

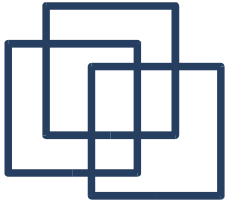
C in a Nutshell, Pointers and Basic Data Structures

Alexandre David
adavid@cs.aau.dk

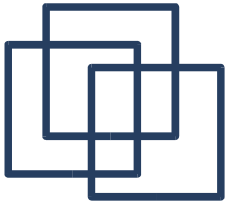
Credits:

Emmanuel Fleury





C in a Nutshell



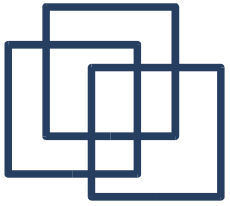
Hello World

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])  
{
```

```
    printf("Hello world!\n");
```

```
    return 0;  
}
```



Hello World

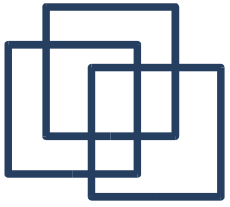
```
#include <stdio.h>
```

```
include header for the definition of printf
```

```
int main(int argc, char *argv[])  
{
```

```
    printf("Hello world!\n");
```

```
    return 0;  
}
```

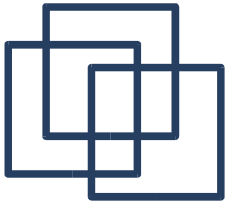


Hello World

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])  
{  
    main->int ; argc: #arguments ; argv: arguments  
    printf("Hello world!\n");
```

```
    return 0;  
}
```



Hello World

```
#include <stdio.h>
```

```
int main(int argc,  
{
```

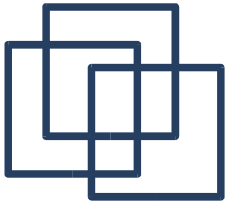
array of *pointers* to char
= array of strings

```
char *argv[ ])
```

```
    printf("Hello world!\n");
```

```
    return 0;
```

```
}
```



Hello World

```
#include <stdio.h>
```

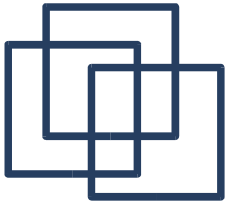
```
int main(int argc, char *argv[])  
{
```

```
    printf("Hello world!\n");
```

printf function, argument = string
= char array terminated by 0

```
    return 0;
```

```
}
```



Hello World

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])  
{
```

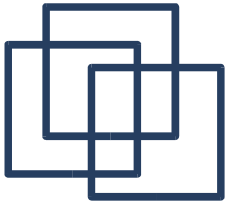
```
    printf("Hello world!\n");
```

printf function, argument = string
= char array terminated by 0

```
    return 0;
```

```
}
```

string length given by
int strlen(char *)
declared in **string.h**



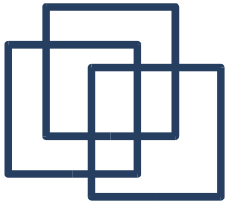
Hello World

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])  
{
```

```
    printf("Hello world!\n");
```

```
    return 0;  
}  
end of program, return to caller program
```



Syntax Reminder

- **if statements:**

```
if (cond) statement else statement
```

```
if (cond) { statements } else { statements }
```

Advice: use { ... }

- **for loops:**

```
for (statements; cond; progress) { statements }
```

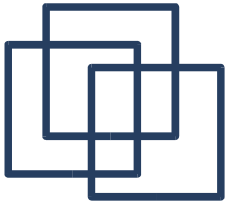
Advice: use { ... }

- **while loops/do - while loops:**

```
while (cond) { statements }
```

```
do { statements } while (cond);
```

Advice: use { ... }



Types

- **Scalars:**

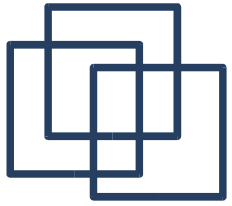
`int, char, short, long,
float, double,
unsigned int, unsigned char, ..`

- **Records:**

`struct name { types };`

- **Type definitions:**

`typedef some_type my_type;`



Types: Examples

Custom type for unsigned integers:

```
typedef unsigned int uint;
```

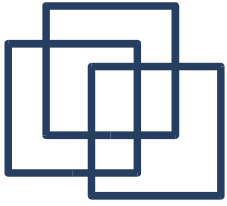
Custom type for a pair of unsigned integers:

```
typedef struct { uint first, second; } pair_t;
```

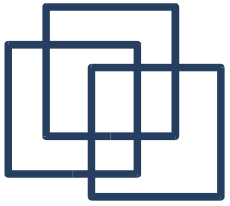
Using your pair:

```
pair_t p;  
p.first = 0;  
p.second = 1;
```

Similar to Java: a struct is like a class with public fields and without method.

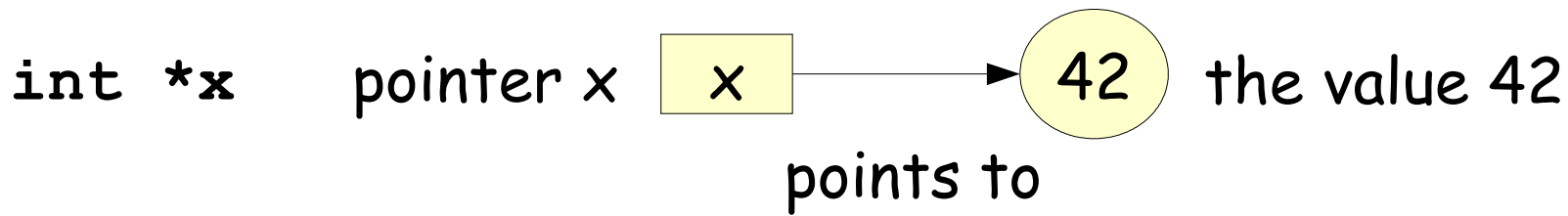


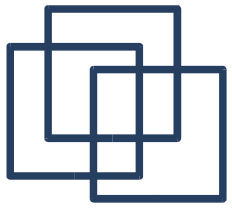
Pointers in a Nutshell



Pointers in a Nutshell

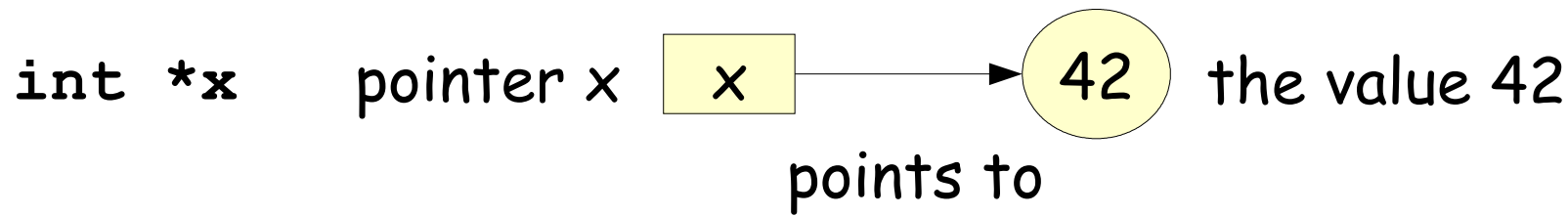
- Pointers and "Pointees": a pointers stores a reference to something.



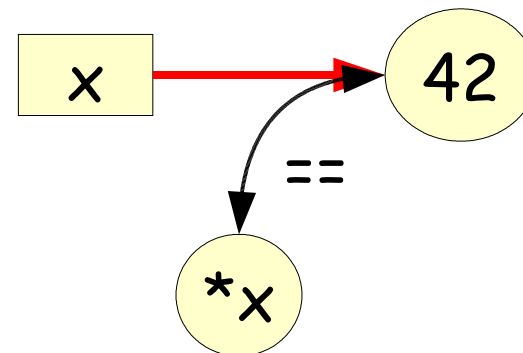


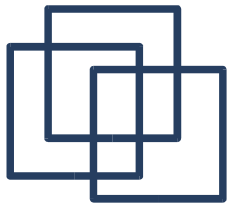
Pointers in a Nutshell

- Pointers and "Pointees": a pointer stores a reference to something.



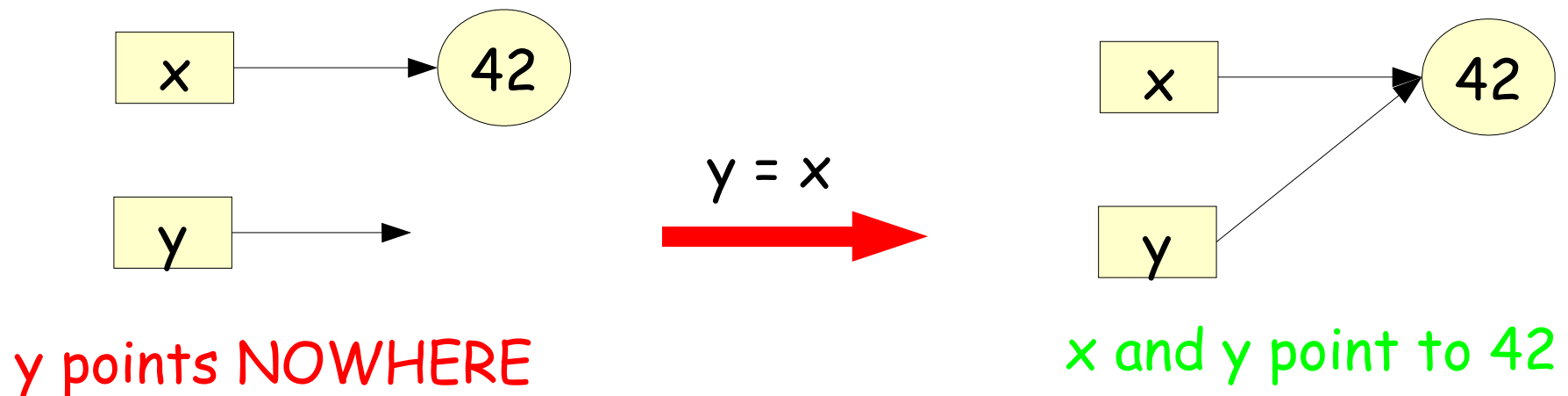
- Dereference operation: starts from the pointer and follows its arrow to access its content ("pointee").

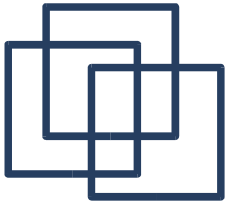




Pointers in a Nutshell

- Pointer assignment between 2 pointers makes them point to the same thing.
- The “pointee” becomes *shared* between the 2 pointers.





Binky's Code Example

```
int *x, *y;
```

x

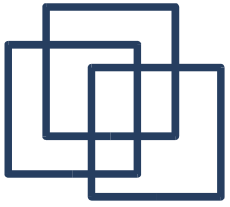


y

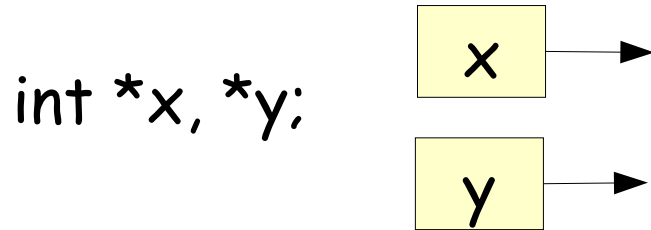


NOWHERE

Allocate 2 pointers. This does not allocate the pointee.

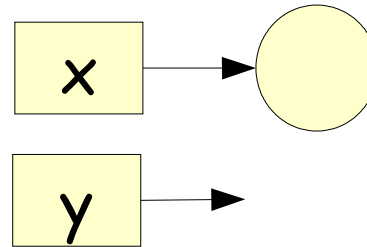


Binky's Code Example

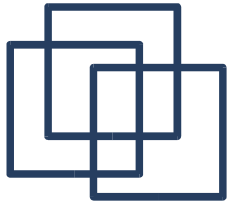


Allocate 2 pointers. This does not allocate the pointee.

`x = (int*) malloc(sizeof(int))`

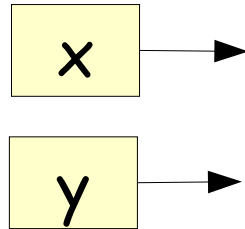


Allocate a pointee and set x to point to it.



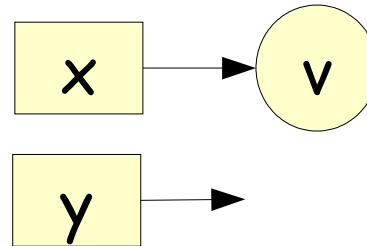
Binky's Code Example

```
int *x, *y;
```

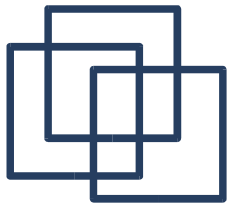


Allocate 2 pointers. This does not allocate the pointee.

```
int v;  
x = &v;
```

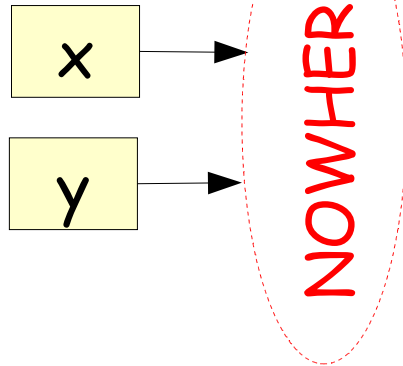


Allocate a pointee and set x to point to it.



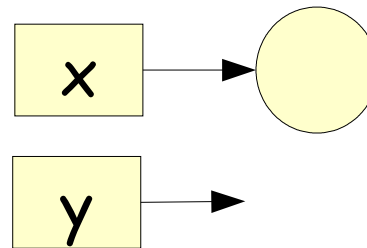
Binky's Code Example

```
int *x, *y;
```



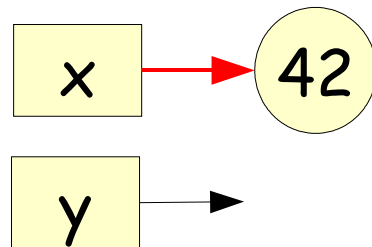
Allocate 2 pointers. This does not allocate the pointee.

```
x = (int*) malloc(sizeof(int))
```

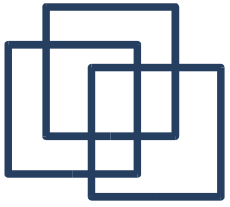


Allocate a pointee and set x to point to it.

```
*x = 42
```

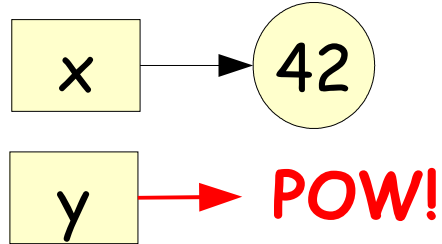


Dereference x to store 42 in its pointee.

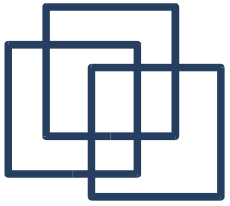


Binky's Code Example

$*y = 13$

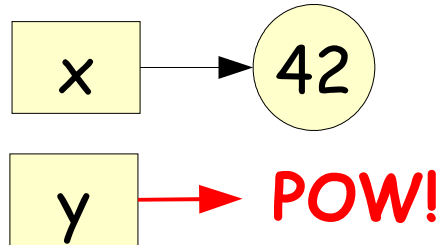


Try to dereference `y`
by storing 13 in its pointee:
there is no pointee!



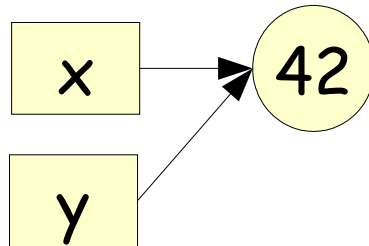
Binky's Code Example

$*y = 13$

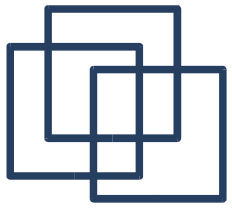


Try to dereference y
by storing 13 in its pointee:
there is no pointee!

$y = x$

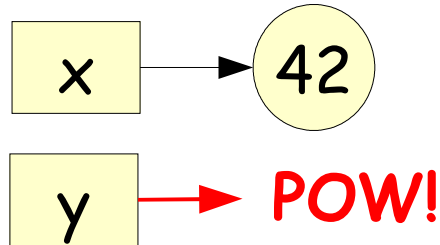


Assign y to x. Now the
pointers share the same
pointee.



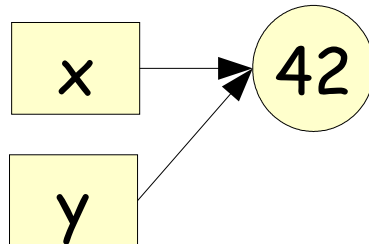
Binky's Code Example

$*y = 13$



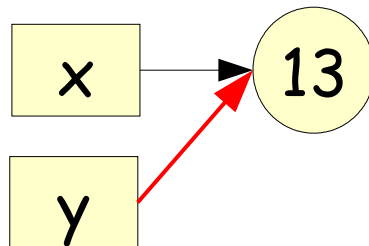
Try to dereference y by storing 13 in its pointee: there is no pointee!

$y = x$

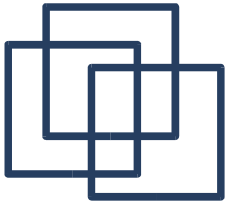


Assign y to x. Now the pointers share the same pointee.

$*y = 13$

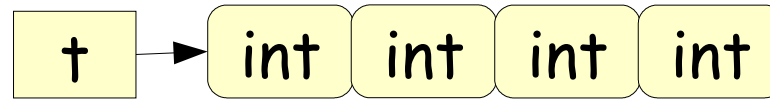


Dereference y and store 13 in its pointee. Note: $*x$ would return 13 now.

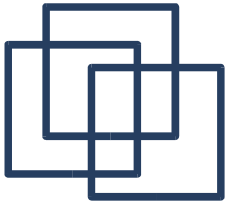


More Pointer Fun!

```
int table[4];  
int *t = table;
```

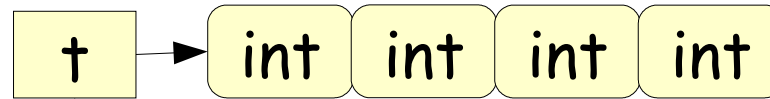


int * : points to one
or more ints (table
of ints).



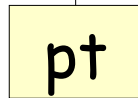
More Pointer Fun!

```
int table[4];  
int *t = table;
```

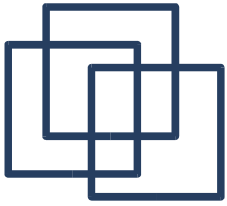


`int *` : points to one or more ints (table of ints).

```
int **pt = &t;
```

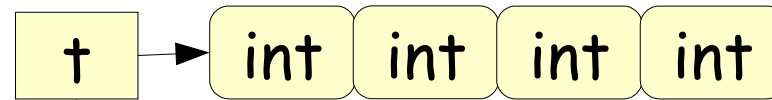


`int **` : points to one or more `int*` (table of `int*`).



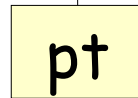
More Pointer Fun!

```
int table[4];  
int *t = table;
```



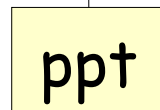
`int *` : points to one or more ints (table of ints).

```
int **pt = &t;
```

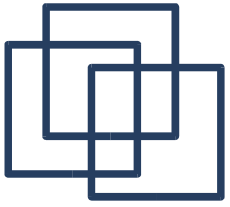


`int **` : points to one or more `int*` (table of `int*`).

```
int ***ppt = &pt;
```



`int ***` : ... you get it now!



Pointers: Example

```
void swap(int *x, int *y)
{
    int z = *x;
    *x = *y;
    *y = z;
}
```

...

```
int a = 1;
```

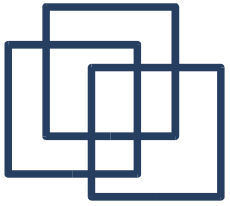
```
int b = 2;
```

...

```
swap(&a, &b);
```

...

a=1



Pointers: Example

```
void swap(int *x, int *y)
{
    int z = *x;
    *x = *y;
    *y = z;
}
```

...

```
int a = 1;
```

```
int b = 2;
```

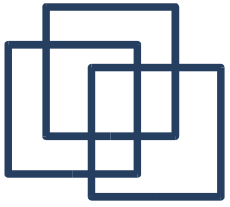
a=1

b=2

...

```
swap(&a, &b);
```

...



Pointers: Example

```
void swap(int *x, int *y)
```

```
{  
    int z = *x;  
    *x = *y;  
    *y = z;  
}
```

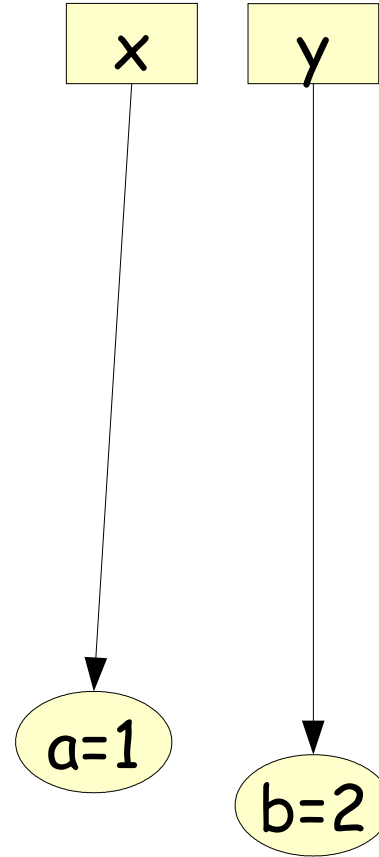
...

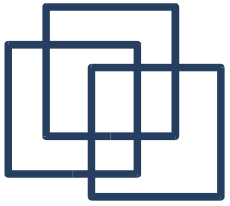
```
int a = 1;  
int b = 2;
```

...

```
swap(&a, &b);
```

...





Pointers: Example

```
void swap(int *x, int *y)
{
    int z = *x;
    *x = *y;
    *y = z;
}
```

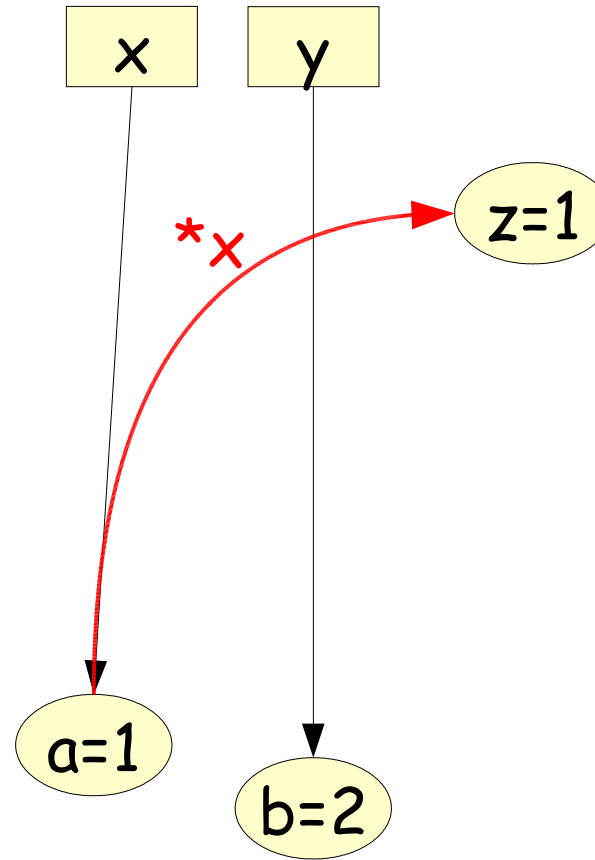
...

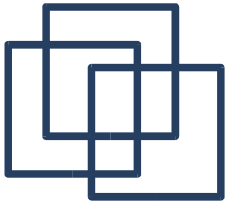
```
int a = 1;
int b = 2;
```

...

```
swap(&a, &b);
```

...





Pointers: Example

```
void swap(int *x, int *y)
{
    int z = *x;
    *x = *y;
    *y = z;
}
```

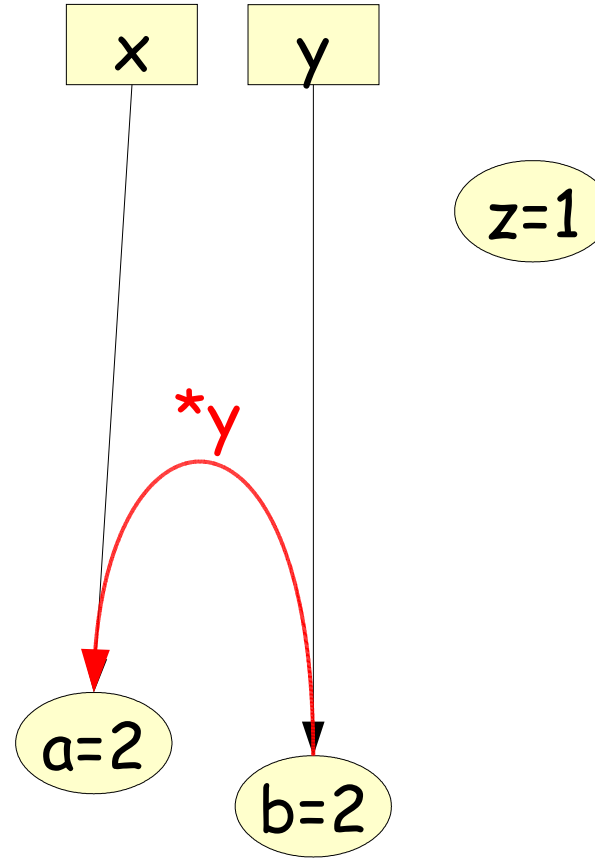
...

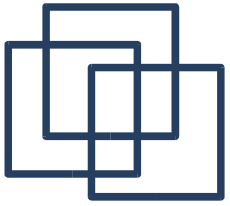
```
int a = 1;
int b = 2;
```

...

```
swap(&a, &b);
```

...





Pointers: Example

```
void swap(int *x, int *y)
{
    int z = *x;
    *x = *y;
    *y = z;
}
```

...

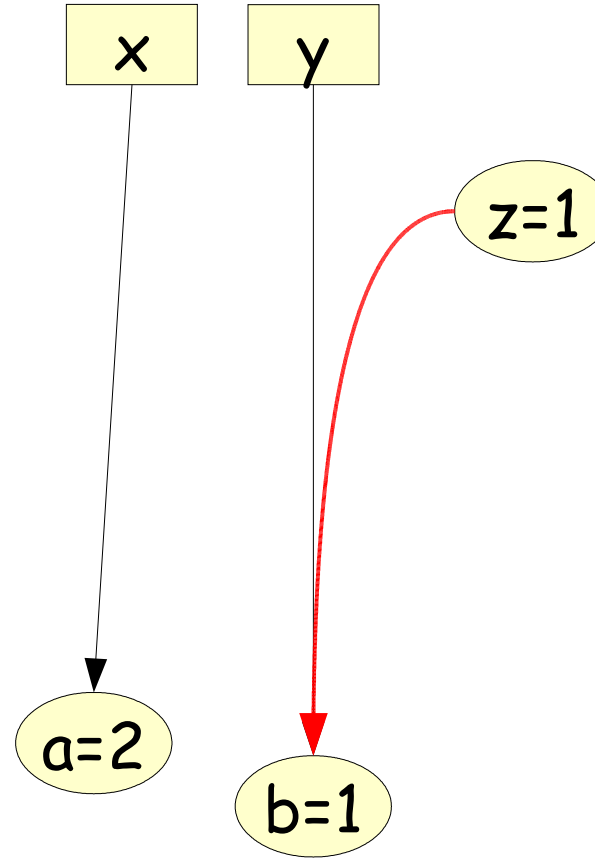
```
int a = 1;
```

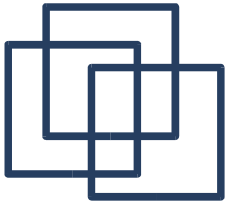
```
int b = 2;
```

...

```
swap(&a, &b);
```

...





Pointer Arithmetics

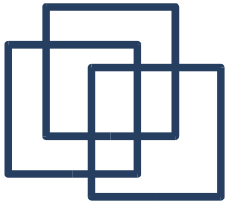
- Addition: **pointer + index -> pointer**

`int *p;` "`p+3`" same as "`&p[3]`" -> `int*`

- Subtraction: **pointer - pointer -> index**

`int a[4], *p = &a[1], *q = &a[3];`

`"q-p" == 2 -> int`



Pointer Arithmetics

- Addition: **pointer + index -> pointer**

`int *p;` "`p+3`" same as "`&p[3]`" -> `int*`

- Subtraction: **pointer - pointer -> index**

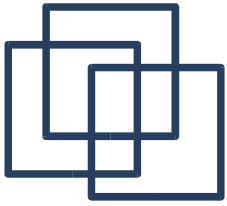
`int a[4], *p = &a[1], *q = &a[3];`

`"q-p" == 2 -> int`

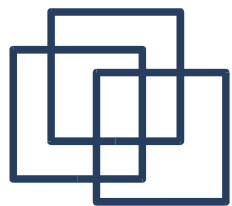
- **Pointers are typed!**

`int *p;` "`p+1`" points to next integer!

`char *c;` "`c+1`" points to next character!



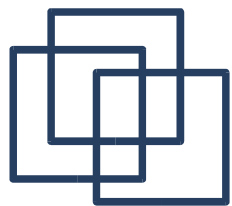
Big-O Notation in a Nutshell



Big-O Notation in a Nutshell

$$T(n) \in O(f(n))$$

means $T(N)$ is upper bounded by $f(n)$
(at a multiplicative constant) for n
"big enough".

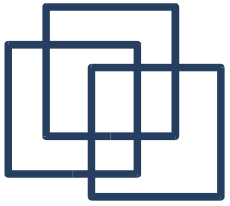


Big-O Notation in a Nutshell

$$T(n) \in O(f(n))$$

means $T(N)$ is upper bounded by $f(n)$
(at a multiplicative constant) for n
"big enough".

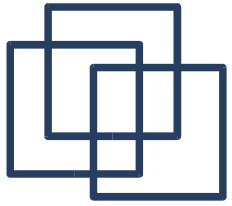
Characterizes the way the
complexity of the computation
grows depending on the size
of the problem.



Big-O Notation Example

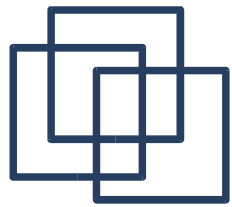
```
void copy(int *a, int *b, int n)
{
    int i;
    for(i = 0; i < n; ++i)
        b[i] = a[i];
}
```

Complexity: $O(n)$



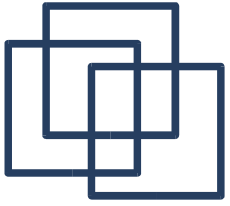
Big-O Notation Summary

Notation	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \cdot \log n)$	Pseudo-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^c)$	Polynomial
$O(c^n)$	Exponential
$O(n!)$	Factorial

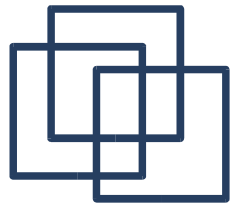


Big-O Notation Summary

Notation	Name	
$O(1)$	Constant	Very Easy
$O(\log n)$	Logarithmic	
$O(n)$	Linear	Easy
$O(n \cdot \log n)$	Pseudo-linear	
$O(n^2)$	Quadratic	Hard
$O(n^3)$	Cubic	
$O(n^c)$	Polynomial	
$O(c^n)$	Exponential	Very Hard
$O(n!)$	Factorial	



Elementary Data Structures



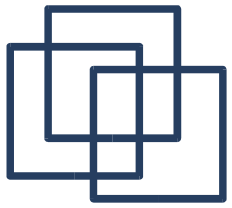
Choosing a Data Structure

Algorithms consume resources:

- Time (CPU Power)
- Space (Memory Space)

Choosing a data structure is a trade-off between time and space complexity:

- How fast can we access/modify the data ?
- How small can we code/compress the data ?
- ...



Common Operators

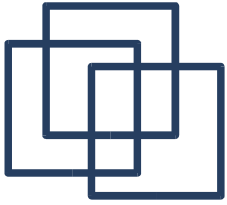
Basic Operators:

- `search(set, key)`
- `insert(set, key)`
- `delete(set, element)`

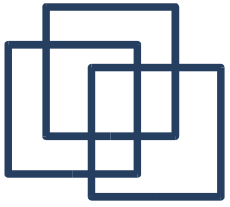
Other Operators:

- `sort(set)`
- `min(set), max(set)`
- `succ(set, elt), pred(set, elt)`
- `empty(set), count(set)`

Choose your data-structure depending
on what operations
will be performed the most !

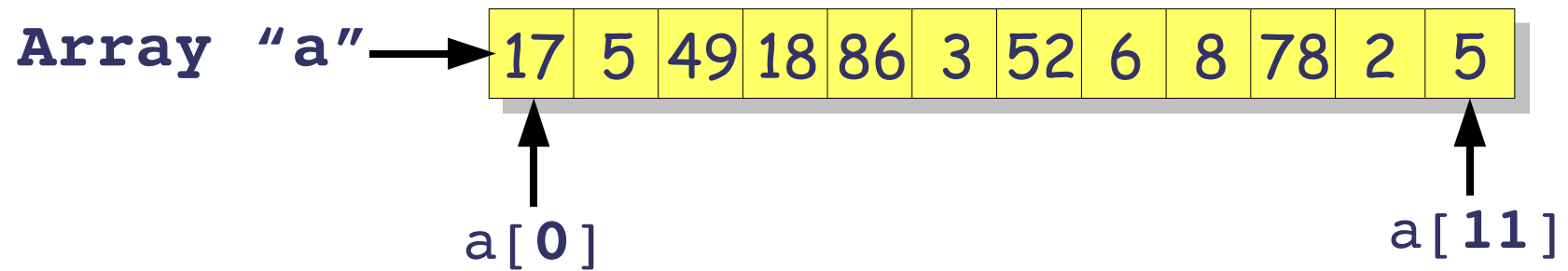


Arrays



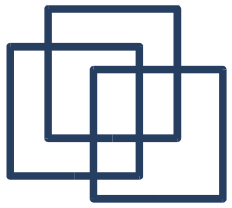
One-dimension Arrays

Declaration: `int a[12];`



$a[i] = *(a+i)$

base address index

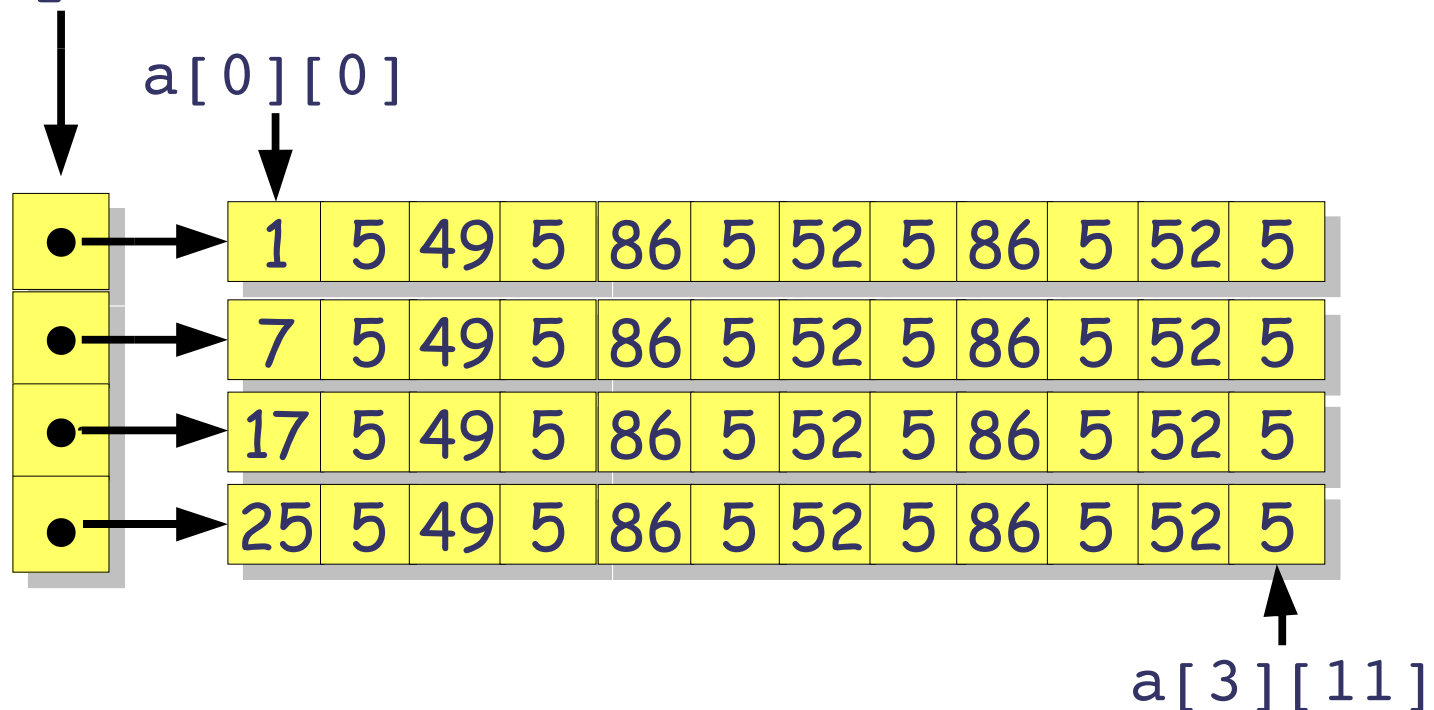


Two-dimensions Arrays

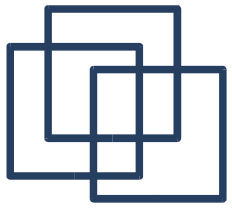
Declaration:

```
int a[4][12];
```

Array "a"



$a[i][j] = *(a[i] + j) = *((a + i) + j)$

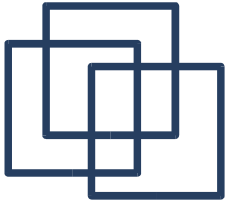


Complexity (Arrays)

Operator	Time	Space
search(set, key)	$O(1)$	$O(1)$
insert(set, key)	$O(n)$	$O(n)$
delete(set, key)	$O(n)$	$O(n)$
min(set) / max(set)	$O(n)$	$O(1)$
succ(set, elt) / pred(set, elt)	$O(n)$	$O(1)$
isempty(set)	$O(1)$	$O(1)$
count(set)	$O(1)$	$O(1)$

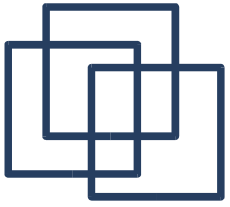
n is the number of elements in the array

key is the index of an element



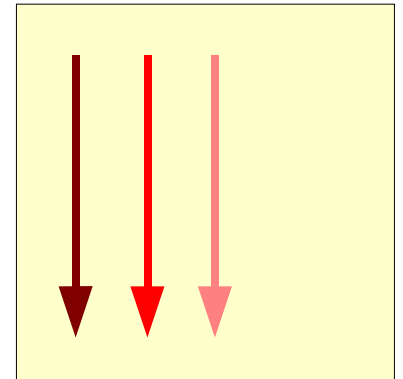
Matrix Copy

(why locality is good...)

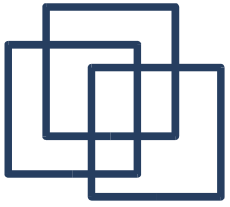


copy1

```
int copy1(float src[SIZE_X][SIZE_Y],  
          float dest[SIZE_X][SIZE_Y]) {  
    int i, j;  
  
    for (j=0; j<SIZE_Y; j++)  
        for (i=0; i<SIZE_X; i++)  
            dest[i][j] = src[i][j];  
  
    return 0;  
}
```

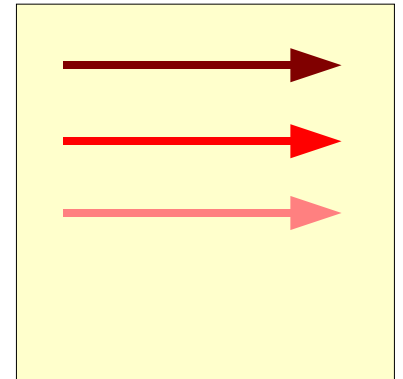


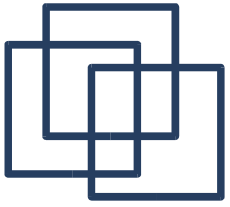
```
#define SIZE_X 20  
#define SIZE_Y 20
```



copy2

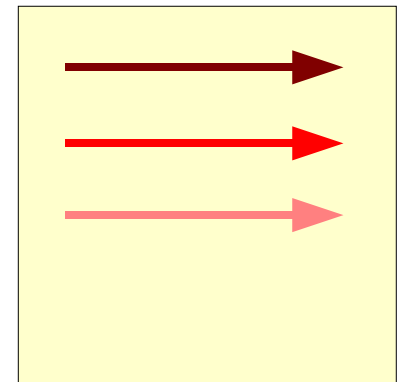
```
int copy2(float src[SIZE_X][SIZE_Y],  
          float dest[SIZE_X][SIZE_Y]) {  
    int i, j;  
  
    for (i=0; i<SIZE_X; i++)  
        for (j=0; j<SIZE_Y; j++)  
            dest[i][j] = src[i][j];  
  
    return 0;  
}
```

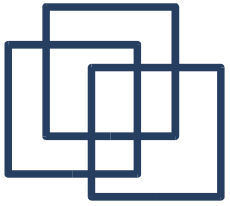




copy3

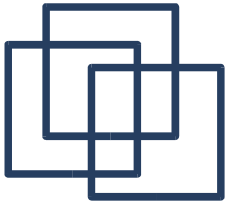
```
int copy3(float* src, float* dest) {  
    int size;  
  
    for (size=(SIZE_X*SIZE_Y); size; size--)  
        *dest++ = *src++;  
  
    return 0;  
}
```





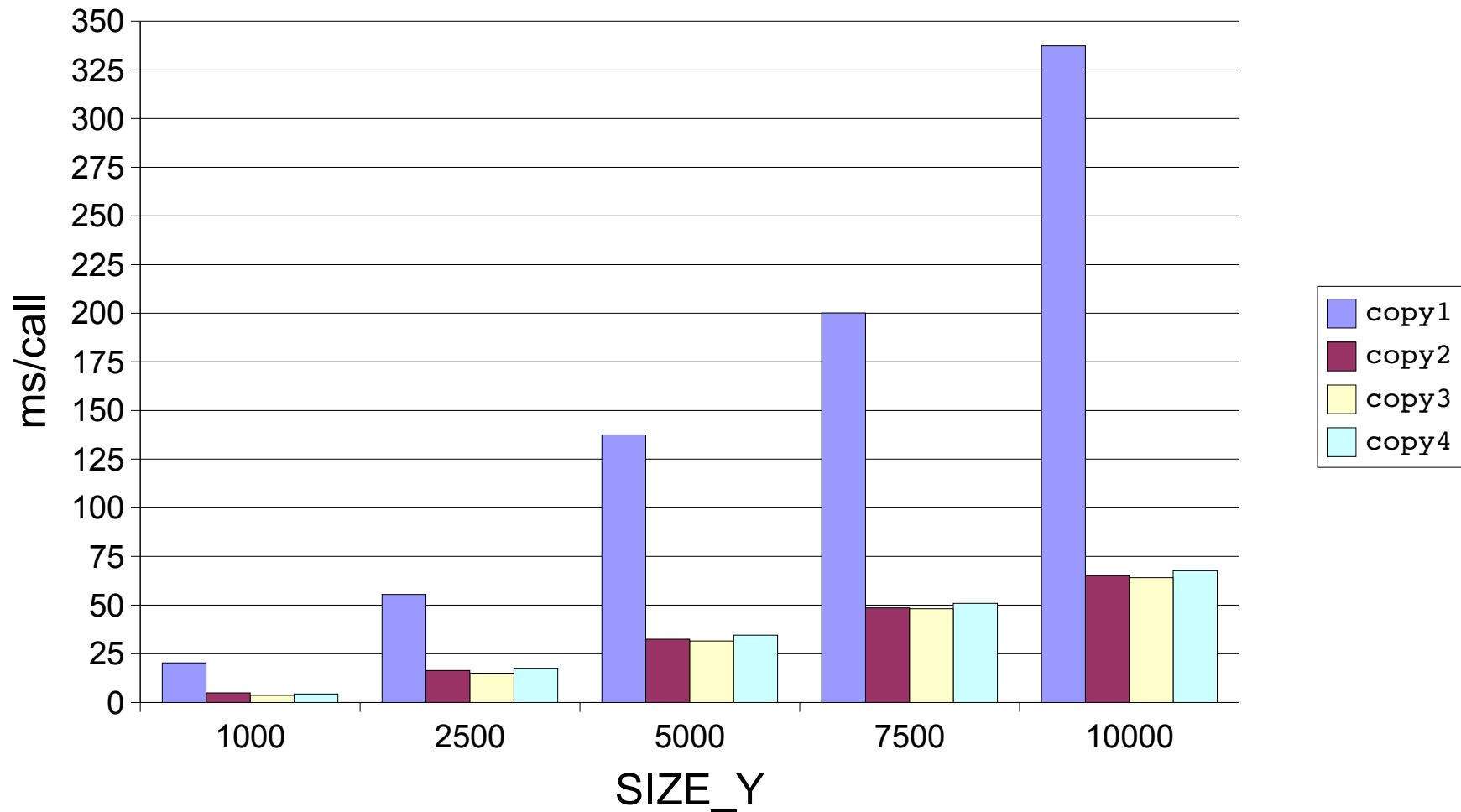
copy4

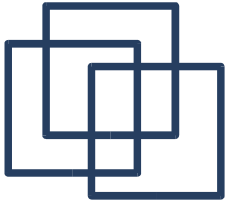
```
int copy4(float* src, float* dest) {  
    memcpy(dest, src,  
           (SIZE_X*SIZE_Y)*sizeof(float));  
  
    return 0;  
}
```



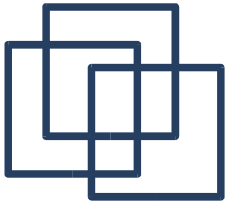
Performance

SIZE_X = 75

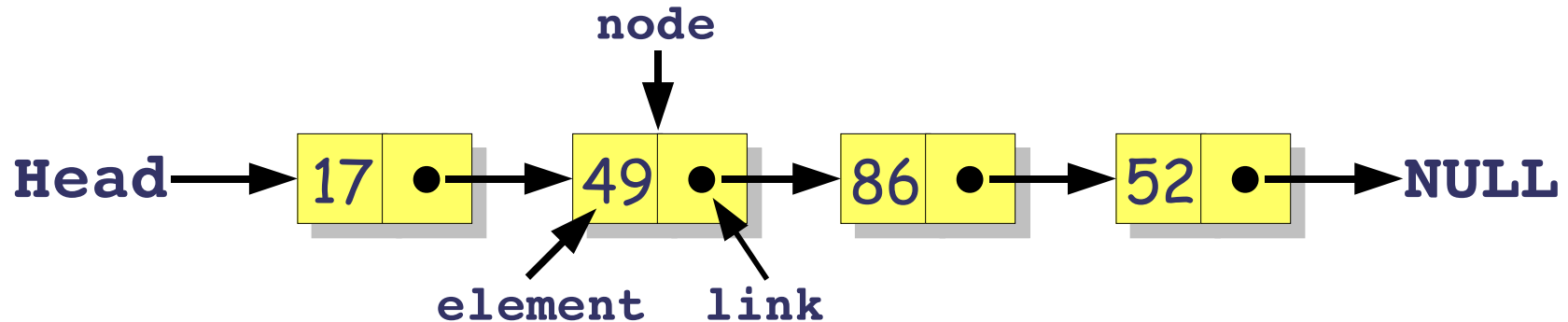




Linked-lists



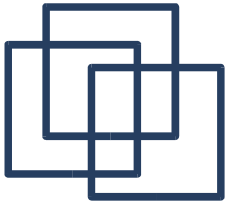
Singly Linked-lists



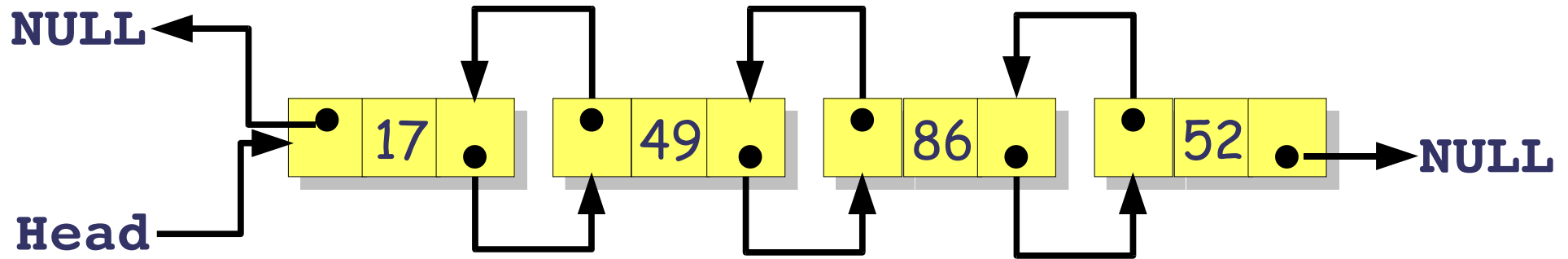
Declaration:

```
struct list {  
    int value;  
    struct list* next;  
};
```

Memory Overhead: One extra pointer for each element



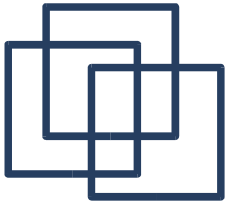
Doubly Linked-lists



Declaration:

```
struct list {  
    struct list* pred;  
    int value;  
    struct list* next;  
};
```

Memory Overhead: Two extra pointers for each element

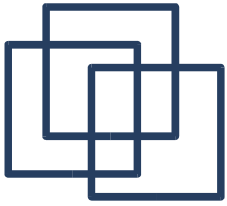


Search

Requirements:

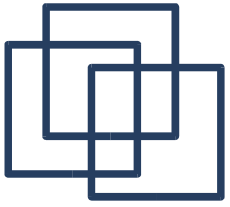
- Return a pointer to the right node
- Return NULL if not found

```
struct list* search(struct list *head,  
                    const int key)
```



Search

```
struct list* search(struct list *head,  
                    const int key) {  
  
    while ((head != NULL)&&(head->value != key))  
        head = head->next;  
  
    return head;  
}
```

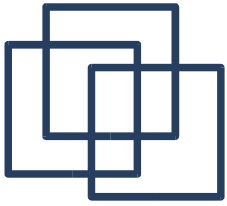


Insert

Requirements:

- Return a pointer to the inserted node
- Return NULL in case the malloc fail
- Can handle the case where the list is NULL

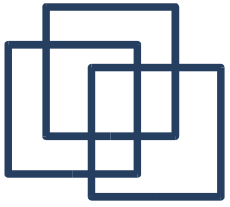
```
struct list* insert(struct list *head,  
                    const int value)
```



Insert

```
struct list* insert(struct list *head, const int value) {  
    struct list *tmp, *new;  
  
    new = (struct list*) malloc(sizeof(struct list));  
  
    if (new != NULL) {  
        new->value = value;  
        new->next = head;  
    }  
  
    return new;  
}
```

also fine if head is null

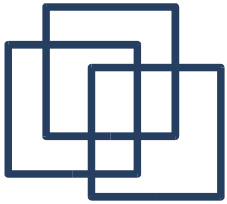


Delete

Requirements:

- Return a pointer to the head node
- Return NULL when deleting a singleton list
- Can handle the case where the list is NULL

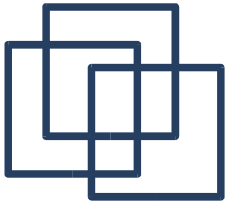
```
struct list* delete(struct list *head,  
                    const int value)
```



Delete

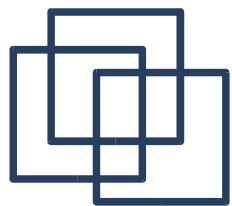
```
struct list* delete(struct list *head, const int value) {  
    struct list *current = head, *previous = head;  
  
    if (head != NULL) {  
        if (head->value != value) {  
            current = current->next;  
  
            while ((current != NULL) && (current->value != value)) {  
                previous = current;  
                current = current->next;  
            }  
            if (current != NULL)  
                previous->next = current->next;  
        } else  
            head = head->next;  
  
        free(current);  
    }  
    return head;  
}
```

The diagram shows a linked list with four nodes: 17, 49, 86, and 52. The 'head' pointer points to the first node (17). The 'previous' pointer points to the node containing 49. The 'current' pointer points to the node containing 86. The node containing 86 is being deleted, and its next pointer (which points to the node containing 52) is being redirected to 'cont.'. The node containing 52 is highlighted in red.



Delete

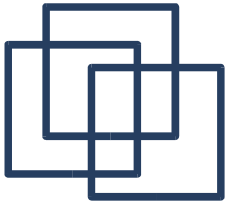
```
struct list* delete(struct list *head, const int value) {  
    struct list *current = head, *previous = head;  
  
    if (head != NULL) {  
        if (head->value != value) {  
            current = current->next;  
  
            while ((current != NULL) && (current->value != value)) {  
                previous = current;  
                current = current->next;  
            }  
            if (current != NULL)  
                previous->next = current->next;  
        } else  
            head = head->next;  
  
        free(current);  
    }  
    return head;  
}
```



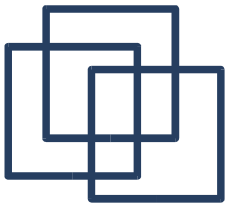
Complexity (Linked-lists)

Operator	Time	Space
search(set, key)	$O(n)$	$O(1)$
insert(set, key)	$O(1)$	$O(1)$
delete(set, key)	$O(n)$	$O(1)$
min(set) / max(set)	$O(n)$	$O(1)$
succ(set, elt) / pred(set, elt)	$O(n)$	$O(1)$
isempty(set)	$O(1)$	$O(1)$
count(set)	$O(n)$	$O(1)$

n is the number of elements in the linked-list
key is a value



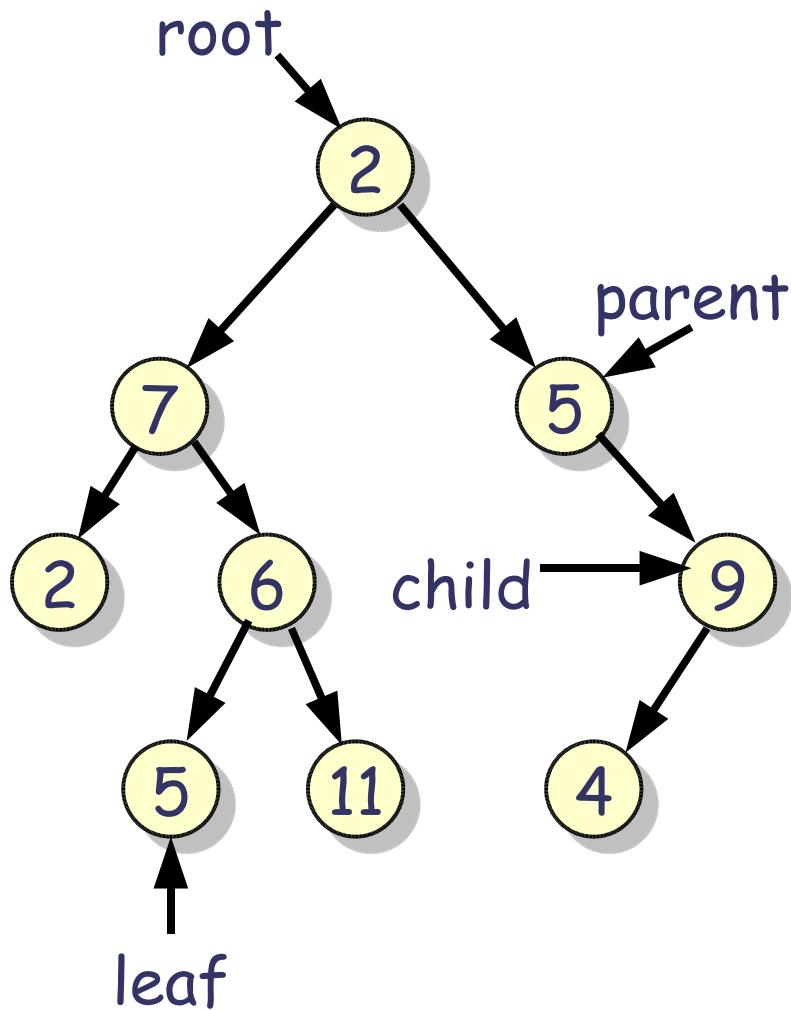
Trees



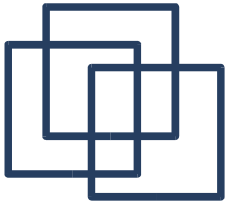
Trees

Definitions:

- Each node has **zero or more** children
- A node with a child is a **parent**
- A node without a child is a **leaf**
- A node without a parent is a **root**

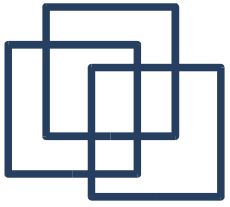


```
struct tree {  
    int value;  
    struct tree *left;  
    struct tree *right;  
};
```



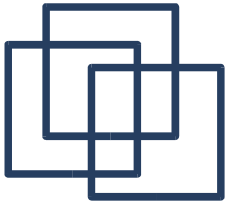
Pre-order

```
void traverse(struct tree *node) {  
  
    printf("Value is: %i\n", node->value);  
  
    if (node->left != NULL)  
        traverse(node->left);  
  
    if (node->right != NULL)  
        traverse(node->right);  
  
    return;  
}
```



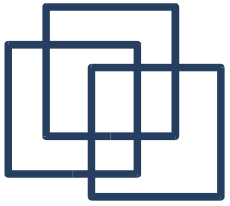
In-order

```
void traverse(struct tree *node) {  
  
    if (node->left != NULL)  
        traverse(node->left);  
  
    printf("Value is: %i\n", node->value);  
  
    if (node->right != NULL)  
        traverse(node->right);  
  
    return;  
}
```

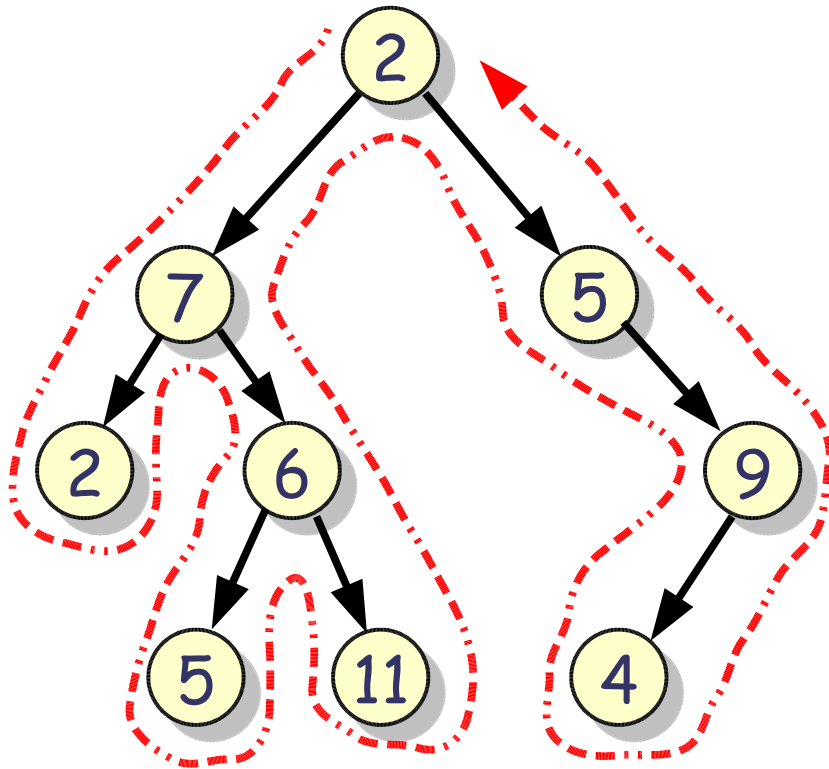


Post-order

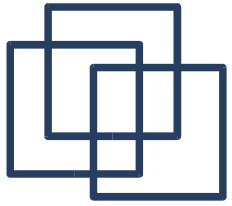
```
void traverse(struct tree *node) {  
  
    if (node->left != NULL)  
        traverse(node->left);  
  
    if (node->right != NULL)  
        traverse(node->right);  
  
    printf("Value is: %i\n", node->value);  
  
    return;  
}
```



Tree Traversal



- Pre-order:
2, 7, 2, 6, 5, 11, 5, 9, 4
- Post-order:
2, 5, 11, 6, 7, 4, 9, 5, 2
- In-order:
2, 7, 5, 6, 11, 2, 5, 4, 9



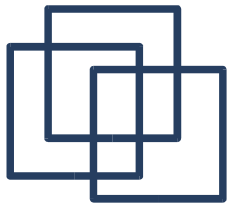
Operators on Trees

- Breadth-first Operators:

The operation on the node is applied before exploring the children

- Depth-first Operators:

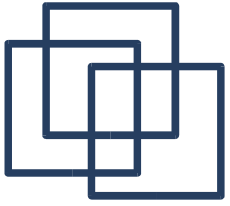
The operation on the node is applied after exploring the children



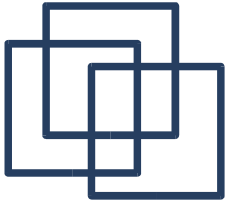
Complexity (Trees)

Operator	Time	Space
search(set, key)	$O(n)$	$O(1)$
insert(set, key)	$O(n)$	$O(1)$
delete(set, key)	$O(n)$	$O(1)$
min(set) / max(set)	$O(n)$	$O(1)$
succ(set, elt)/pred(set, elt)	$O(n)$	$O(1)$
isempty(set)	$O(1)$	$O(1)$
count(set)	$O(n)$	$O(1)$

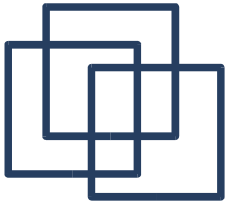
n is the number of elements in the tree



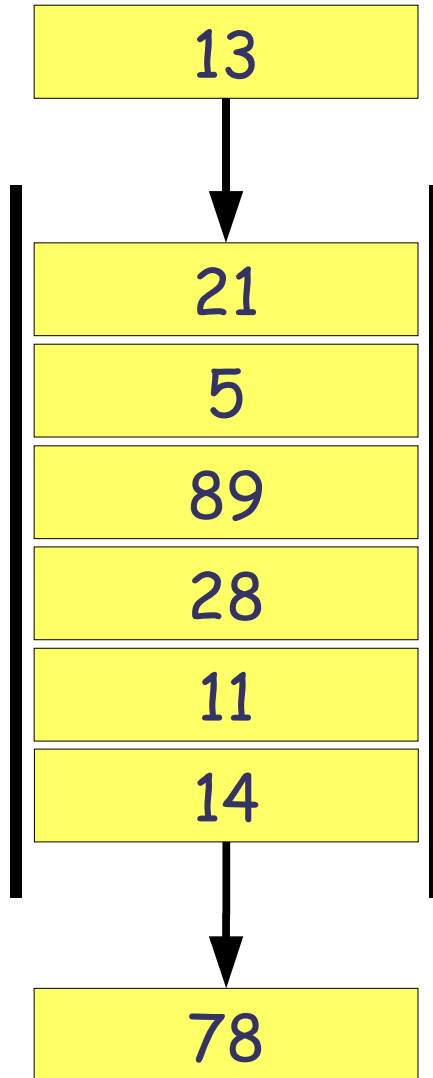
Some Usual Data Structures



Queues (FIFO)

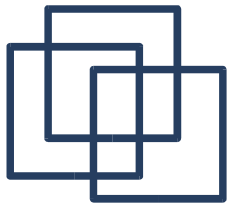


Queues (FIFO)



First In First Out

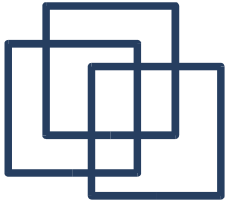
- Implemented via:
 - Array
 - Linked-list
- Applications:
 - Buffers and Spoolers
(networks, printer, scheduler, ...)



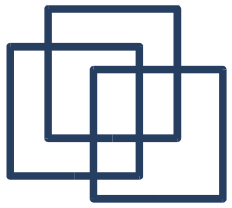
Complexity (Queues)

Operator	Time	Space
search(set, key)	$O(n)$	$O(n)$
insert(set, key)	$O(1)$	$O(1)$
delete(set, key)	$O(n)$	$O(n)$
min(set) / max(set)	$O(n)$	$O(n)$
succ(set, elt) / pred(set, elt)	$O(n)$	$O(n)$
isempty(set)	$O(1)$	$O(1)$
count(set)	$O(n)$	$O(n)$

n is the number of elements in the queue
key is a value



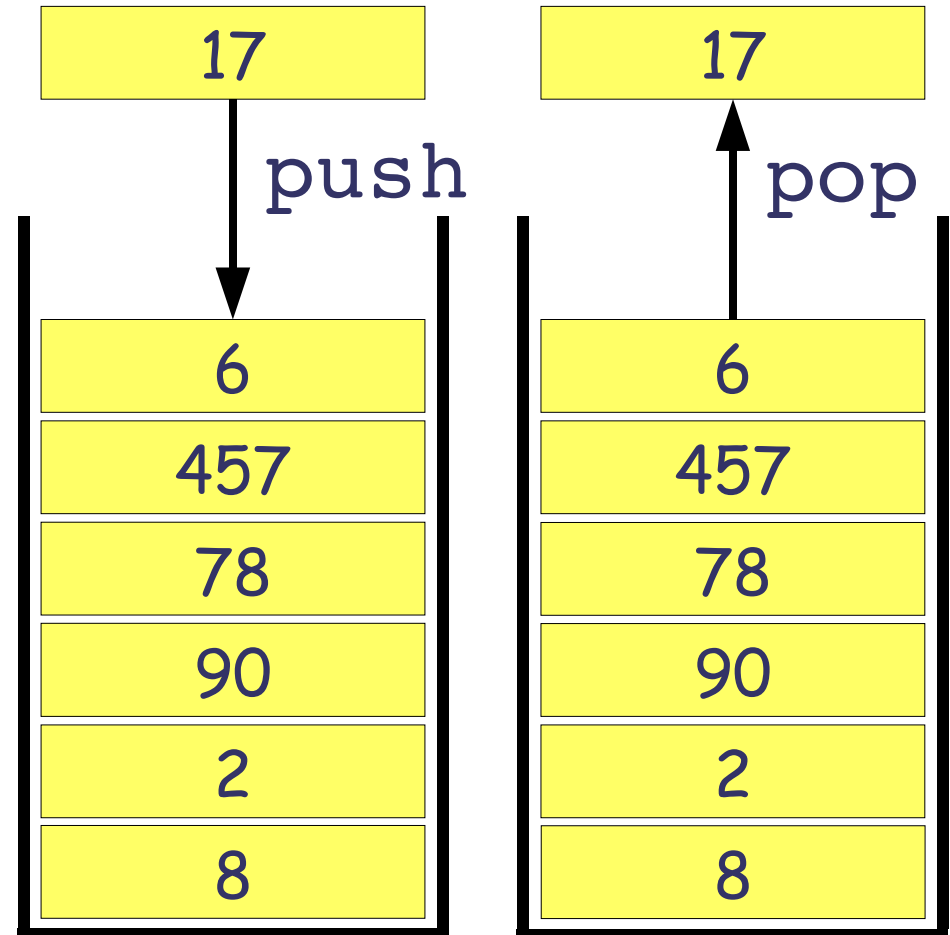
Stacks (FILO)

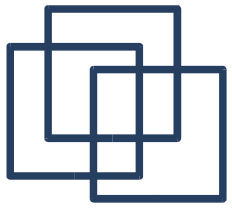


Stacks (FILO)

First In Last Out

- Implemented via:
 - Array
 - Linked-list
- Applications:
 - Stack based calculus (CPU, parser, ...)

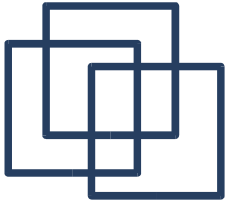




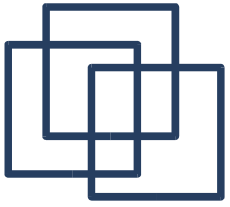
Complexity (Stacks)

Operator	Time	Space
search(set, key)	$O(n)$	$O(n)$
insert(set, key)	$O(1)$	$O(1)$
delete(set, key)	$O(n)$	$O(n)$
min(set) / max(set)	$O(n)$	$O(n)$
succ(set, elt) / pred(set, elt)	$O(n)$	$O(n)$
isempty(set)	$O(1)$	$O(1)$
count(set)	$O(n)$	$O(n)$

n is the number of elements in the stack
key is a value



Binary Heaps

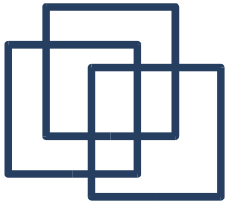


Binary Heaps

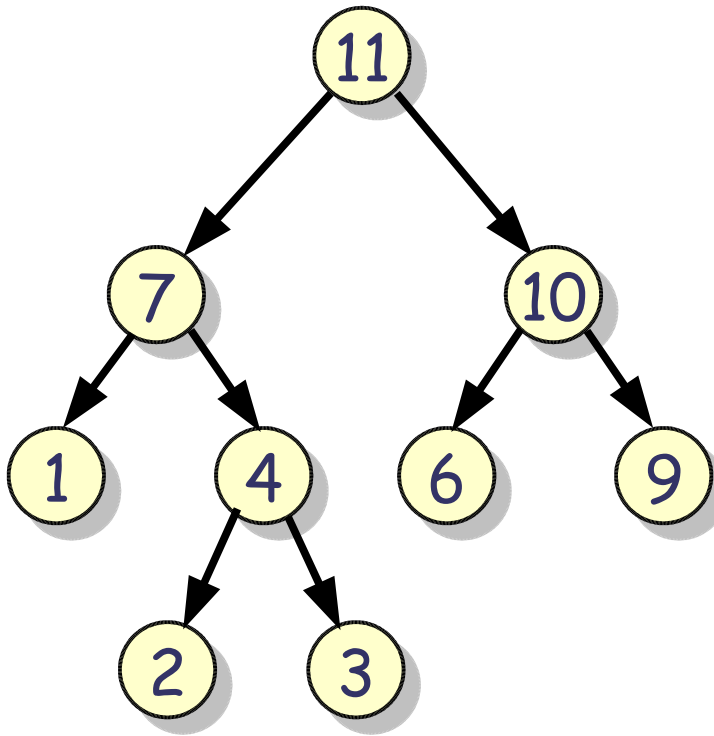
A **heap** is a tree structure such that, if A and B are nodes of a heap and B is a child of A , then:

$$\text{key}(A) \geq \text{key}(B)$$

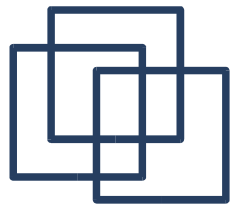
A **binary heap** is a heap based on a **binary tree**



Binary Heaps



- Implemented via:
 - Array
($a[i]$ has two children $a[2i+1], a[2i+2]$)
 - Tree
- Applications:
 - Quick access to data
(database, ...)

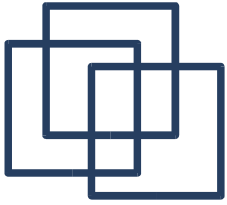


Complexity (Binary Heaps)

Operator	Time	Space
search(set, key)	$O(\log n)$	$O(1)$
insert(set, key)	$O(\log n)$	$O(1)$
delete(set, key)	$O(\log n)$	$O(1)$
min(set) / max(set)	$O(\log n)$	$O(1)$
succ(set, elt) / pred(set, elt)	$O(\log n)$	$O(1)$
isempty(set)	$O(1)$	$O(1)$
count(set)	$O(n)$	$O(1)$

n is the number of elements in the heap

key is a value



Questions ?