# C programming

*Data structure & algorithms*

Section 1

## Data structures

# What is data structure

A scheme for organizing related pieces of information

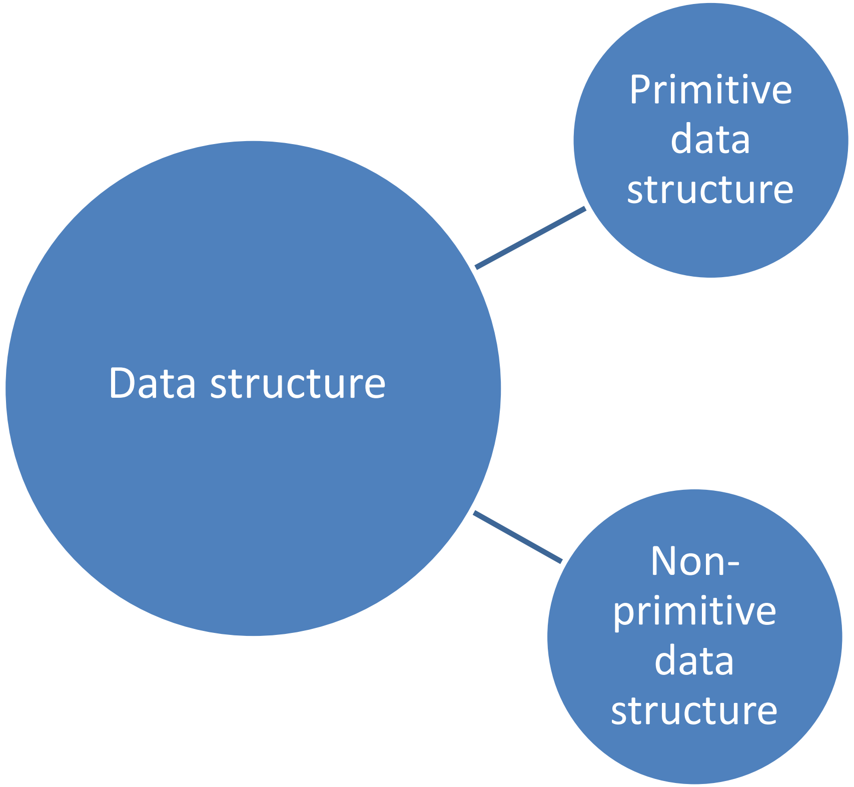
A way in which sets of data are organized in a particular system

An organized aggregate of data items

A computer interpretable format used for storing, accessing, transferring and archiving data

The way data is organized to ensure efficient processing: this may be in lists, arrays, stacks, queues or trees

## Data structure classification



- These are data structures that can be manipulated directly by machine instructions.
- In C language, the different primitive data structures are int, float, char, double.

- These are data structures that can not be manipulated directly by machine instructions. Arrays, linked lists, files etc., are some of non-primitive data structures and are classified into *linear data structures* and *non-linear data structures*

## Data structure design aspect

Data Structure = Organised Data + Allowed Operations

**There are two design aspects to every data structure:**

**the interface part**

The publicly accessible functions of the type. Functions like creation and destruction of the object, inserting and removing elements (if it is a container), assigning values etc.

**the implementation part**:

Internal implementation should be independent of the interface. Therefore, the details of the implementation aspect should be hidden out from the users.

## Example: collection

- Programs often deal with collections of items.
- These collections may be organized in many ways and use many different program structures to represent them, yet, from an abstract point of view, there will be a few common operations on any collection.

| | |
|---|---|
| **create** | Create a new collection |
| **add** | Add an item to a collection |
| **delete** | Delete an item from a collection |
| **find** | Find an item matching some criterion in the collection |
| **destroy** | Destroy the collection |

Section 2

## Data structure example

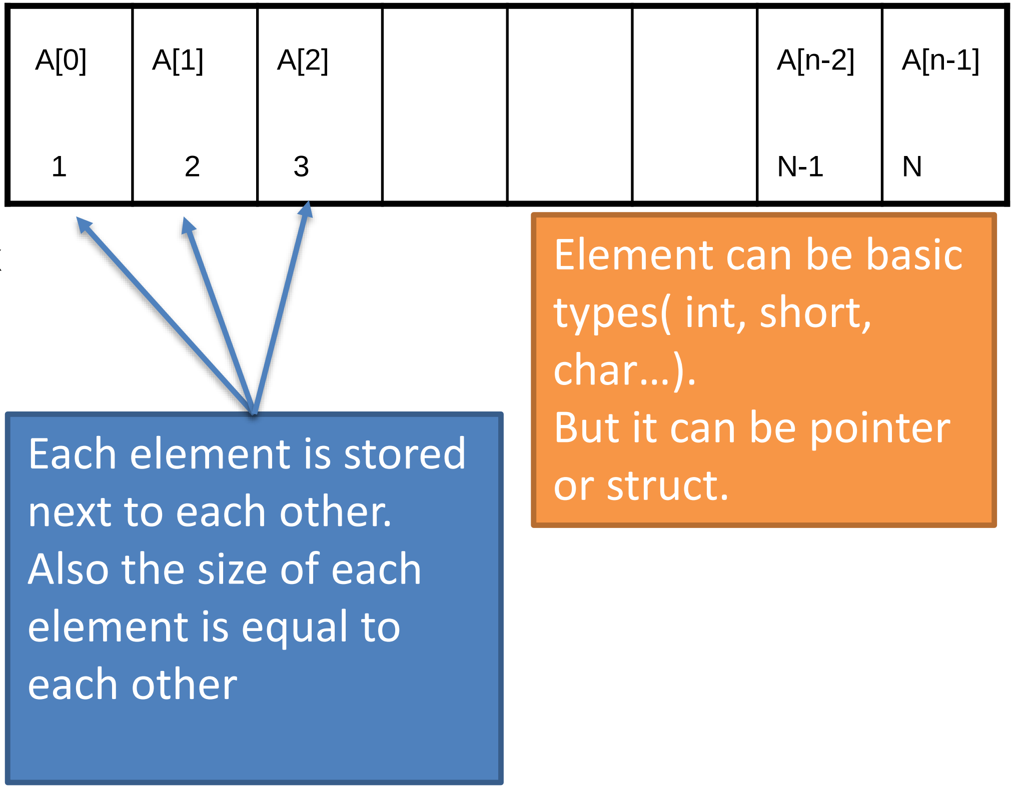
# Array

**An Array is the simplest form of implementing a collection**

- Each object in an array is called an *array element*
- Each element has the same data type (although they may have different values)
- Individual elements are accessed by index using a consecutive range of integers

**One Dimensional Array or vector**

```
int A[10];
for ( i = 0; i < 10; i++)
A[i] = i +1;  /* i is index value */
```



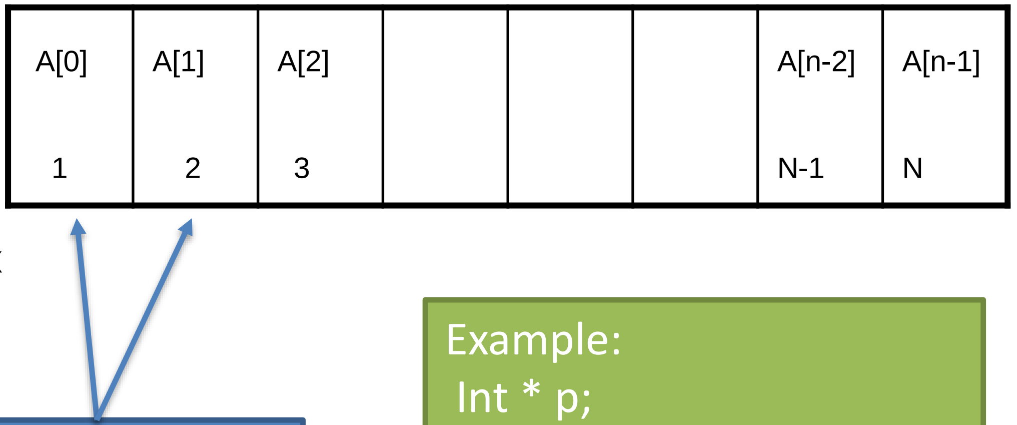
Each element is stored next to each other. Also the size of each element is equal to each other

Element can be basic types( int, short, char…). But it can be pointer or struct.

# Array

**An Array is the simplest form of implementing a collection**

- Each object in an array is called an *array element*
- Each element has the same data type (although they may have different values)
- Individual elements are accessed by index using a consecutive range of integers



**One Dimensional Array or vector**

```
int A[10];
for ( i = 0; i < 10; i++)
A[i] = i +1;  /* i is index value */
```

Because of each element have same data size. We can use pointer to access to array

Example:

```
 Int * p;
 p = &A[0]
For (i=0;i  < 10 ; i++)
{
        *(p + i) = i+1;
}
```

# Multi-dimension array

Syntax:

type name[size1][size2]...[sizeN];

```
int a[3][3] = {
   {1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6} ,   /*  initializers for row indexed by 1 */
   {7, 8, 9}   /*  initializers for row indexed by 2 */
};
```

OR

```
int a[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

$$\begin{matrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{matrix}$$

$\longleftrightarrow$ Example for storing a 3x3 matrix

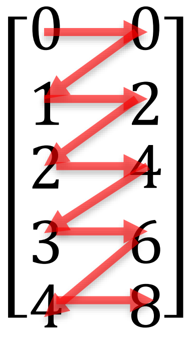
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

# Accessing multi-dimension array

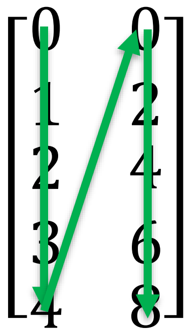
```
/* an array with 5 rows and 2 columns*/
  int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
  int i, j;

  /* output each array element's value */
  for ( i = 0; i < 5; i++ ) {

     for ( j = 0; j < 2; j++ ) {
        printf("a[%d][%d] = %d\n", i,j, a[i][j] );
     }
  }
```

Example for access 5x2 matrix Following major order

Row major order



Column major order



## Array: limitation

Simple and Fast but must specify size during construction

If you want to insert/ remove an element to/ from a fixed position in the list, then you must move elements already in the list to make room for the subsequent elements in the list.

Thus, on an average, you probably copy half the elements.

In the worst case, inserting into position 1 requires to move all the elements.

## Array: limitation

Copying elements can result in longer running times for a program if insert/ remove operations are frequent, especially when you consider the cost of copying is huge (like when we copy strings)

An array cannot be extended dynamically, one have to allocate a new array of the appropriate size and copy the old array to the new array

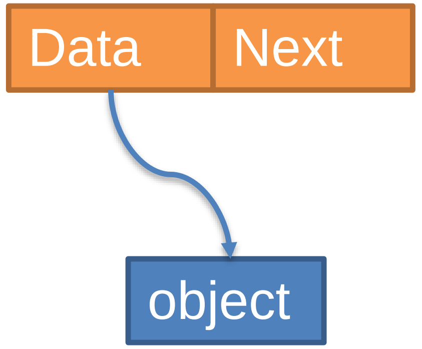
# Linked list

- **The linked list is a very flexible dynamic data structure: items may be added to it or deleted from it at will**
  - ✓ Dynamically allocate space for each element as needed
  - ✓ Include a pointer to the next item
  - ✓ the number of items that may be added to a list is limited only by the amount of memory available

Linked list can be perceived as connected (linked) nodes

Each node of the list contains

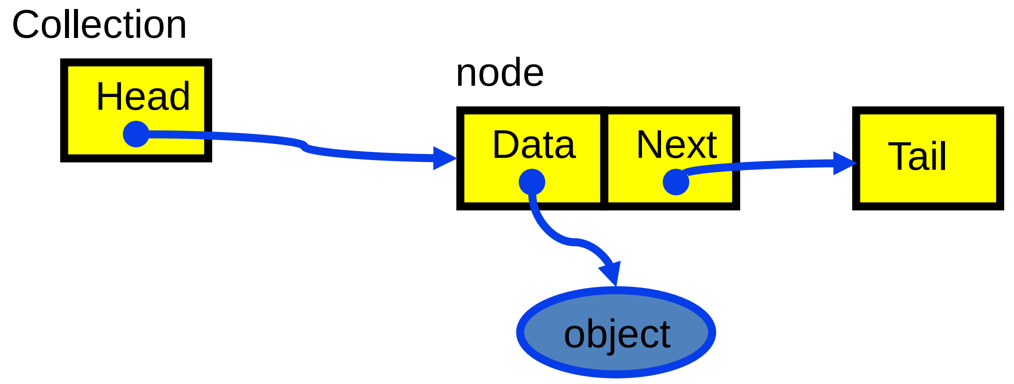
- the data item
- a pointer to the next node
- The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.



# Linked Lists

- Collection structure has a pointer to the list head
  - ✓ Initially NULL
- Add first item
  - ✓ Allocate space for node
  - ✓ Set its data pointer to object
  - ✓ Set Next to NULL
  - ✓ Set Head to point to new node

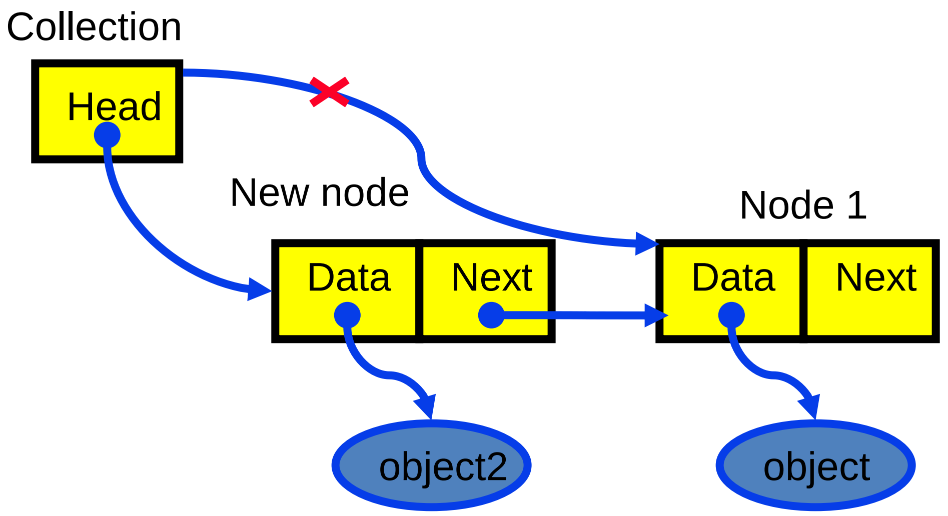
The variable (or handle) which represents the list is simply a pointer to the node at the *head* of the list.



# Linked Lists

## Add a node

- Allocate space for node
- Set its data pointer to object
- Set Next to current Head
- Set Head to point to new node



# Linked list – Add implementation

- Implementation example

Declare a pointer element point to struct type itself (C allowed it)

```
struct t_node {
   void *item;
   struct t_node *next;
   } node;
typedef struct t_node *Node;
struct collection {
   Node head;
   ……
   };
int AddToCollection( collection c, void *item ) {
   node new = malloc( sizeof( struct t_node ) );
   new->item = item;
   new->next = c->head;
   c->head = new;
   return TRUE;
   }
```

# Linked list – Find implementation

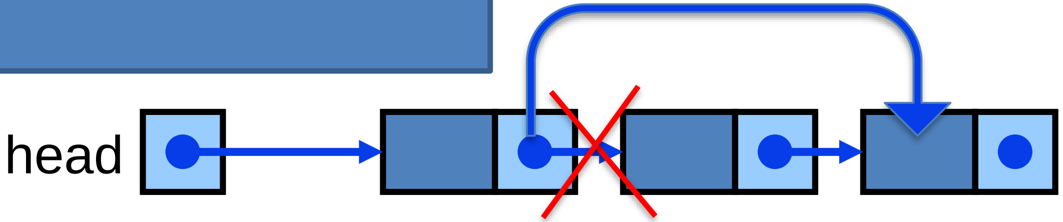
- Implementation example

```
void *FindinCollection( collection c, void *key )
{
    Node n = c->head;
    while ( n != NULL ) {
     if ( KeyCmp( ItemKey( n->item ), key ) == 0 )
     {
          return n->item;
               n = n->next;
       }
     return NULL;
     }
```

# Linked list – Delete

```
void *DeleteFromCollection( Collection c, void *key ) {
    Node n, prev;
    n = prev = c->head;
    while ( n != NULL ) {
     if ( KeyCmp( ItemKey( n->item ), key ) == 0 )
     {
         prev->next = n->next;
         return n;
      }
      prev = n;
      n = n->next;
       }
    return NULL;
    }
```

Simply remove connection between two nodes and reconnect it with other



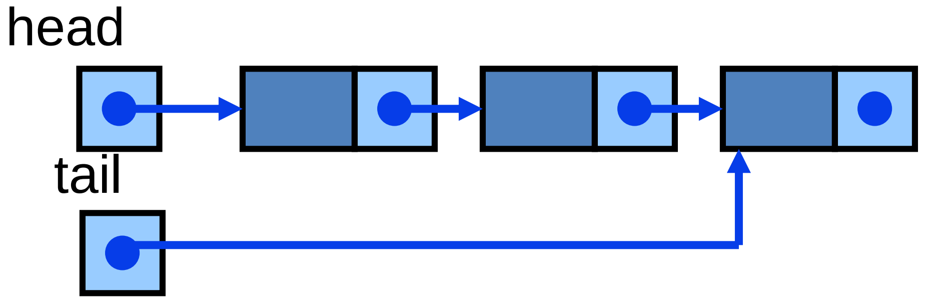
# Linked list – Variations

- ✓ Simplest implementation
  - Add to head
  - Last-In-First-Out (LIFO) semantics
- ✓ Modifications
  - First-In-First-Out (FIFO)
  - Keep a tail pointer

By ensuring that the tail of the list is always pointing to the head, we can build a circularly linked list

head is tail->next

LIFO or FIFO using ONE pointer



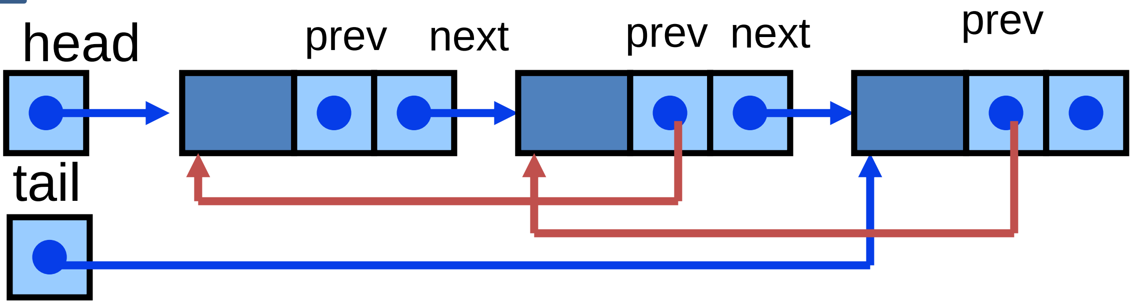
# Linked list – Doubly linked

- Doubly linked lists
  - ✓ Can be scanned in both directions

```
struct t_node {
    void *item;
    struct t_node *prev, *next;
    } node;
```

Applications requiring both way search

Eg. Name search in telephone directory

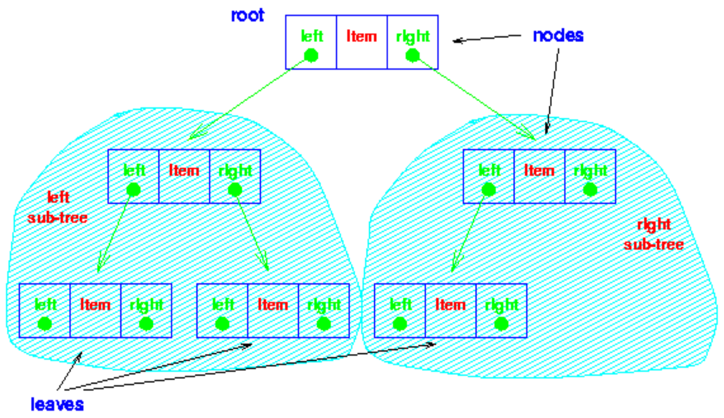


# Binary tree

- The simplest form of Tree is a Binary Tree
  - ✓ Binary Tree Consists of
    - Node (called the ROOT node)
    - Left and Right sub-trees
    - Both sub-trees are binary trees
    - The nodes at the lowest levels of the tree (the ones with no sub-trees) are called leaves

Each sub-tree is itself a binary tree

In an *ordered binary tree*
the keys of all the nodes in
- the left sub-tree are less than that of the root
- the keys of all the nodes in the right sub-tree are greater than that of the root,
- the left and right sub-trees are themselves ordered binary trees.



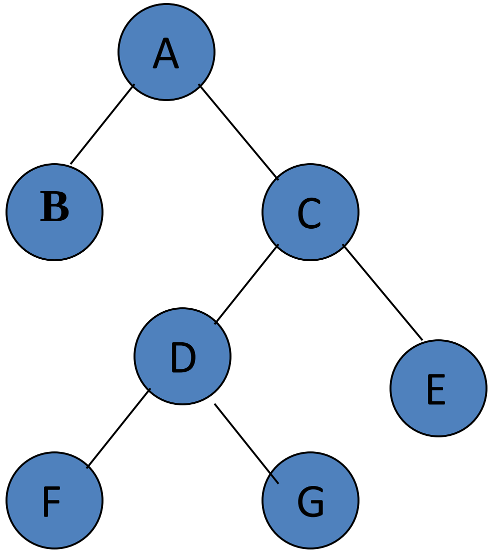
# Binary tree

Two nodes are *brothers* if they are left and right sons of the same father

If A is the root of a binary tree and B is the root of its left/right subtree then

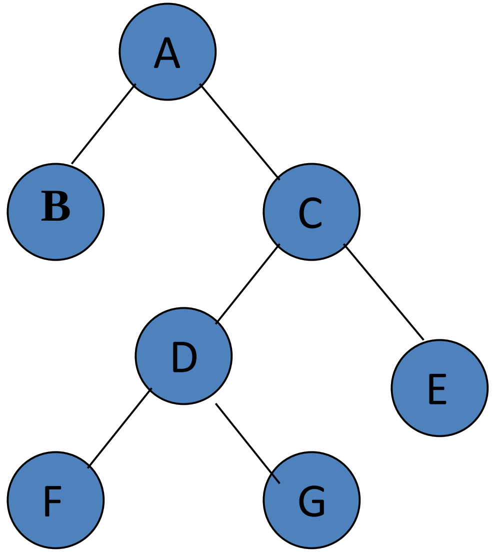
- A is the *father* of B
- B is the *left/right son* of A

Node n1 is an *ancestor* of n2 (and n2 is *descendant* of n1) if n1 is either the father of n2 or the father of some ancestor of n2



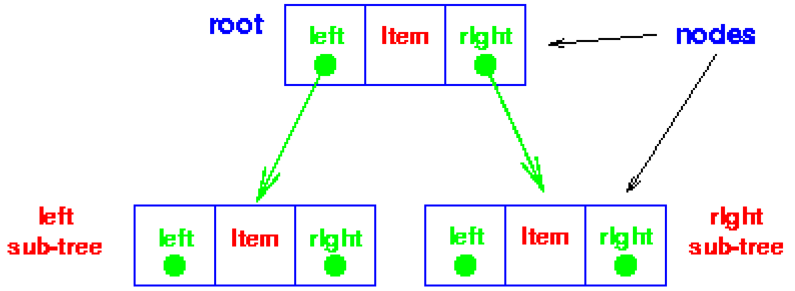
# Binary tree

- *Strictly Binary Tree*: If every nonleaf node in a binary tree has non empty left and right subtrees
- *Level* of a node: Root has level 0. Level of any node is one more than the level of its father
- *Depth*: Maximum level of any leaf in the tree
- A binary tree can contain at most $2^l$ nodes at level l
- Total nodes for a binary tree with depth d = $2^{d+1} - 1$



# Binary Tree - Implementation

```
struct t_node {
     void *item;
     struct t_node *left;
     struct t_node *right;
     };

typedef struct t_node *Node;

struct t_collection {
     Node root;
     ……
     };
```



# Binary tree - Find

```
▪  extern int KeyCmp( void *a, void *b );
▪  /* Returns -1, 0, 1 for a < b, a == b, a > b */

▪  void *FindInTree( Node t, void *key ) {
▪      if ( t == (Node)0 ) return NULL;
▪      switch( KeyCmp( key, ItemKey(t->item) ) ) {
▪          case -1 : return FindInTree( t->left, key );
▪          case 0:   return t->item;
▪          case +1 : return FindInTree( t->right, key );
▪          }
▪      }

▪  void *FindInCollection( collection c, void *key ) {
▪      return FindInTree( c->root, key );
▪      }
```

Less, search left

Greater, search right

# Binary tree - Traversing

Traverse: Pass through the tree, enumerating each node once

**PreOrder (also known as depth-first order)**

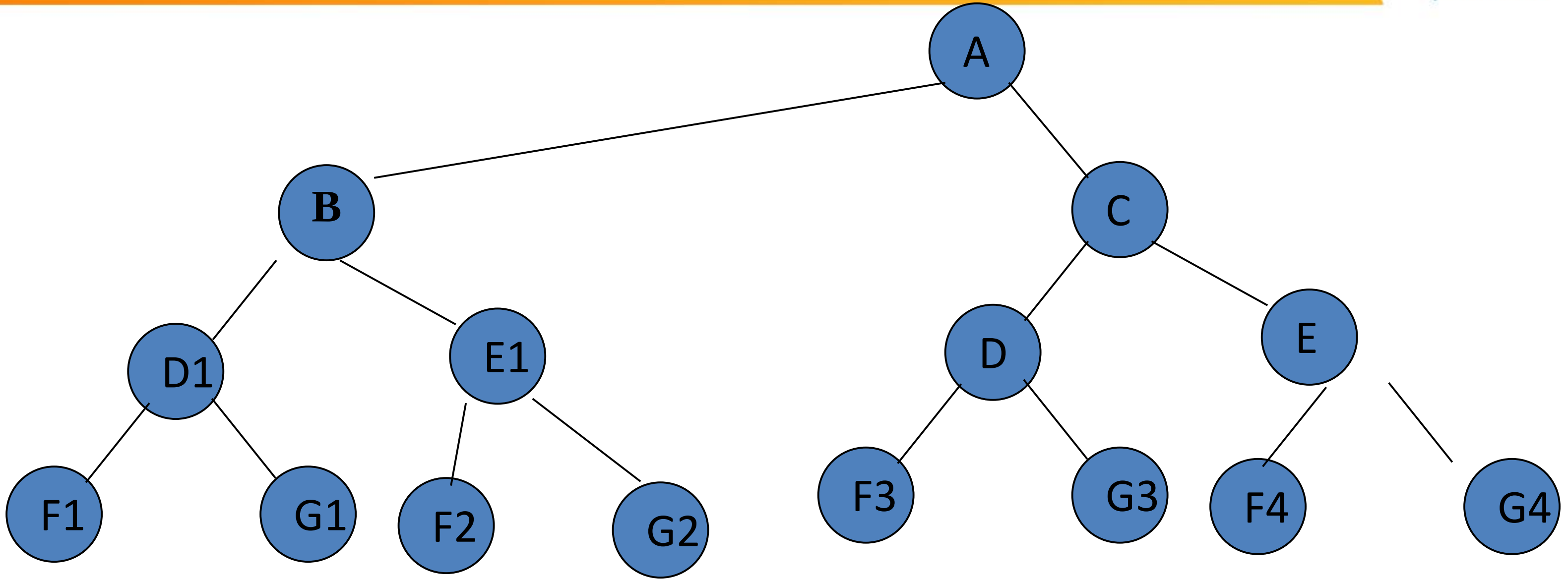
1. Visit the root
2. Traverse the left subtree in preorder
3.Traverse the right subtree in preorder

**InOrder (also known as symmetric order)**

1. Traverse the left subtree in inorder
2. Visit the root
3. Traverse te right subtree in inorder

**PostOrder (also known as symmetric order)**

1. Traverse the left subtree in postorder
2. Traverse the right subtree in postorder
3. Visit the root



PreOrder: A ->B->D1->F1->G1 -> E1 -> F2 ->G2 -> C->D->F3->G3->E->F4->G4

InOrder: F1->D1->G1->B->E1->F2->G2 ->C->D->F3->G3->F4->E->G4

PostOrder: F1->G1->D1->B->E1->F2->G2 ->C->D->F3->G3->F4->E->G4

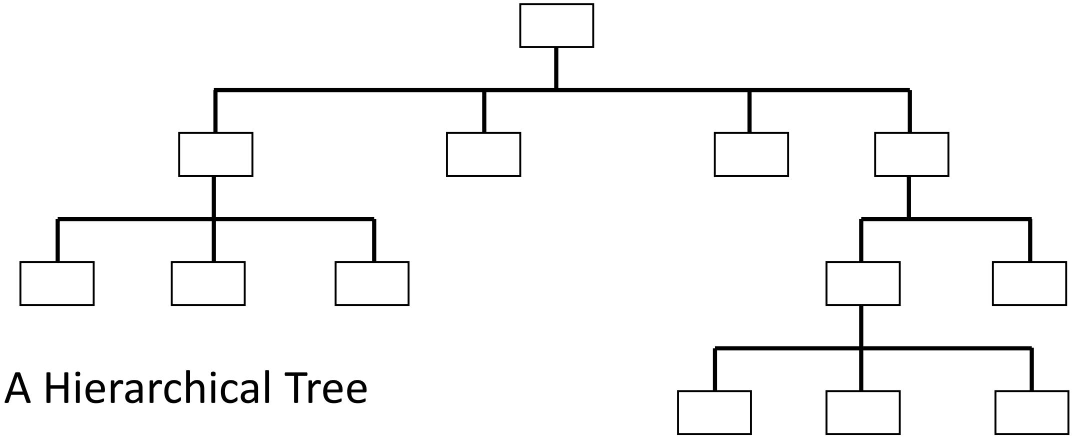
# Binary tree - application

- A binary tree is a useful data structure when two-way decisions must be made at each point in a process
  - ✓ Example: Finding duplicates in a list of numbers
- A binary tree can be used for representing an expression containing operands (leaf) and operators (nonleaf node). Traversal of the tree will result in infix, prefix or postfix forms of expression

Two binary trees are MIRROR SIMILAR if they are both empty or if they are nonempty, the left subtree of each is mirror similar to the right subtree

## General tree

- A tree is a finite nonempty set of elements in which one element is called the ROOT and remaining element partitioned into m >=0 disjoint subsets, each of which is itself a tree
- Different types of trees – binary tree, n-ary tree, red-black tree,  AVL tree



A Hierarchical Tree

# Heap

Heaps are based on the notion of a **complete tree**

A binary tree is **completely full** if it is of height, *h*, and has 2*h*+1-1 nodes.

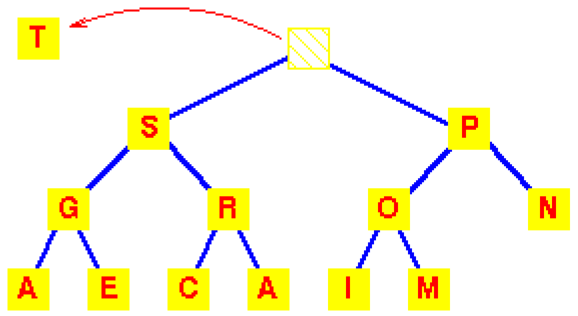
- A binary tree of height, *h*, is **complete** *if*
  - ✓ it is empty *or*
  - ✓ its left subtree is complete of height *h*-1 and its right subtree is completely full of height *h*-2 *or*
  - ✓ its left subtree is completely full of height *h*-1 and its right subtree is complete of height *h*-1.
- A complete tree is filled from the left:
  - ✓ all the leaves are on
  - ✓ the same level *or* two adjacent ones *and*
  - ✓ all nodes at the lowest level are as far to the left as possible.
- A binary tree has the **heap property** *if*
  - ✓ it is empty *or*
  - ✓ the key in the root is larger than that in either child and both subtrees have the heap property.

# Heap

- A heap can be used as a priority queue:
- the highest priority item is at the root and is trivially extracted. But if the root is deleted, we are left with two sub-trees and we must *efficiently* re-create a single tree with the heap property.
- The value of the heap structure is that we can both extract the highest priority item and insert a new one in **O(log*n*)** time.

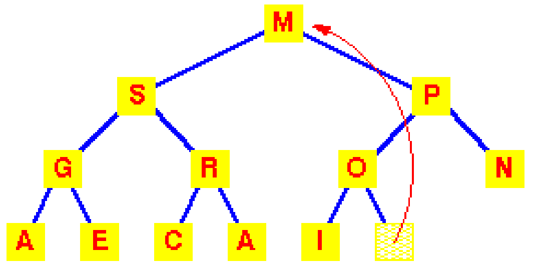
**Example:**

**A deletion will remove the T at the root**

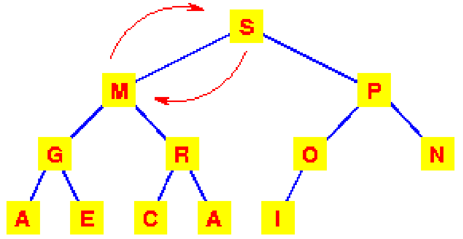


# Heap

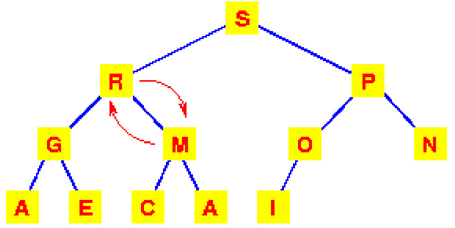
To work out how we're going to maintain the heap property, use the fact that a complete tree is filled from the left. So that the position which must become empty is the one occupied by the M. Put it in the vacant root position.



This has violated the condition that the root must be greater than each of its children. So interchange the M with the larger of its children. The left subtree has now lost the heap property. So again interchange the M with the larger of its children.



We need to make at most *h* interchanges of a root of a subtree with one of its children to fully restore the heap property.

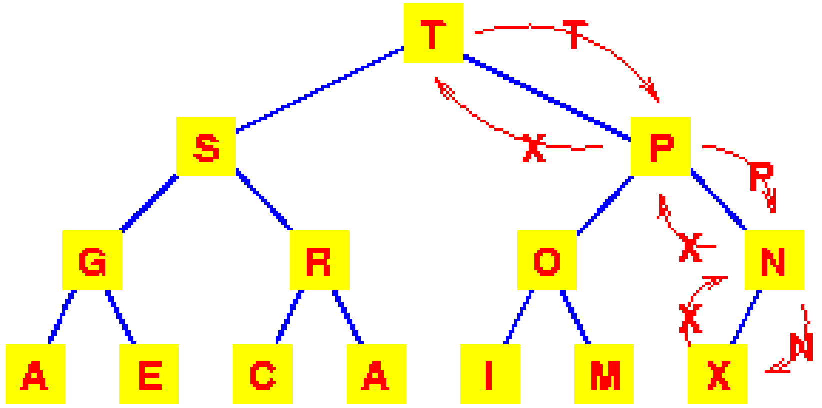


**O(h) or O(log n)**

# Heap

## Adding to a Heap

- To add an item to a heap, we follow the reverse procedure.
- Place it in the next leaf position and move it up.
- Again, we require O(*h*) or O(log*n*) exchanges.



# Data Structure Comparisons

| | Arrays | Linked List | Trees |
|---|---|---|---|
| | Simple, fast<br>Inflexible | Simple<br>Flexible | Still Simple<br>Flexible |
| Add | O(1)<br>O(n) *inc sort* | O(1)<br>*sort -> no adv* | O(log n) |
| Delete | O(n) | O(1) - *any*<br>O(n) - *specific* | O(log n) |
| Find | O(n)<br>O(logn)<br>*binary search* | O(n)<br>*(no bin search)* | O(log n) |

# Queue

Queues are dynamic collections which have some concept of order

- **FIFO queue**
  - ✓ A queue in which the first item added is always the first one out.
- **LIFO queue**
  - ✓ A queue in which the item most recently added is always the first one out.
- **Priority queue**
  - ✓ A queue in which the items are sorted so that the highest priority item is always the next one to be extracted.

**Queues can be implemented by Linked Lists**

# Stacks

- **Stacks are a special form of collection with LIFO semantics**
- Two methods
  - ✓ `int push( Stack s, void *item );`

    - **add item to the top of the stack**
  - ✓ `void *pop( Stack s );`

    - **remove most recently pushed item from the top of the stack**
- Like a plate stacker
- Other methods

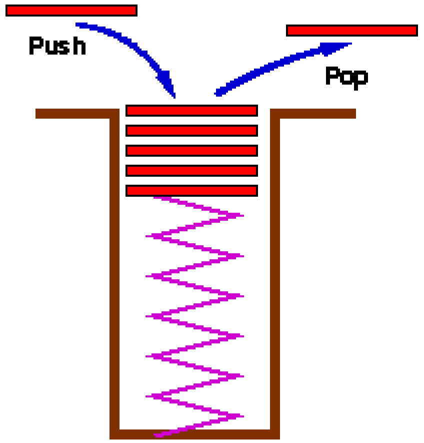
  `int IsEmpty( Stack s );`

  **Determines whether the stack has anything in it**

  `void *Top( Stack s );`

  **Return the item at the top without deleting it**

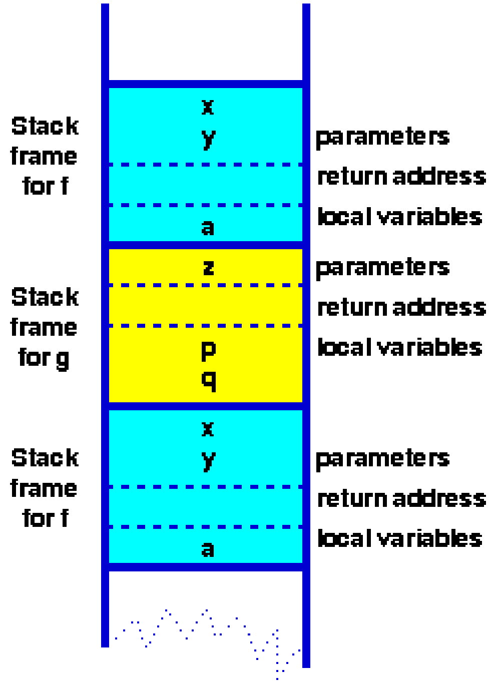
  *** Stacks are implemented by Arrays or Linked List**



# Stack

- Stack very useful for Recursions
- Key to call / return in functions & procedures

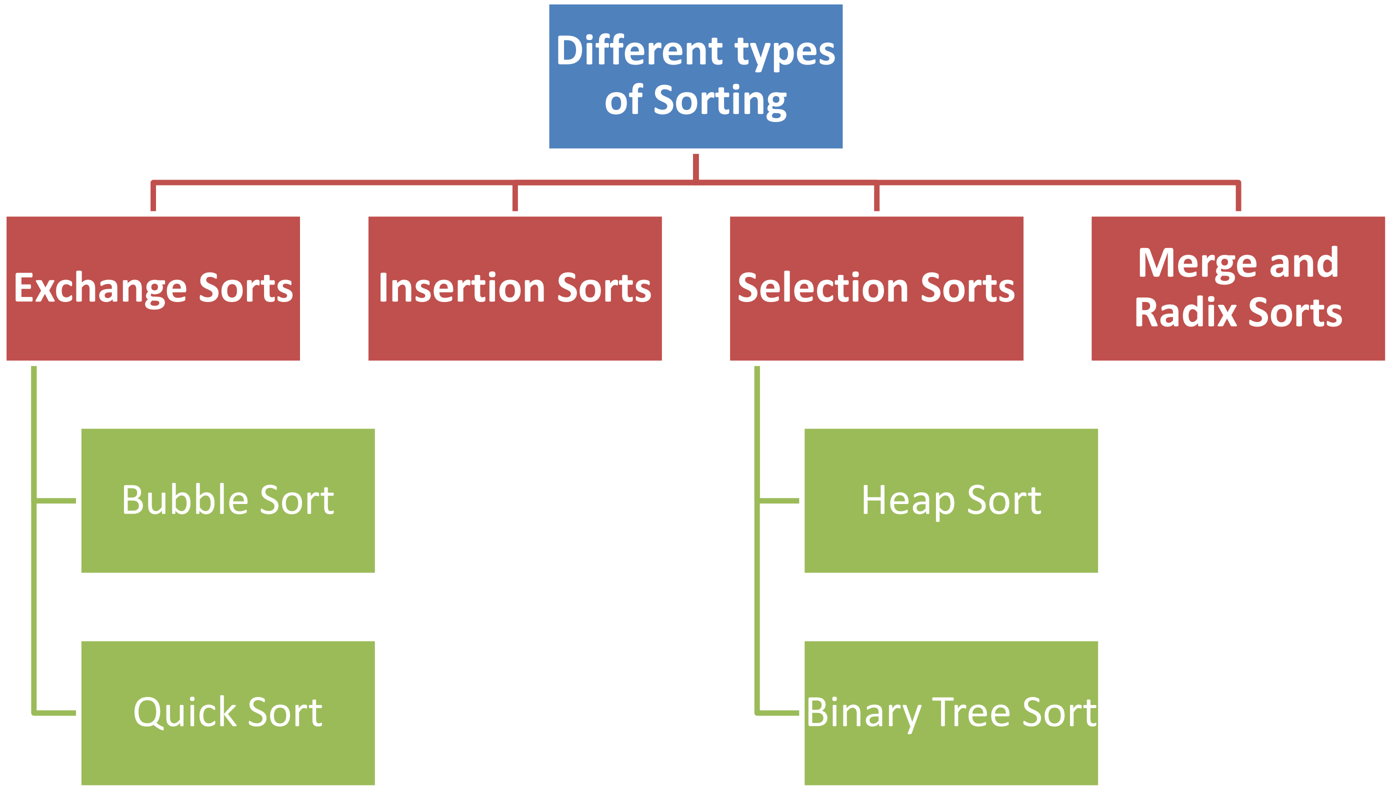
```
function f( int x, int y) {
   int a;
   if ( term_cond ) return …;
   a = ….;
   return g( a );
   }

function g( int z ) {
   int p, q;
   p = …. ; q = …. ;
   return f(p,q);
   }
```



Section 3

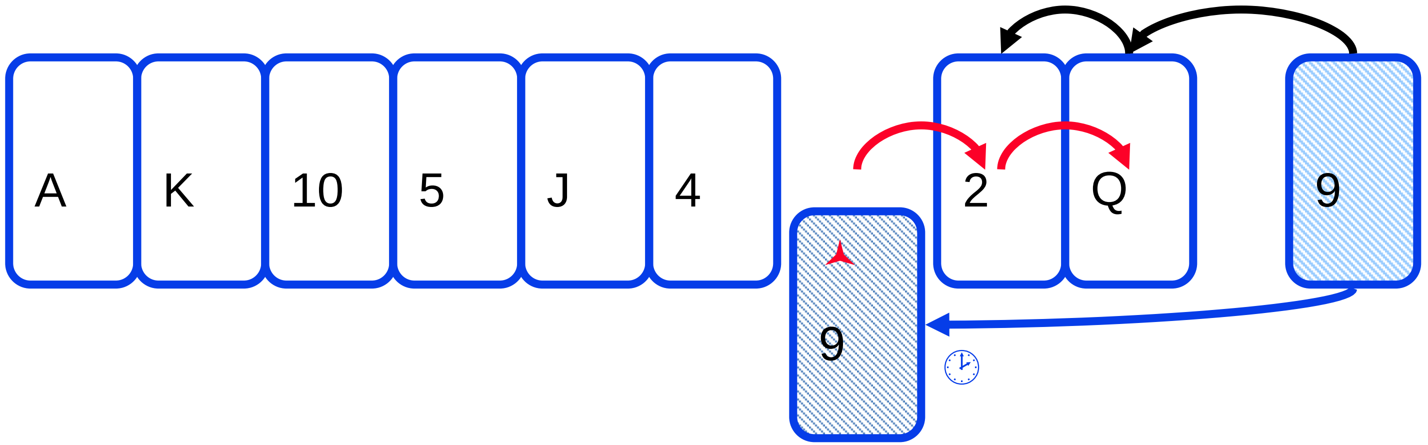
## Sorting

# Sorting algorithms

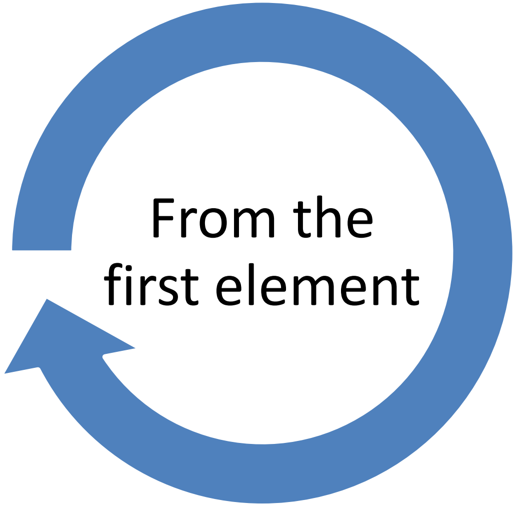


# Insertion sort

- Scan back from the end until you find the first card larger than the new one
- Move all the lower ones up one slot O(n)
- insert it



# Bubble sort



- Exchange pairs if they're out of order
- Repeat from the first to n-1
- Stop when you have only one element to check

```
/* Bubble sort for integers */
#define SWAP(a,b)   { int t; t=a; a=b; b=t; }

void bubble( int a[], int n ) {
  int i, j;
            for(i=0;i<n;i++) { /* n passes thru the array */
                      /* From start to the end of unsorted part */
                      for(j=1;j<(n-i);j++) {
                      /* If adjacent items out of order, swap */
                       if( a[j-1]>a[j] ) SWAP(a[j-1],a[j]);
                      }
            }
  }
                         Overall O(n2)
```

# Selection sort

Algorithm:

- Pass through elements sequentially;
- In the *i*th pass, we select the element with the lowest value in A[i] through A[n], then swap the lowest value with A[i].

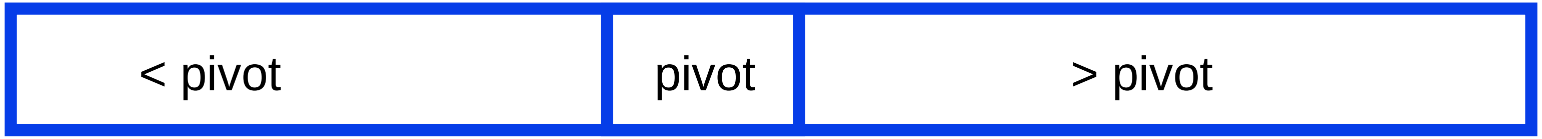
Time complexity: $O(n^2)$

# Quick sort

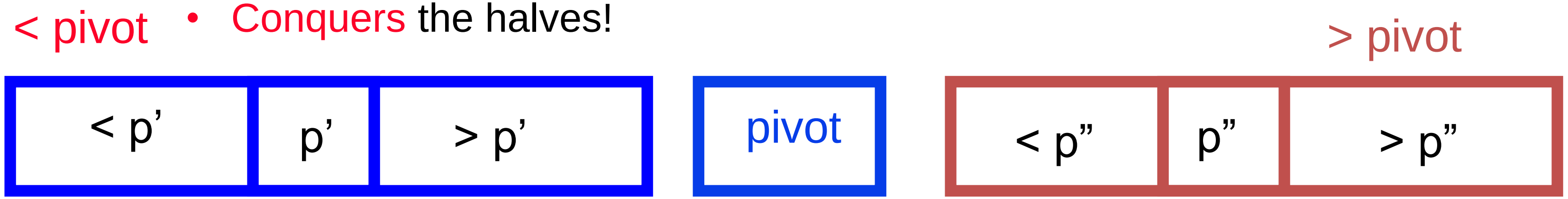
- Quick sort, also known as partition sort, sorts by employing a divide-and-conquer strategy.
- Algorithm:
  - ✓ Pick an pivot element from the input;
  - ✓ Partition all other input elements such that elements less than the pivot come before the pivot and those greater than the pivot come after it (equal values can go either way);
  - ✓ Recursively sort the list of elements before the pivot and the list of elements after the pivot.
  - ✓ The recursion terminates when a list contains zero or one element.
- Time complexity: O($n\log n$) or O($n^2$)
- Demo: http://pages.stern.nyu.edu/~panos/java/Quicksort/
- Example: Sort the list {25, 57, 48, 37, 12}

# Quick sort

- Example of Divide and Conquer algorithm
- Two phases
  - ✓ Partition phase
    - Divides the work into half



  - ✓ Sort phase
    - Conquers the halves!



# Heap

## Heaps also provide a means of sorting:

- construct a heap,
- add each item to it (maintaining the heap property!),
- when all items have been added, remove them one by one (restoring the heap property as each one is removed).
- Addition and deletion are both **O(log*n*)** operations. We need to perform ***n*** additions and deletions, leading to an **O(*n*log*n*)** algorithm
- Generally slower

# Sorting comparision

| | | |
|---|---|---|
| ✓ Insertion | $O(n^2)$ | ***Guaranteed*** |
| ✓ Bubble | $O(n^2)$ | ***Guaranteed*** |
| ✓ Heap | $O(n \log n)$ | ***Guaranteed*** |
| ✓ Quick | $O(n^2)$ - $O(n \log n))$ | ***Most of the time!*** |
| ✓ Bin | $O(n)$ - $O(n\log n)$ | ***Keys in small range*** |
| ✓ Radix | $O(n)$ - $O(n+m)$ | ***Bounded keys/duplicates*** |

Section 4

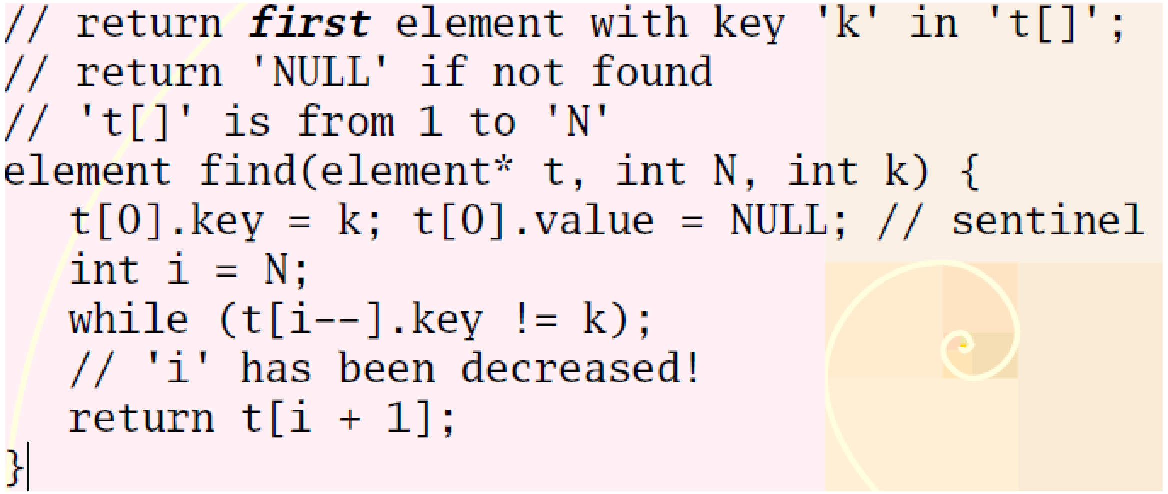
## Searching

## Searching

- Fundamental operation
- Finding an element in a (huge) set of other elements
  - ✓ Each element in the set has a key
- Searching is the looking for an element with a given key
  - ✓ distinct elements may have (share) the same key
  - ✓ how to handle this situation?
    - first, last, any, listed, ...
- May use a specialized data structure
- *Things to consider*
  - ✓ the average time
  - ✓ the worst-case time and
  - ✓ the best possible time.

## Sequential Search

- Store elements in an array
  - ✓ Unordered



```
// return first element with key 'k' in 't[]';
// return 'NULL' if not found
// 't[]' is from 1 to 'N'
element find(element* t, int N, int k) {
   t[0].key = k; t[0].value = NULL; // sentinel
   int i = N;
   while (t[i--].key != k);
   // 'i' has been decreased!
   return t[i + 1];
}
```

## Sequential Search Analysis

- Generic simple algorithm
- Space complexity: O(1)
- Time complexity
  - ✓ Time is proportional to *n*
  - ✓ We call this time complexity *O(n)*
    - Worst case: N + 1 comparisons
    - Best case: 1 comparison
    - Average case (successfull): (1+2+...+N)/N = (N+1)/2
- Both arrays (unsorted) and linked lists

# Sequential Search in a (sorted) Linked List 1/2

- Keep the list sorted
  - ✓ Easy to implement with linked list (***exercice: do it)!***

```
// return first node with key 'k' in 'l';
// return 'NULL' if not found
// 'l' is sorted
node find(list l, int k) {
   node z = list_end(l);
   node_setKey(z, k); // sentinel
   for (node n = list_start(l);
     node_getKey(n) > k;
     n = node_next(n));
   if (node_getKey(n) != k) return NULL;
   return n;
}
```

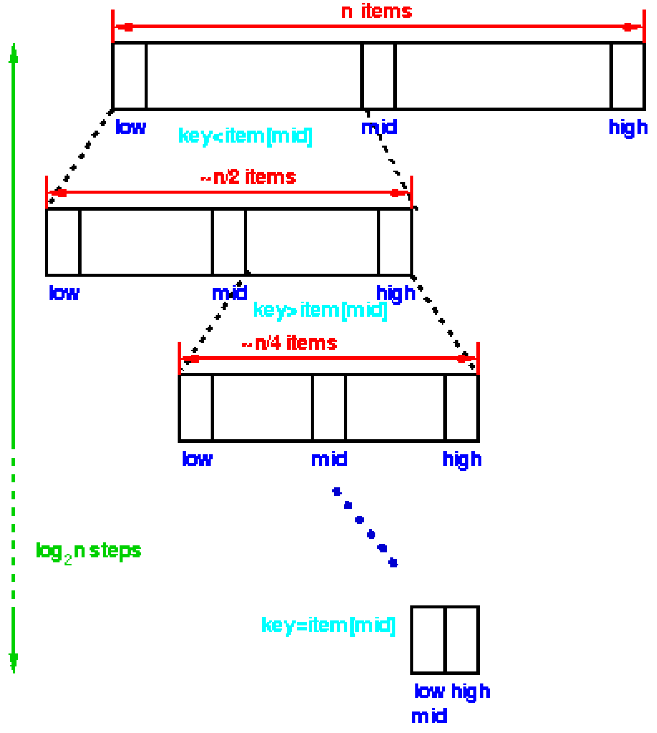
## Sequential Search Improvements

- Static caching
  - ✓ Use the *relative access frequency of elements*
    - store the *most often accessed elements at the first places*
- Dynamic caching
  - ✓ For each access, move the element to the first position
    - *Needs a linked list data structure to be efficient*
- Very difficult to analyze the complexity in theory: very efficient in practice

## Binary Search

- Sorted array on a key
- first compare the key with the item in the middle position of the array
- If there's a match, we can return immediately.
- If the key is less than the middle key, then the item sought must lie in the lower half of the array
- if it's greater then the item sought must lie in the upper half of the array
- Repeat the procedure on the lower (or upper) half of the array - **RECURSIVE**

Time complexity *O*(log *n*)



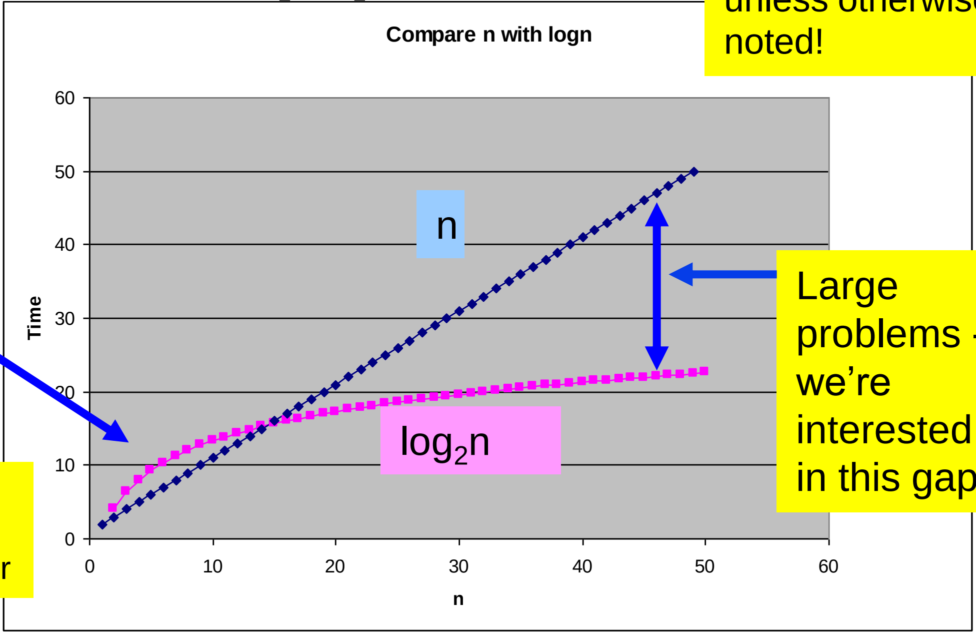
# Binary Search Implementation

```
static void *bin_search( collection c, int low, int high, void *key ) {
        int mid;
        if (low > high) return NULL; /* Termination check */
        mid = (high+low)/2;
        switch (memcmp(ItemKey(c->items[mid]),key,c->size)) {
            case 0: return c->items[mid]; /* Match, return item found */
            case -1: return bin_search( c, low, mid-1, key); /* search lower half */
            case 1: return bin_search( c, mid+1, high, key ); /* search upper half */
            default : return NULL;
            }
        }

void *FindInCollection( collection c, void *key ) {
/* Find an item in a collection
        Pre-condition:
            c is a collection created by ConsCollection
            c is sorted in ascending order of the key
            key != NULL
        Post-condition: returns an item identified by key if one exists, otherwise returns NULL */

             int low, high;
             low = 0; high = c->item_cnt-1;
            return bin_search( c, low, high, key );
         }
```

# Binary Search vs Sequential Search

- **Find method**
  - ✓ **Sequential search**
    - Worst case time: $c_1 n$
  - ✓ **Binary search**
    - Worst case time: $c_2 \log_2 n$

Logs
Base 2 is by far the most common in this course. Assume base 2 unless otherwise noted!



Small problems - we're not interested!

Large problems - we're interested in this gap!

Binary search
More complex
Higher constant factor

Fpt Software | CAMPUS

# Thank you