C programming

Data structure & algorithms

Section 1

Data structures

What is data structure

Example of data usage in programs:

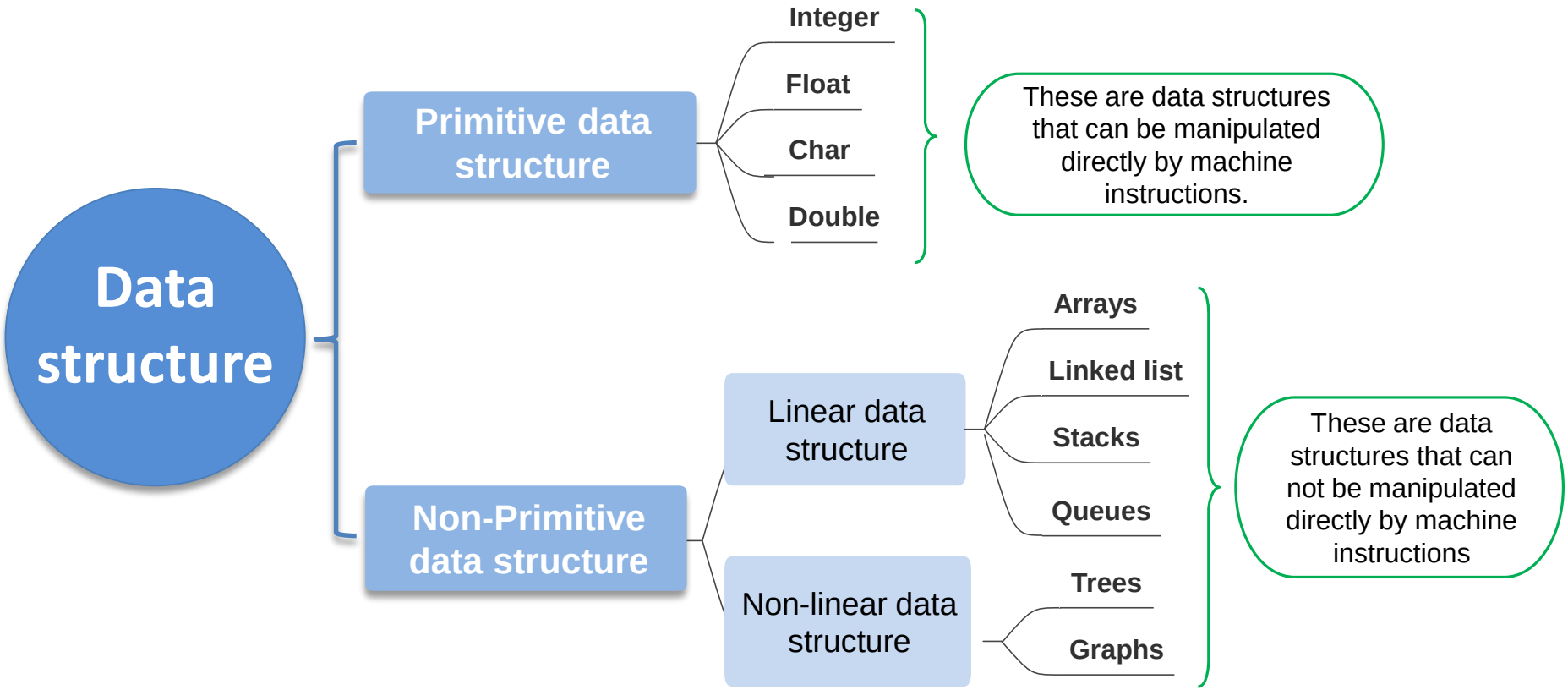
Program A: Count the quantity of a product.
The data we need is just an integer.

Program B: Count the quantities of N products.
We can use an array. Each element is an integer that stores quantity of a product.

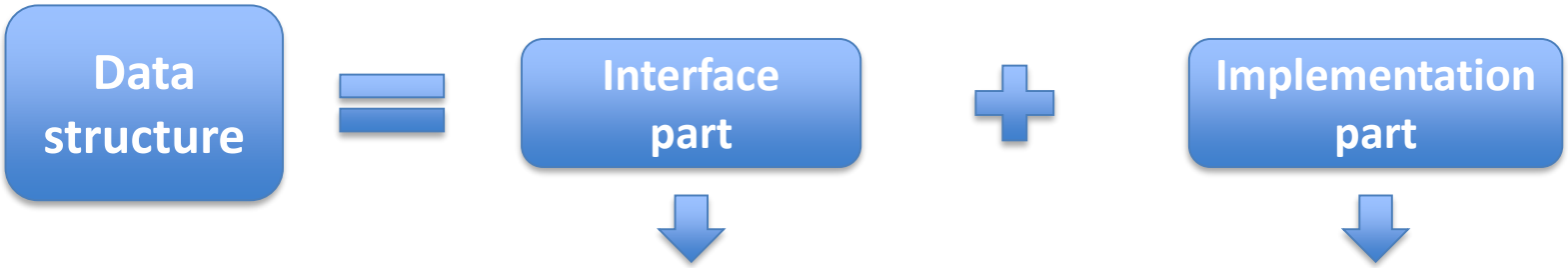
Program C: Manage information of N products like ID, quantity, price…
We should use an array. Each element is a structure which stores information of a product.

- A program can have many ways to use data. Each way requires a certain data format which can be called a data structure.
- **Data structure** is a specialized format for organizing and storing data so that it can be accessed and worked with in appropriate ways to make a program efficient.

Data structure classification



Data structure design aspect



Example: Collection

Programs often deal with collections of objects.

Ex: A program that manages a class that will use collection of students, subjects.

There will be a few common operations (interface part) on any collection like: Create, add, find, delete, destroy.

These collections may use organized data in many ways and use many different program structures to implement. We can use arrays, links lists or trees to store data. We can use linear search or binary search for Find operation.

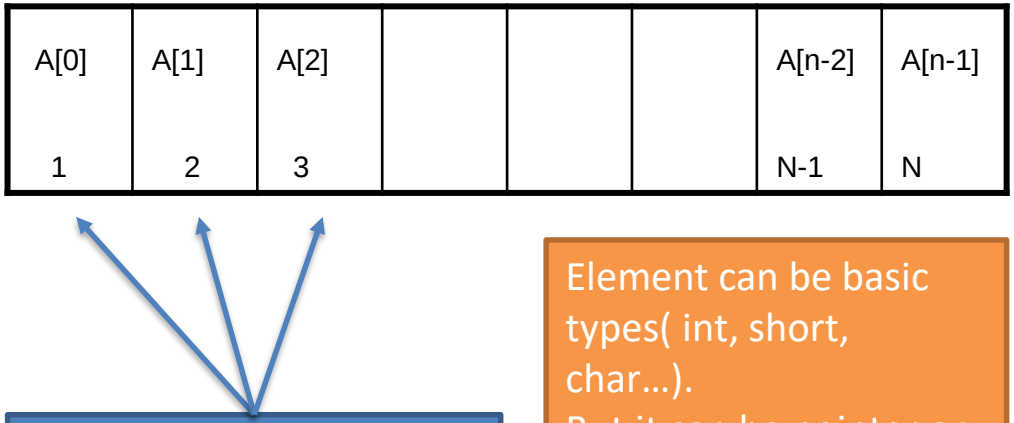
Section 2

Data structure example

Array

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



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

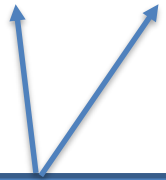
Element can be basic types(int, short, char…).
But it can be pointer or struct.

One dimensional array example:

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

Array

A[0]	A[1]	A[2]				A[n-2]	A[n-1]
1	2	3				N-1	N



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

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

What happens if the pointer access over the size of the array? N = 20?

Multi-dimension array

Syntax:

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

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

OR

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

Example for storing a 3x3 matrix

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

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

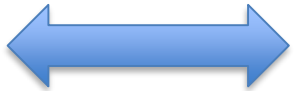
What will happen if you put too few or too more elements in an array when you initialized it?

Accessing multi-dimension array

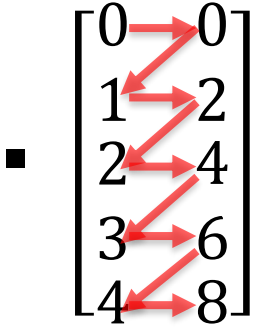
Example for access 5x2 matrix

```
/* 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] );
   }
}
```

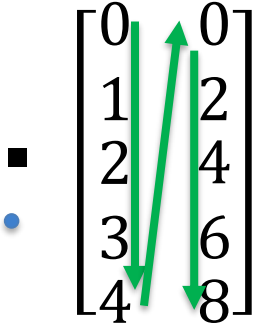


Row major order



How does the code change to access follow column major order?

Column major order



Array: Limitation

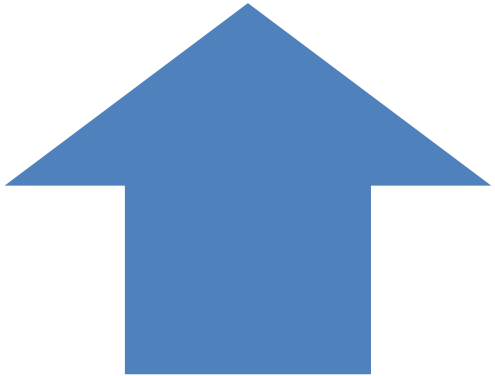
Simple and Fast but must specify size during construction

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

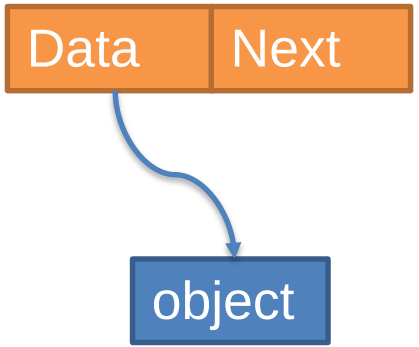


Copying elements can result in longer running times for a program if insert/ remove operations are frequent, especially when you consider the cost of copying is huge (like when we copy strings)

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

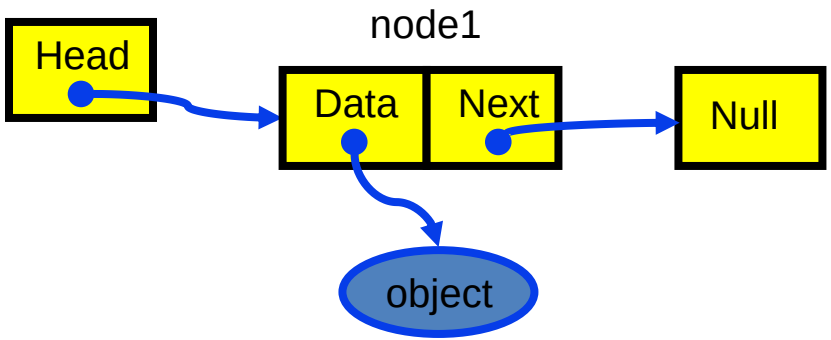
- **The linked list is a very flexible dynamic data structure: Items may be added to it or deleted from it at will.**
  - ✓ Dynamically allocate space for each element as needed
  - ✓ The number of items that may be added to a list is limited only by the amount of memory available
- Linked list can be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



Linked List

- 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 List

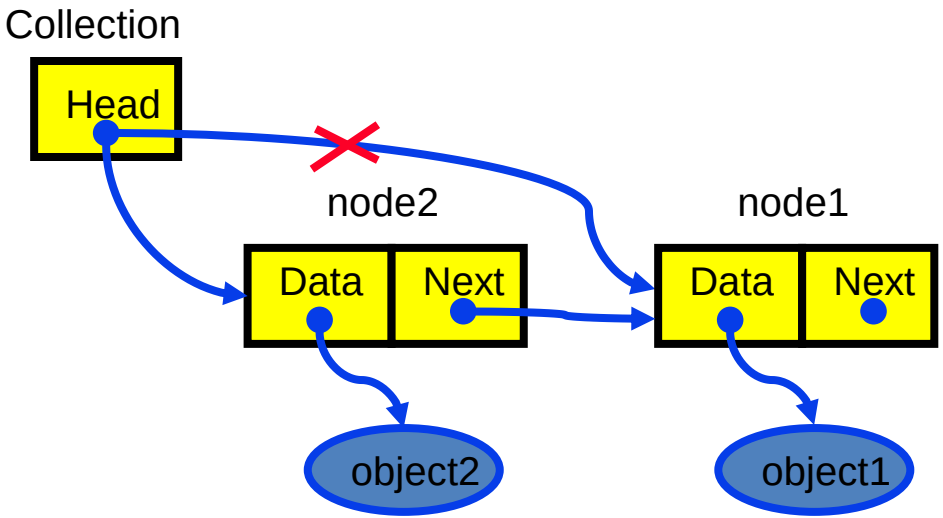
- **Add a node**

✓ Allocate space for node

✓ Set its data pointer to object

✓ Set Next to current Head

✓ Set Head to point to new node



Linked List – Example

- Add implementation

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

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

Linked List – Example

- Find Implementation

✓ Use a node pointer to check each element of the list in turn.

✓ If one node whose key is equal to the key being searched, the node will be return.

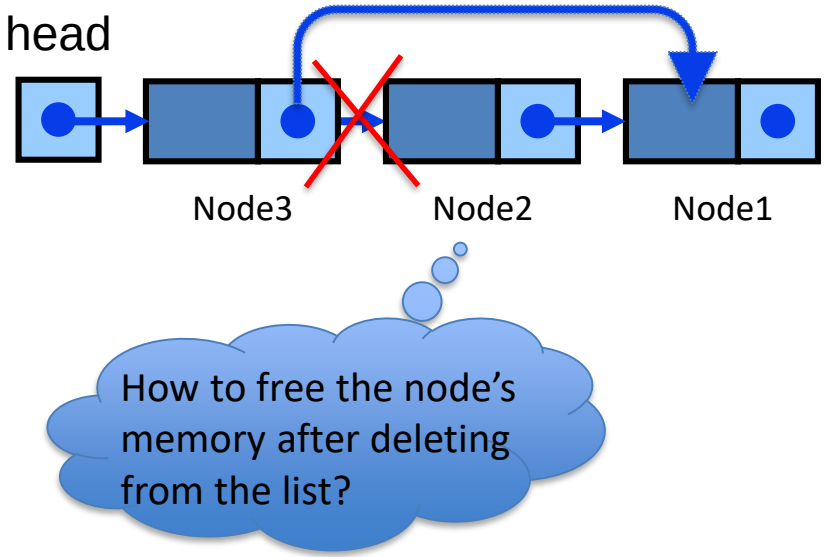
✓ If no node is found, the program will return null.

```
void *FindinC (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 – Example

- Delete Implementation

Simply remove connection between two nodes and reconnect it with other

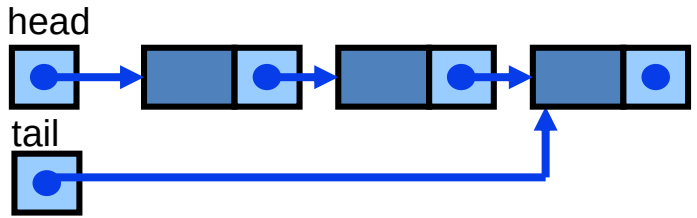


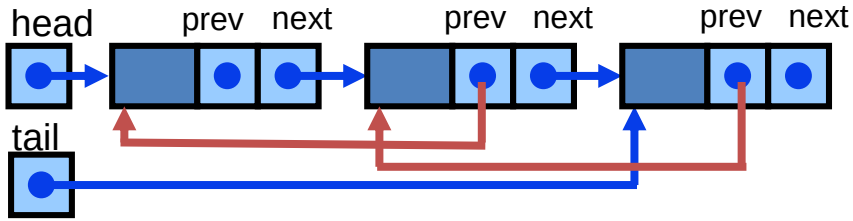

```
void *DeleteFromC(Collection c,void *key){
  Node n, prev = c->head;

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

Linked List – Variations

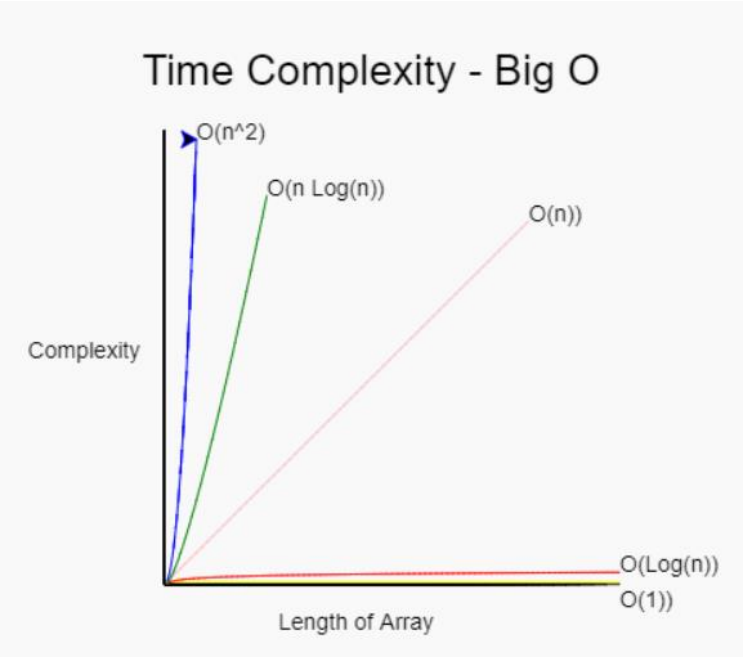
- **Simplest implementation**
  - ✓ Add to head
  - ✓ Last-In-First-Out (LIFO) semantics
- **Modifications**
  - ✓ First-In-First-Out (FIFO)
  - ✓ Keep a tail pointer
- **Circularly linked list**
  - ✓ By ensuring that the tail's next of the list is always pointing to the head.
  - ✓ Head is tail->next
  - ✓ LIFO or FIFO using **ONE** pointer
- **Doubly linked list**
  - ✓ Using one more prev pointer to point to previous node.
  - ✓ Can be scanned in both directions
  - ✓ For application which require both way search. Eg: Name search in directory.





Data Structure Comparisons

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



Queue

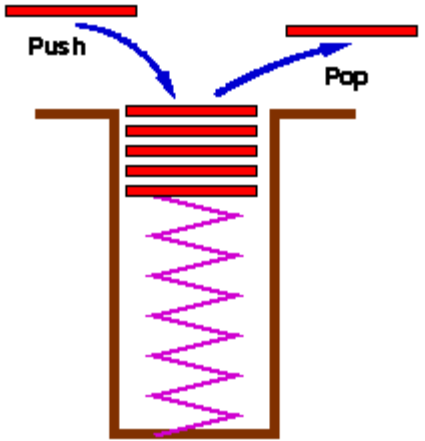
Queues are dynamic collections which have some concept of order

- **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**

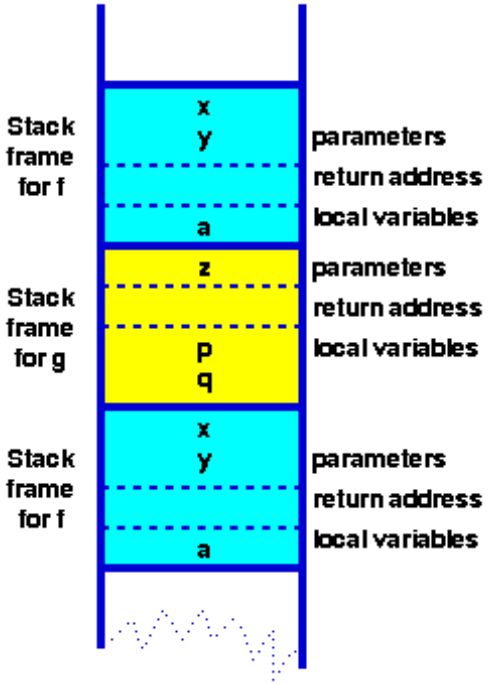


*** Stacks are implemented by Arrays or Linked List**

Stacks

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

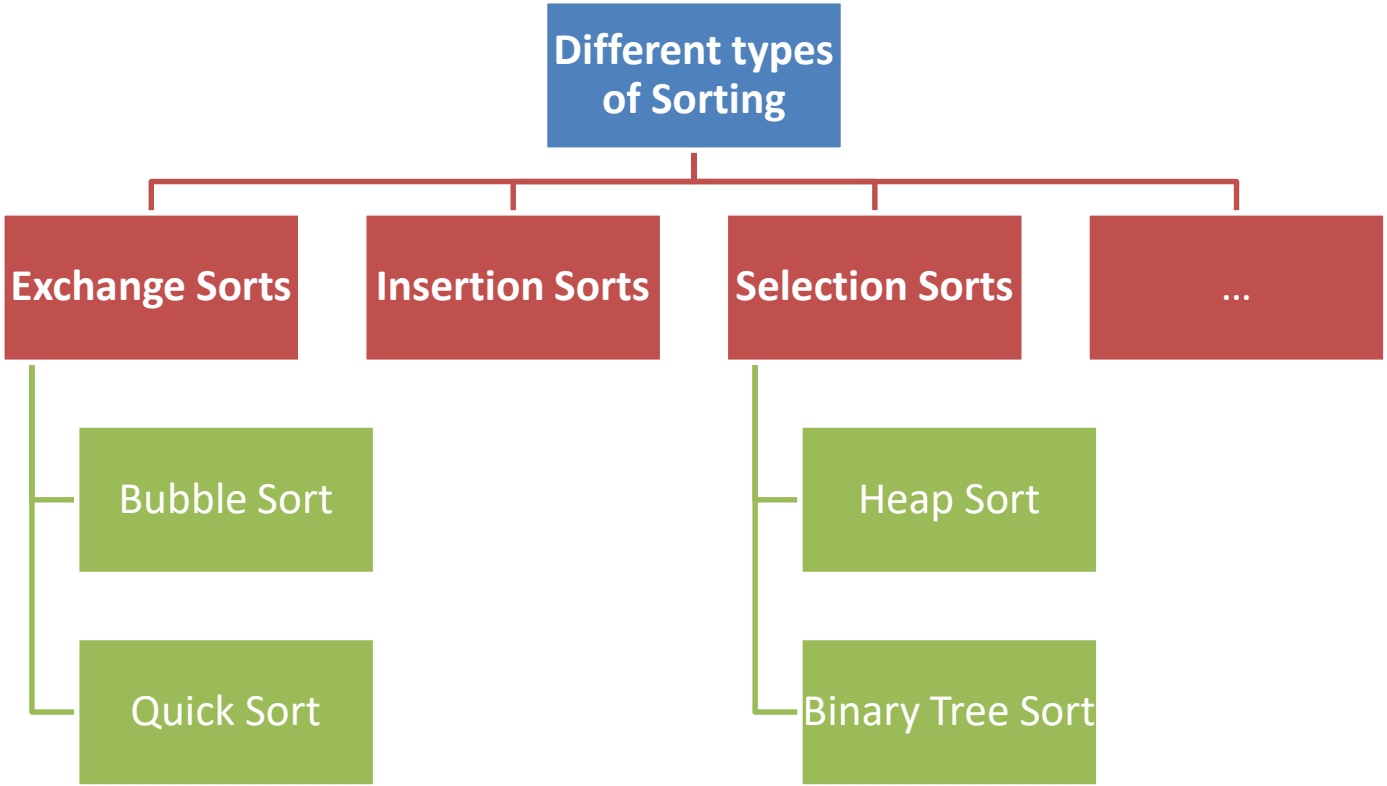
```
function f( int x, int y)
{
    int a;
    if ( term_condition ) return …;
    a = ….;
    return g( a );
}
function g( int z )
{
    int p, q;
    p = …. ; q = …. ;
    return f(p,q);
}
```



Section 3

Sorting

Sorting algorithms



Insertion sort

GIF Demo:


```
/* Insertion sort for integers */
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Overall $O(n2)$

- **Algorithm:**

✓ Iterate from arr[1] to arr[n] over the array.

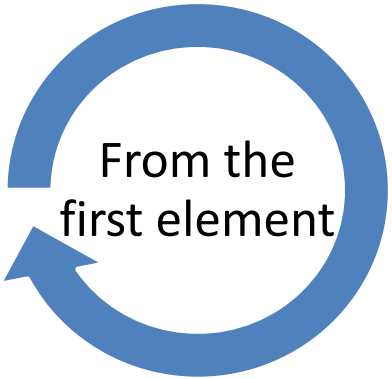
✓ Compare the current element (key) to its predecessor.

✓ If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Bubble sort

GIF Demo:

6 5 3 1 8 7 2 4



- 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

GIF Demo:

8	5	2	6	9	3	1	4	0	7

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: $O(n^2)$

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

void selectionSort(int arr[], int n)
{
   int i, j, min_idx;
   for (i = 0; i < n-1; i++)
   {
      /* Find the min element in unsorted array */
      min_idx = i;
      for (j = i+1; j < n; j++)
      {
         if (arr[j] < arr[min_idx])
         {
            min_idx = j;
         }
      }
      /* Swap the found min with the first */
      SWAP (arr[min_idx], arr[i]);
   }
}
```

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)
- Quick sort tutorial

Quick sort

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

6 5 3 1 8 7 2 4

  - ✓ Sort phase
    - Conquers the halves!

< pivot

> pivot

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***

- Sorting Demo

Section 4

Searching

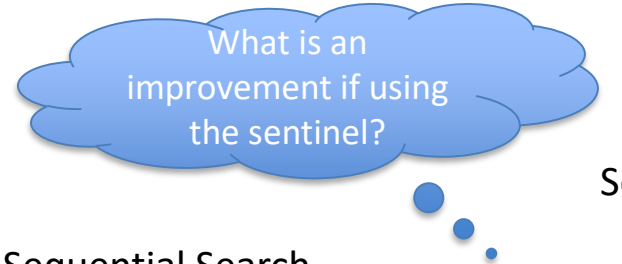
Searching

- 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, ...
- *Things to consider*
  - ✓ The average time
  - ✓ The worst-case time and
  - ✓ The best possible time.

Sequential Search

- Store elements in an Unordered array.
- Pre-condition:
  - ✓ *A[] is from 1 to N*
- Post-condition:
  - ✓ *Return data of first element with key 'k' in A[];*
  - ✓ *Return NULL if not found*

Each element of the array is compared to the key, in the order it appears in the array, until the first element matching the key is found



Sequential Search

```
data find (item A[], int N, int k)
{
   A[0].data = NULL;
   int i = N;
   while( (A[i].key != k) & (i > 0) )
   {
       i--;
   }
   return A[i].data;
}
```

Sequential Search with a sentinel

```
data find (item A[], int N, int k)
{
   A[0].key = k; /*sentinel*/
   A[0].data = NULL;
   int i = N;
   while( A[i].key !=  k )
   {
       i--;
   }
  return A[i].data;
}
```

Sequential Search Analysis

- Generic simple algorithm
- 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 (successful): (1+2+...+N)/N = (N+1)/2
- Both arrays (unsorted) and linked lists

Sequential Search in a Linked List

- Store elements in a linked list L.
- Pre-condition:
  - ✓ *The end node in L have data = NULL*
- Post-condition:
  - ✓ *Return data of first element with key 'k' in 'L';*
  - ✓ *Return NULL if not found*

Each node's key in the list is compared to the key, in the order it appears in the List, until the first node matching the key is found

```
data find(list L, int k)
{
    node z = list_end(L);
    node n = list_start(L);
    z->key = k;                /* sentinel */
    while( n->key != k )
    {
        n = n->next;
    }
    return n->data;
}
```

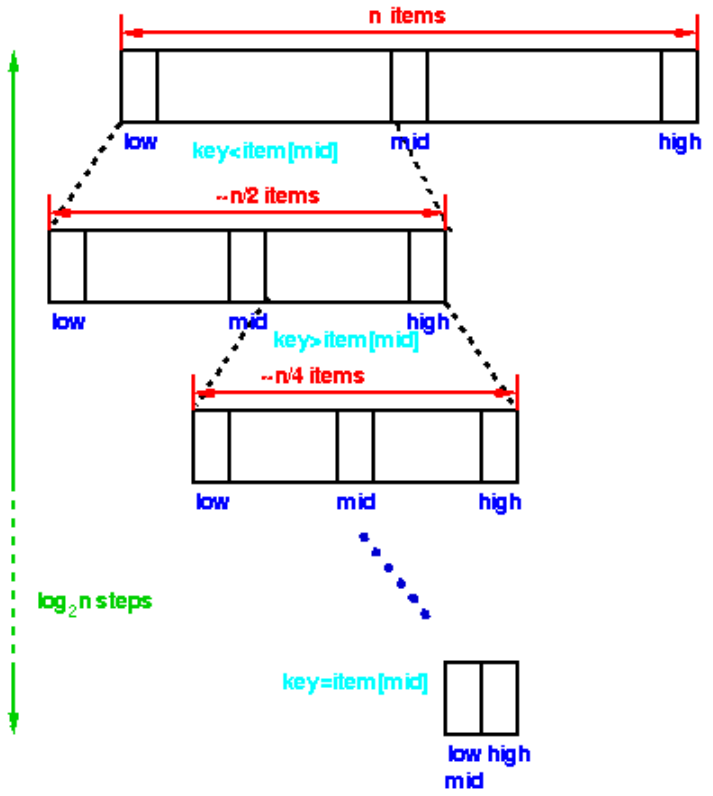
Sequential Search Improvements

- 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*)



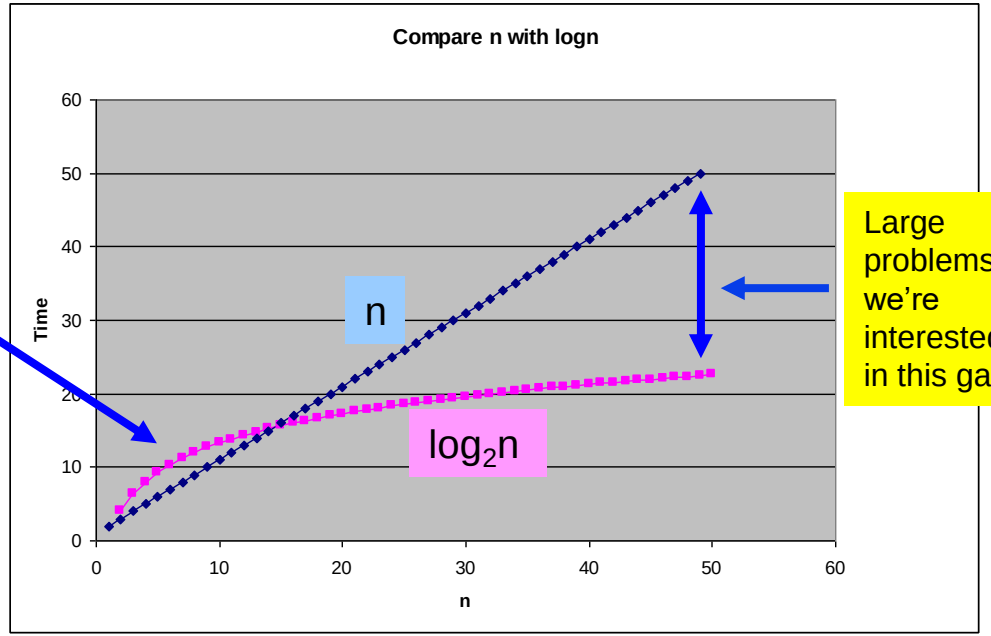
Binary Search Implementation

- Find an item in a collection C
- Pre-condition:
  - ✓ *C is sorted in ascending order of the key.*
  - ✓ *Key != NULL*
- Post-condition:
  - ✓ *Return an item identified by key if one exists,*
  - ✓ *Otherwise returns NULL*

```
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 ) {
  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: $c_2 \log_2 n$



Thank you