
What is interesting is that new keys are always inserted at the bottom, the tree grows from the bottom up because of splits and is always balanced by construct.

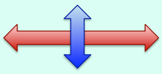
A binary tree grows from the top, and it's its height that increases fast. In the case of the B-Tree, it's the width that increases (and not so fast, as nodes contain many keys; I have given an example with four keys, but in a database it's more like one hundred).

binary Tree



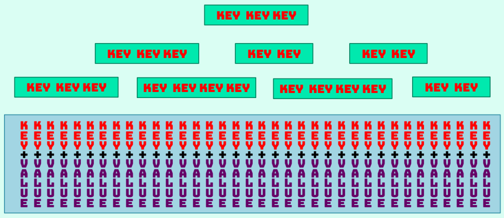
As the average speed to find an item depends on the depth of the tree, B-Trees perform very well.

B-Tree



(Although of course you have to scan the key values in each node when you search or navigate)

In practice key + value



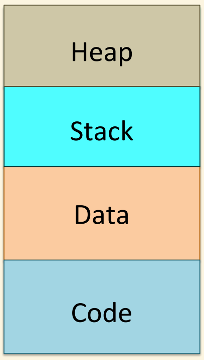
In the variant used in databases B-Trees are used to locate the file position of values in a kind of directory (as in "phone directory", not as in "computer folder") called "index".

Functions available in C

Although there isn't in C any built-in structure comparable to a Java "Collection", the data structures that have been presented are very common software patterns, much used, and several libraries contain special functions for handling linked lists and trees.

Pointers on functions

Before we proceed, we must see (because these functions use them) something that is rather advanced in C but sometimes quite useful, which is a pointer on a function.



von Neumann Architecture

Remember that in the von Neumann architecture, everything is in memory: data, but also code. Being in memory, the very first instruction in a function has a memory address. We can in C define a special type of pointer for storing the address of a function.

Note: you may have a warning here because `strcmp()` expects `char *`, not `void *`, parameters, but it works fine.

Defining a pointer to a function in C

```
int (*comp)(void *, void *);
comp = strcmp;
int cmp = (*comp)(p1, p2);
int cmp = comp(p1, p2);
```

The two statements are equivalent

The syntax for defining a function pointer is a bit weird. The telling sign is a pair of parentheses that enclose `*name` (`*comp` here, supposed to be a comparison function). The first line defines a function pointer called comp.This function must return an `int` and take two `void *` parameters. You can then assign to comp the name (you could also write `&strcmp`) of an existing function, then call it .

Because data structures rely a lot on key comparison and because keys can be any type, in libraries you are usely asked to pass a pointer to (in practice, the name of) the suitable function to use.

Pointers to comparison functions are very common in libraries

Pointers to char * **strcmp()**

Pointers to numbers?

Pointers to date structures?

Several sort keys?

As a side note, you can define in a structure an attribute that is a pointer to a function.
In fact, you can have many such pointers in a structure, and assign to them the address of existing functions when you initialize your structure.
At this point, your function pointers are beginning to look a lot like methods ...
This is how C became C++ and object-oriented, as we shall see later.