INF539: refresher basic data structures in C

F. Morain

Laboratoire d'Informatique de l'École polytechnique

September 17th, 2020 – version 1.1

Plan

Good reading: Cormen/Leiserson/Rivest *Introduction to Algorithms*; also, X lecture notes (using Java) for INF361, INF411, INF421 (theoretical).

I. Arrays

Z) Warming up

```c
#include <stdio.h>

int main(int argc, char *argv[]){
    int a[10];
    int i;

    for(i = 0; i < 10; i++)
       a[i] = i;
    printf("a =");
    for(i = 0; i < 10; i++)
       printf(" %d", a[i]);
    printf("\n");
    return 0;
}
```

Array: dynamic allocation

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

int main(int argc, char *argv[]){
    int *a = (int *)malloc(10 * sizeof(int));
    int i;

    for(i = 0; i < 10; i++)
       a[i] = i;
    printf("a =");
    for(i = 0; i < 10; i++)
       printf(" %d", a[i]);
    printf("\n");
    free(a);
    return 0;
}
```


A) Insertion sort

Principle: we suppose that `t[0..i[` is already sorted and we insert `t[i]` where it belongs.

t	9	2	5	13	4	2	6	8	10

Program:

```c
static void insertionSort(int[] t, int n){
  for(int i = 1; i < n; i++){
    // t[0..i-1] is sorted
    // we look for the right place for t[i]
    int j = findPlace(t, i);
    // finish
    insert(t, i, j);
  }
}
```


A) Insertion sort

Principle: we suppose that `t[0..i[` is already sorted and we insert `t[i]` where it belongs.

t	9	2	5	13	4	2	6	8	10

⇑

Program:

```c
static void insertionSort(int[] t, int n){
  for(int i = 1; i < n; i++){
    // t[0..i-1] is sorted
    // we look for the right place for t[i]
    int j = findPlace(t, i);
    // finish
    insert(t, i, j);
  }
}
```


A) Insertion sort

Principle: we suppose that `t[0..i[` is already sorted and we insert `t[i]` where it belongs.

t	9	2	5	13	4	2	6	8	10

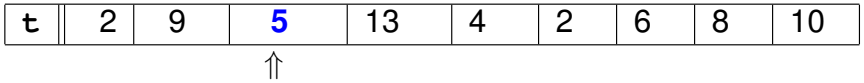
⇑

Program:

```c
static void insertionSort(int[] t, int n){
  for(int i = 1; i < n; i++){
    // t[0..i-1] is sorted
    // we look for the right place for t[i]
    int j = findPlace(t, i);
    // finish
    insert(t, i, j);
  }
}
```

A) Insertion sort

Principle: we suppose that `t[0..i[` is already sorted and we insert `t[i]` where it belongs.

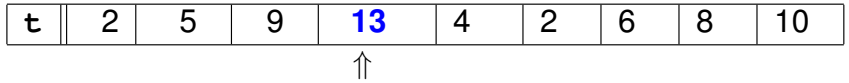


Program:

```
static void insertionSort(int[] t, int n){
  for(int i = 1; i < n; i++){
    // t[0..i-1] is sorted
    // we look for the right place for t[i]
    int j = findPlace(t, i);
    // finish
    insert(t, i, j);
  }
}
```


A) Insertion sort

Principle: we suppose that `t[0..i[` is already sorted and we insert `t[i]` where it belongs.

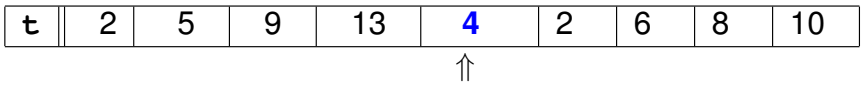


Program:

```
static void insertionSort(int[] t, int n){
  for(int i = 1; i < n; i++){
    // t[0..i-1] is sorted
    // we look for the right place for t[i]
    int j = findPlace(t, i);
    // finish
    insert(t, i, j);
  }
}
```


A) Insertion sort

Principle: we suppose that `t[0..i[` is already sorted and we insert `t[i]` where it belongs.

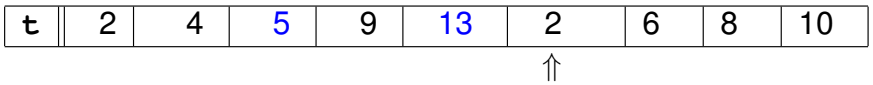


Program:

```
static void insertionSort(int[] t, int n){
  for(int i = 1; i < n; i++){
    // t[0..i-1] is sorted
    // we look for the right place for t[i]
    int j = findPlace(t, i);
    // finish
    insert(t, i, j);
  }
}
```


A) Insertion sort

Principle: we suppose that `t[0..i[` is already sorted and we insert `t[i]` where it belongs.

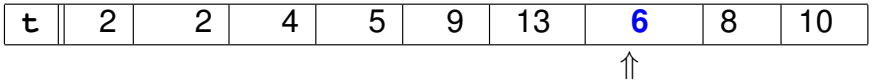


Program:

```
static void insertionSort(int[] t, int n){
  for(int i = 1; i < n; i++){
    // t[0..i-1] is sorted
    // we look for the right place for t[i]
    int j = findPlace(t, i);
    // finish
    insert(t, i, j);
  }
}
```

A) Insertion sort

Principle: we suppose that `t[0..i[` is already sorted and we insert `t[i]` where it belongs.

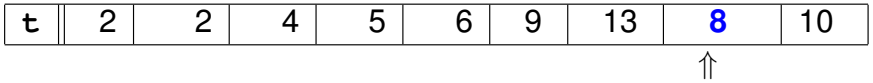


Program:

```
static void insertionSort(int[] t, int n){
  for(int i = 1; i < n; i++){
    // t[0..i-1] is sorted
    // we look for the right place for t[i]
    int j = findPlace(t, i);
    // finish
    insert(t, i, j);
  }
}
```


A) Insertion sort

Principle: we suppose that `t[0..i[` is already sorted and we insert `t[i]` where it belongs.

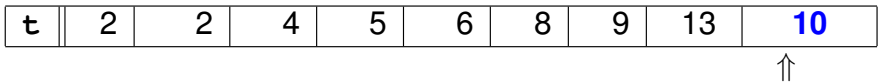


Program:

```
static void insertionSort(int[] t, int n){
  for(int i = 1; i < n; i++){
    // t[0..i-1] is sorted
    // we look for the right place for t[i]
    int j = findPlace(t, i);
    // finish
    insert(t, i, j);
  }
}
```


A) Insertion sort

Principle: we suppose that `t[0..i[` is already sorted and we insert `t[i]` where it belongs.



Program:

```
static void insertionSort(int[] t, int n){
  for(int i = 1; i < n; i++){
    // t[0..i-1] is sorted
    // we look for the right place for t[i]
    int j = findPlace(t, i);
    // finish
    insert(t, i, j);
  }
}
```


A) Insertion sort

Principle: we suppose that `t[0..i[` is already sorted and we insert `t[i]` where it belongs.

Program:

```
static void insertionSort(int[] t, int n){
  for(int i = 1; i < n; i++){
    // t[0..i-1] is sorted
    // we look for the right place for t[i]
    int j = findPlace(t, i);
    // finish
    insert(t, i, j);
  }
}
```

The program (2/3)

```
// t[0..i-1] is sorted
// look for the place of t[i] defined by
// j s.t. t[j] <= t[i] < t[j+1]
static int findPlace(int[] t, int i){
  int j;

  for(j = i-1; j >= 0; j--)
    if(t[j] <= t[i])
      break;
  // if j < 0, t[i] < t[0], we return 0
  // if t[j] <= t[i] < t[j+1], we return j+1
  return (j+1);
}
```


The program (3/3)

```
static void insert(int[] t, int i, int j){
  int tmp = t[i], k; // copy

  // t[0..j-1][j..i-1] -> t[0..j-1][][j+1..i]
  for(k = i; k > j; k--)
    t[k] = t[k-1];
  t[j] = tmp;
}
```

Rem. We can find a more compact code using careful programming.

Rem. See `man qsort` for programs using the `libc` version of quicksort.

The program (3/3)

```
static void insert(int[] t, int i, int j){
  int tmp = t[i], k; // copy

  // t[0..j-1][j..i-1] -> t[0..j-1][][j+1..i]
  for(k = i; k > j; k--)
    t[k] = t[k-1];
  t[j] = tmp;
}
```

Rem. We can find a more compact code using careful programming.

Rem. See `man qsort` for programs using the `libc` version of quicksort.

B) Eratosthenes and beyond

Eratosthenes: how do we enumerate prime numbers in $[2, X]$?

It is much easier to enumerate composite numbers and then deduce the primes as being not composite.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20

Prop. A composite number C has a prime factor $\leq \sqrt{C}$.

B) Eratosthenes and beyond

Eratosthenes: how do we enumerate prime numbers in $[2,X]$?

It is much easier to enumerate composite numbers and then deduce the primes as being not composite.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20

Prop. A composite number C has a prime factor $\leq \sqrt{C}$.

B) Eratosthenes and beyond

Eratosthenes: how do we enumerate prime numbers in $[2,X]$?

It is much easier to enumerate composite numbers and then deduce the primes as being not composite.

	2	3	4	5	6	7	8	9	10
			2						
11	12	13	14	15	16	17	18	19	20

Prop. A composite number C has a prime factor $\leq \sqrt{C}$.

B) Eratosthenes and beyond

Eratosthenes: how do we enumerate prime numbers in $[2,X]$?

It is much easier to enumerate composite numbers and then deduce the primes as being not composite.

	2	3	4	5	6	7	8	9	10
			2		**2**				
11	12	13	14	15	16	17	18	19	20

Prop. A composite number C has a prime factor $\leq \sqrt{C}$.

B) Eratosthenes and beyond

Eratosthenes: how do we enumerate prime numbers in $[2,X]$?

It is much easier to enumerate composite numbers and then deduce the primes as being not composite.

	2	3	4	5	6	7	8	9	10
			2		**2**		**2**		
11	12	13	14	15	16	17	18	19	20

Prop. A composite number C has a prime factor $\leq \sqrt{C}$.

B) Eratosthenes and beyond

Eratosthenes: how do we enumerate prime numbers in $[2,X]$?

It is much easier to enumerate composite numbers and then deduce the primes as being not composite.

	2	3	4	5	6	7	8	9	10
			2		**2**		**2**		**2**
11	12	13	14	15	16	17	18	19	20
	2		**2**		**2**		**2**		**2**

Prop. A composite number C has a prime factor $\leq \sqrt{C}$.

B) Eratosthenes and beyond

Eratosthenes: how do we enumerate prime numbers in $[2,X]$?

It is much easier to enumerate composite numbers and then deduce the primes as being not composite.

	2	**3**	4	5	6	7	8	9	10
			2		**2**		**2**		**2**
11	12	13	14	15	16	17	18	19	20
	2		**2**		**2**		**2**		**2**

Prop. A composite number C has a prime factor $\leq \sqrt{C}$.

B) Eratosthenes and beyond

Eratosthenes: how do we enumerate prime numbers in $[2,X]$?

It is much easier to enumerate composite numbers and then deduce the primes as being not composite.

	2	**3**	4	5	6	7	8	9	10
			2		**2**		**2**	**3**	**2**
					3				
11	12	13	14	15	16	17	18	19	20
	2		**2**	**3**	**2**		**2**		**2**
	3						**3**		

Prop. A composite number C has a prime factor $\leq \sqrt{C}$.

B) Eratosthenes and beyond

Eratosthenes: how do we enumerate prime numbers in $[2,X]$?

It is much easier to enumerate composite numbers and then deduce the primes as being not composite.

	2	**3**	4	**5**	6	7	8	9	10
			2		**2**		**2**	**3**	**2**
					3				**5**
11	12	13	14	15	16	17	18	19	20
	2		**2**	**3**	**2**		**2**		**2**
	3			**5**			**3**		**5**

Prop. A composite number C has a prime factor $\leq \sqrt{C}$.

B) Eratosthenes and beyond

Eratosthenes: how do we enumerate prime numbers in $[2,X]$?

It is much easier to enumerate composite numbers and then deduce the primes as being not composite.

	2	**3**	4	**5**	6	**7**	8	9	10
			2		**2**		**2**	**3**	**2**
					3				**5**
11	12	13	14	15	16	17	18	19	20
	2		**2**	**3**	**2**		**2**		**2**
	3		**7**	**5**			**3**		**5**

Prop. A composite number C has a prime factor $\leq \sqrt{C}$.

B) Eratosthenes and beyond

Eratosthenes: how do we enumerate prime numbers in $[2,X]$?

It is much easier to enumerate composite numbers and then deduce the primes as being not composite.

	2	**3**	4	**5**	6	**7**	8	9	10
			2		**2**		**2**	**3**	**2**
					3				**5**
11	12	13	14	15	16	17	18	19	20
	2		**2**	**3**	**2**		**2**		**2**
	3		**7**	**5**			**3**		**5**

Prop. A composite number C has a prime factor $\leq \sqrt{C}$.

B) Eratosthenes and beyond

Eratosthenes: how do we enumerate prime numbers in $[2,X]$?

It is much easier to enumerate composite numbers and then deduce the primes as being not composite.

	2	**3**	4	**5**	6	**7**	8	9	10
			2		**2**		**2**	**3**	**2**
					3				**5**
11	12	**13**	14	15	16	**17**	18	**19**	20
	2		**2**	**3**	**2**		**2**		**2**
	3		**7**	**5**			**3**		**5**

Prop. A composite number C has a prime factor $\leq \sqrt{C}$.

- empty sets denote primes;
- non-empty sets contain the prime factors of the composite number.

Eratosthenes: algorithm

Sieve:

1. Set $T[x] = \emptyset$ for $x \in [2,X]$;
2. **for** $p = 2$ **to** $\lfloor \sqrt{X} \rfloor$ **do**
   if $T[p] = \emptyset$ **then** (*p *is prime* *)
   2.1 $x := 2p$;
   2.2 **while** $x \leq X$ **do** (**remove multiples of* p *)
   2.2.1 $T[x] := T[x] \cup \{p\}$;
   2.2.2 $x := x + p$.

Postsieve: find all $x \in [2..X]$ s.t. $T[x] = \emptyset$.

Rem. replace 2.2 by $x := p^2$; 2.3.2 by $x := x + 2p$ as soon as $p > 2$. Some tricks are available (Brent, etc.).

II. Linked lists

Many pictures: (1, 3, 2, 4) or
`-> 1 -> 3 -> 2 -> 4 -> null`.

Which type of object? we store pairs (**`int`**, arrow).
An arrow is the adress of the following pair.

$\Rightarrow$ a linked list is formed with links.

Recursive definition: linked list $= \emptyset$ or $(link, linkedlist)$.

Array or list?

We use an array when:

- one knows the size (or a close upper bound) in advance;
- one needs to access the different cells in a random order (direct access to **`t[i]`**).

We use a list when:

- one doesn't know the size *a priori*;
- one doesn't need to access the i-th element;
- one has to perform additions/suppressions of items in head or tail;
- one wants to save space.

Definitions in C

We mimic a module with specification/implementation.

File `list.h`:

```
/* basic type */
typedef struct __list {
    int c;
    struct __list *next;
} __list, *list;

/* exported functions */
extern list list_alloc();
extern list_create(int);
extern list_add_first(int, list);
extern int list_length(list L);
```


First functions

File `list.c`:

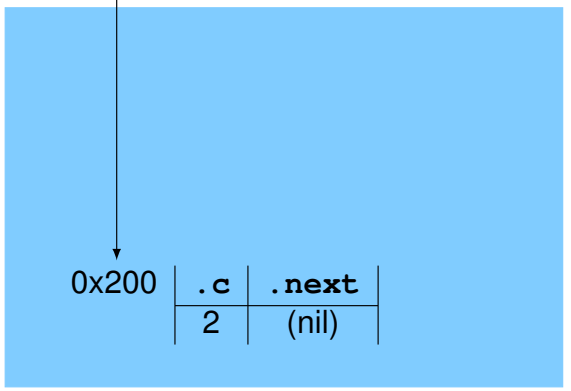
```
#include <stdlib.h> /* standard library for malloc
#include "list.h" /* we use our list module */

/* OUTPUT: -> */
list list_alloc(){
    return (list)malloc(sizeof(__list));
}
/* OUTPUT: -> c -> NULL */
list list_create(int c){
    list tmp = list_alloc();
    tmp->c = c;
    tmp->next = NULL;
    return tmp;
}
/* returns c -> L; cost O(1) */
list list_add_first(int c, list L){
    list tmp = list_alloc();
    tmp->c = c;
    tmp->next = L;
    return tmp;
}
```

What happens in memory

```
link L = NULL; //(nil)
```

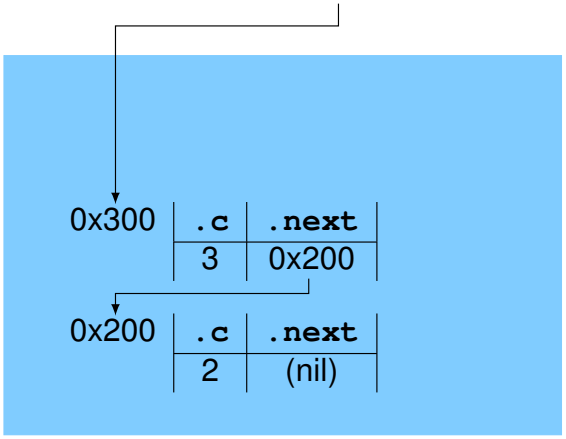

What happens in memory

```
link L = NULL; //(nil)
L = list_add_first(2, L); //0x200
```



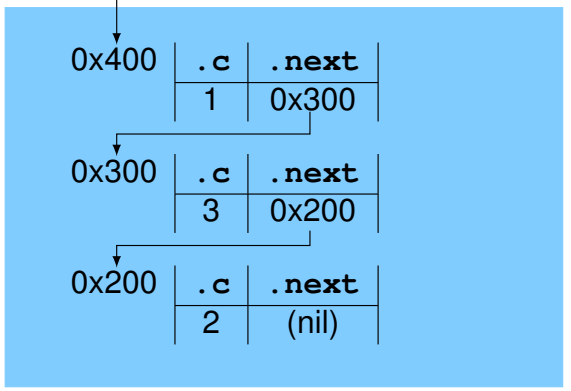
What happens in memory

```
link L = NULL; //(nil)
L = list_add_first(2, L); //0x200
L = list_add_first(3, L); //0x300
```



What happens in memory

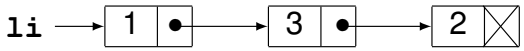
```
link L = NULL; //(nil)
L = list_add_first(2, L); //0x200
L = list_add_first(3, L); //0x300
L = list_add_first(1, L); //0x400
```



Other pictures

Any of:
li = (1, 3, 2)
or `li -> 1 -> 3 -> 2 -> null`

More abstract:



Closer to memory:

```
li -> 0x400:[1, 0x300] -> 0x300:[3, 0x200]
      -> 0x200:[2, (nil)] -> (nil)
```


Two basic functions

```
/* OUTPUT: length of the list */
int list_length(list L){
    if(L == NULL)
        return 0;
    else
        return 1 + list_length(L->next);
}

/* OUTPUT: if c is in L, return the first part of
list list_is_in(int c, list L){
    if(L == NULL || L->c == c)
        return L;
    return list_is_in(c, L->next);
}
```


Removing and freeing

```
void list_free(list L){
    if(L != NULL){
        list_free(L->next);
        free(L);
    }
}
/* [c, L] outputs L */
list list_remove_first(int *ptr_c, list L){
    list tmp;

    if(L == NULL){
        *ptr_c = -1; /* why not? */
        return L;
    }
    *ptr_c = L->c;
    tmp    = L->next;
    free(L);
    return tmp;
}
```


Complexity issues

Adding/deleting head element: $O(1)$.
Adding/deleting tail element: $O(n)$ if L has size n.

To do better: doubly linked list

```
L -> 1 <-> 3 <-> 2 <-> 4 -> NULL
```

Need modify the structure:

```
typedef struct __dlist {
    int c;
    struct __dlist *next, *prev;
} __dlist, *dlist;
```

Even better: double ended queue (dequeue) with a pointer on the head as well as on the tail.

Ex. write allocation, insertion/deletion head or tail that cost $O(1)$ operations.

Complexity issues

Adding/deleting head element: $O(1)$.
Adding/deleting tail element: $O(n)$ if L has size n.

To do better: doubly linked list

```
L -> 1 <-> 3 <-> 2 <-> 4 -> NULL
```

Need modify the structure:

```
typedef struct __dlist {
    int c;
    struct __dlist *next, *prev;
} __dlist, *dlist;
```

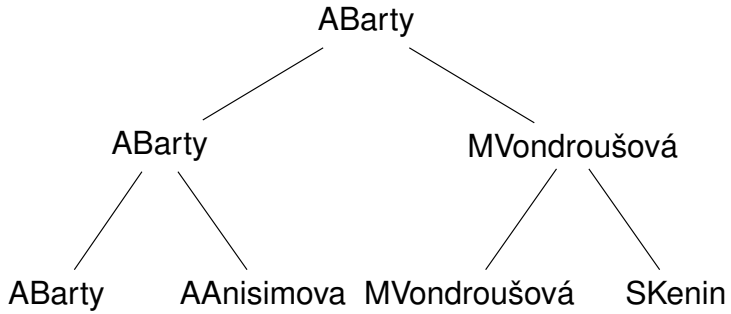
Even better: double ended queue (dequeue) with a pointer on the head as well as on the tail.

Ex. write allocation, insertion/deletion head or tail that cost $O(1)$ operations.

III. Trees

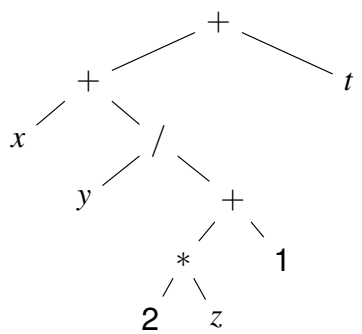
A) Introduction

Roland-Garros 2019:



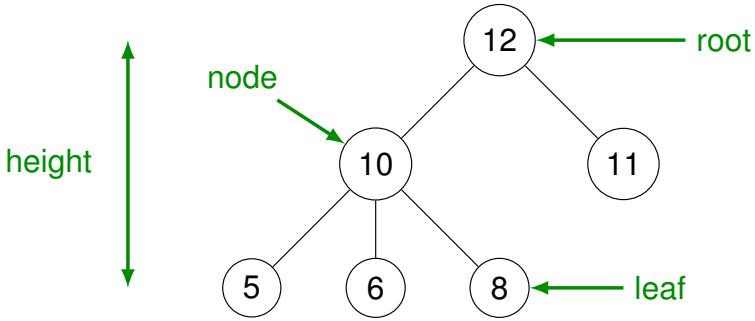
Another example

Representing $x+y/(2z+1)+t$:

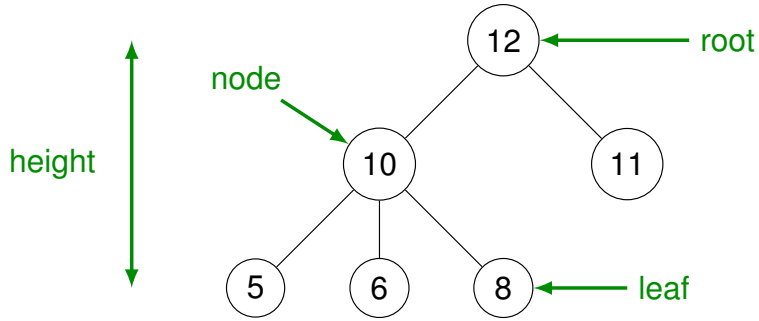


Useful also in compilation.

Terminology

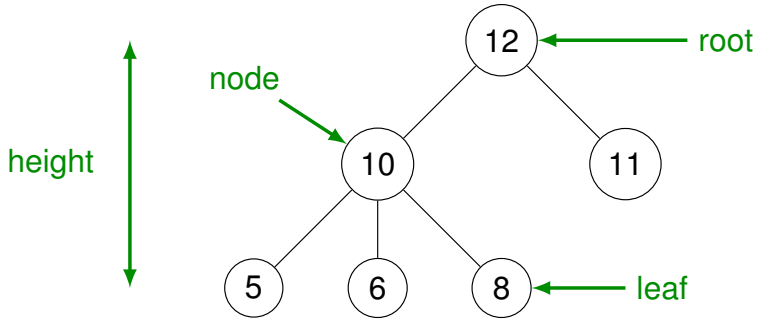


Terminology



The tree has 6 nodes: one root (12), 1 internal node (10), 4 leaves (5, 6, 8, 11); 10 is a child of 12; the tree has height 3 (the empty tree has height 0).

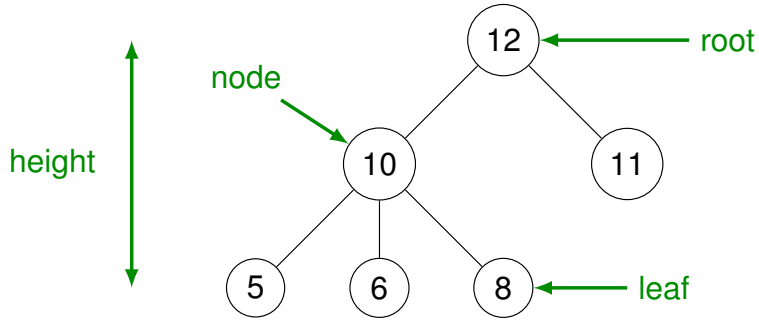
Terminology



The tree has 6 nodes: one root (12), 1 internal node (10), 4 leaves (5, 6, 8, 11); 10 is a child of 12; the tree has height 3 (the empty tree has height 0).

All nodes except the root have a parent, the leaves have no children.

Terminology



The tree has 6 nodes: one root (12), 1 internal node (10), 4 leaves (5, 6, 8, 11); 10 is a child of 12; the tree has height 3 (the empty tree has height 0).

All nodes except the root have a parent, the leaves have no children.

(12, 10, 6) or (12, 11) are paths in the tree.

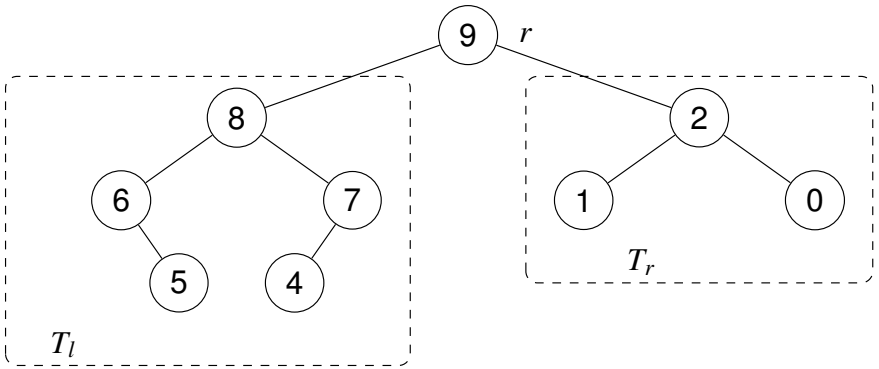
Required operations

- addition (insertion), suppression;
- searching for a given node;
- printing all nodes in hierarchical order (by level);
- following a branch (e.g., list of parents).

Rem. pictures, properties may not have any relationship with a given implementation.

B) Binary trees

Def. (recursive) $T = \emptyset$ or (r, T_l, T_r).



Dynamic implementation

Close to the recursive definition: $T = \emptyset$ ou (r, T_l, T_r).

File `tree.h`:

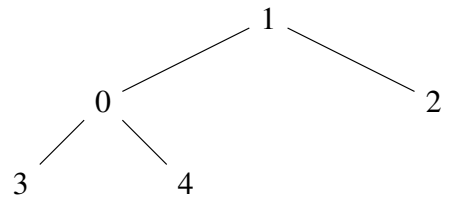
```
typedef struct node {
    int c;
    struct node *left, *right;
} node, *tree;
extern tree tree_alloc();
extern tree tree_create_leaf(int c);
```

File `tree.c`:

```
tree tree_alloc(){
    return (tree)malloc(sizeof(node));
}
tree tree_create_leaf(int c){
    tree T = tree_alloc();
    T->left = NULL; T->right = NULL; T->c = c;
    return T;
}
```

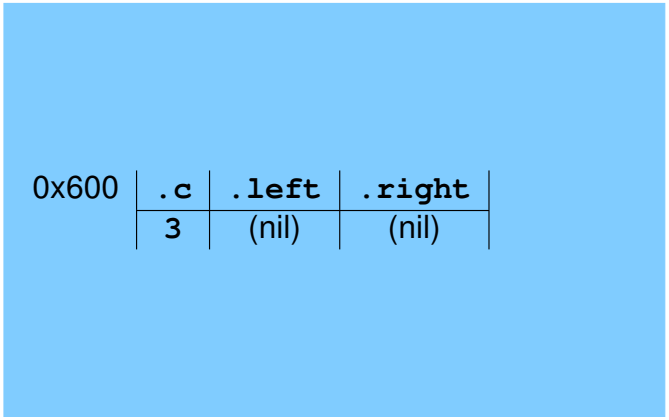

Tests

```
int main(int argc, char *argv[]){
    tree T = NULL, L, R, two;

    L = tree_create_leaf(3);
    R = tree_create_leaf(4);
    T = tree_join(0, L, R);
    two = tree_create_leaf(2);
    T = tree_join(1, T, two);
    return 0;
}
```



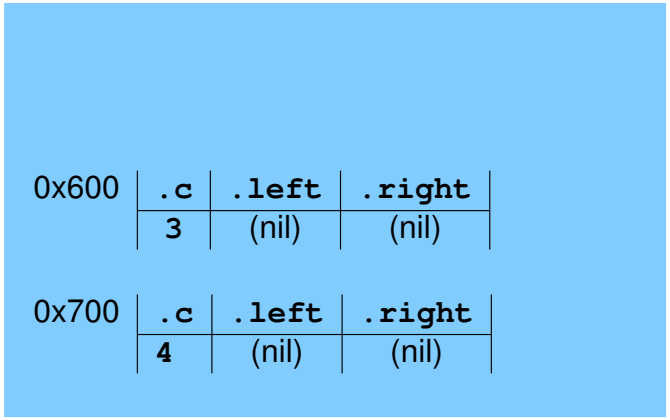
Memory picture

```
L = tree_create_leaf(3); //0x600
```



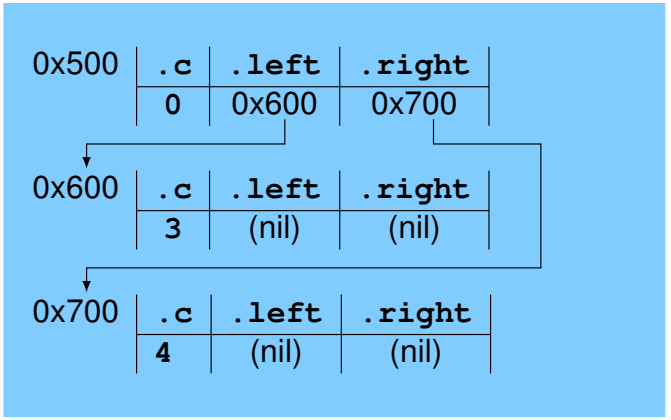
Memory picture

```
L = tree_create_leaf(3); //0x600
R = tree_create_leaf(4); //0x700
```



Memory picture

```
L = tree_create_leaf(3); //0x600
R = tree_create_leaf(4); //0x700
T = tree_join(0, L, R); //0x500
```



Tree traversals

Idea: inspect all nodes once.

Applications: printing, searching for a given information, etc.

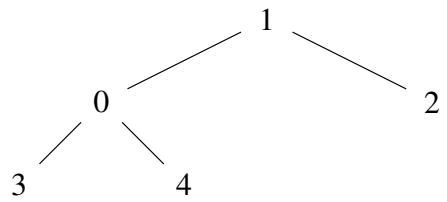
Classical traversals: on (r, T_l, T_r)

- prefix order: traverse r, then T_l, then T_r (closest to the definition);
- infix order: traverse T_l, then r, then T_r (boils down to flattening the tree);
- postfix order: traverse T_l, then T_r, then r.

The choice of the traversal depends on the application.

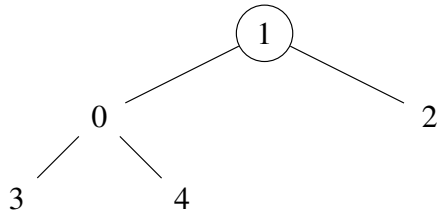
The three traversals for printing

```
/* r, T_l, T_r */
void tree_prefix_print(tree T){
    if(T != NULL){
        printf(" %d", T->c);
        tree_prefix_print(T->left);
        tree_prefix_print(T->right);
    }
}
```



The three traversals for printing

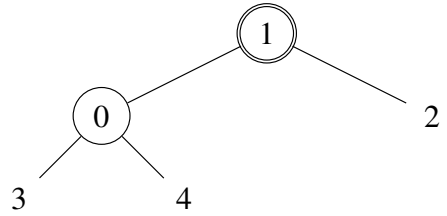
```
/* r, T_l, T_r */
void tree_prefix_print(tree T){
    if(T != NULL){
        printf(" %d", T->c);
        tree_prefix_print(T->left);
        tree_prefix_print(T->right);
    }
}
```



$\Rightarrow 1$

The three traversals for printing

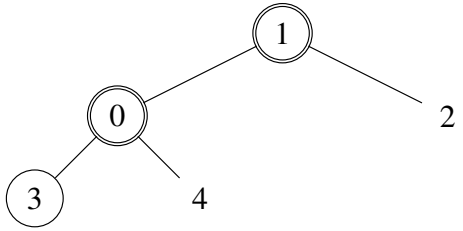
```
/* r, T_l, T_r */
void tree_prefix_print(tree T){
    if(T != NULL){
        printf(" %d", T->c);
        tree_prefix_print(T->left);
        tree_prefix_print(T->right);
    }
}
```



$\Rightarrow 1,0$

The three traversals for printing

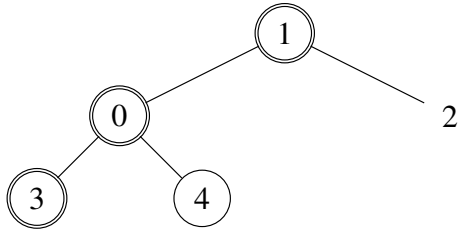
```
/* r, T_l, T_r */
void tree_prefix_print(tree T){
    if(T != NULL){
        printf(" %d", T->c);
        tree_prefix_print(T->left);
        tree_prefix_print(T->right);
    }
}
```



$\Rightarrow 1,0,3$

The three traversals for printing

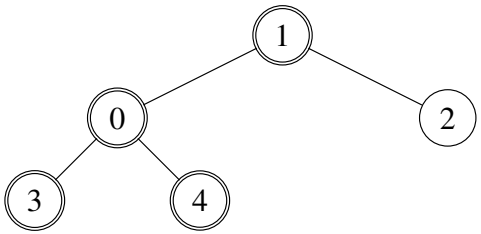
```
/* r, T_l, T_r */
void tree_prefix_print(tree T){
    if(T != NULL){
        printf(" %d", T->c);
        tree_prefix_print(T->left);
        tree_prefix_print(T->right);
    }
}
```



$\Rightarrow 1,0,3,4$

The three traversals for printing

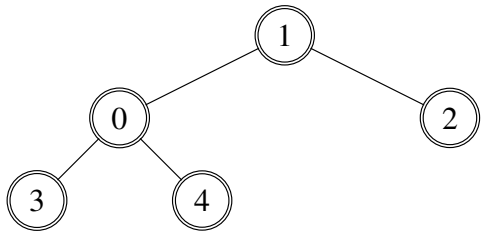
```
/* r, T_l, T_r */
void tree_prefix_print(tree T){
    if(T != NULL){
        printf(" %d", T->c);
        tree_prefix_print(T->left);
        tree_prefix_print(T->right);
    }
}
```



$\Rightarrow 1,0,3,4,2$

The three traversals for printing

```
/* r, T_l, T_r */
void tree_prefix_print(tree T){
    if(T != NULL){
        printf(" %d", T->c);
        tree_prefix_print(T->left);
        tree_prefix_print(T->right);
    }
}
```



$\Rightarrow 1,0,3,4,2$

The other two traversals

```
// L, r, R
void tree_infix_print(tree T){
    if(T != NULL){
        tree_infix_print(T->left);
        printf(" %d", T->c);
        tree_infix_print(T->right);
    }
}
// L, R, r
void tree_postfix_print(tree T){
    if(T != NULL){
        tree_postfix_print(T->left);
        tree_postfix_print(T->right);
        printf(" %d", T->c);
    }
}
```


IV. Hashing

Pb: we want to store a set $\mathscr{E}$ having N integers (say) in a table, so that accessing any x costs as little as possible.

Ex. if $\mathscr{E} = [1,N]$, use `int t[N];` and `t[i] = -1` if empty.

We mimic this by computing for each $x \in \mathscr{E}$ an address $h(x) \in \{0,1,\ldots,M-1\}$ for some well chosen M. If $M \ll N$, several elements might fall at the same address (collision).

Insertion of x: $O(1)$

- compute $h(x)$;
- $\mathscr{T}_{h(x)} := \mathscr{T}_{h(x)} \cup \{x\}$.

Searching x: $O(1)$

- compute $h(x)$;
- return $x \in \mathscr{T}_{h(x)}$?

IV. Hashing

Pb: we want to store a set $\mathscr{E}$ having N integers (say) in a table, so that accessing any x costs as little as possible.

Ex. if $\mathscr{E} = [1,N]$, use `int t[N];` and `t[i] = -1` if empty.

We mimic this by computing for each $x \in \mathscr{E}$ an address $h(x) \in \{0,1,\ldots,M-1\}$ for some well chosen M. If $M \ll N$, several elements might fall at the same address (collision).

Insertion of x: $O(1)$

- compute $h(x)$;
- $\mathscr{T}_{h(x)} := \mathscr{T}_{h(x)} \cup \{x\}$.

Searching x: $O(1)$

- compute $h(x)$;
- return $x \in \mathscr{T}_{h(x)}$?

IV. Hashing

Pb: we want to store a set $\mathscr{E}$ having N integers (say) in a table, so that accessing any x costs as little as possible.

Ex. if $\mathscr{E} = [1,N]$, use `int t[N];` and `t[i] = -1` if empty.

We mimic this by computing for each $x \in \mathscr{E}$ an address $h(x) \in \{0,1,\ldots,M-1\}$ for some well chosen M. If $M \ll N$, several elements might fall at the same address (collision).

Insertion of x: $O(1)$

- compute $h(x)$;
- $\mathscr{T}_{h(x)} := \mathscr{T}_{h(x)} \cup \{x\}$.

Searching x: $O(1)$

- compute $h(x)$;
- return $x \in \mathscr{T}_{h(x)}$?

Example

Hash function: $h(z) = z \bmod M$.

Ex. $\mathscr{E} = \{11,59,32,44,301,26,199\}$; $M = 10$; $\mathscr{T}$ is an array of lists:

i	0	1	2	3	4	5	6	7	8	9
$\mathscr{T}_i$										

Example

Hash function: $h(z) = z \bmod M$.

Ex. $\mathscr{E} = \{11,59,32,44,301,26,199\}$; $M = 10$; $\mathscr{T}$ is an array of lists:

i	0	1	2	3	4	5	6	7	8	9
$\mathscr{T}_i$		11								

$11 = 1 \times 10 + 1 \Rightarrow 11 \bmod 10 = 1$

Example

Hash function: $h(z) = z \bmod M$.

Ex. $\mathscr{E} = \{11, 59, 32, 44, 301, 26, 199\}$; $M = 10$; $\mathscr{T}$ is an array of lists:

i	0	1	2	3	4	5	6	7	8	9
$\mathscr{T}_i$		11								59

$59 = 5 \times 10 + 9 \Rightarrow 59 \bmod 10 = 9$

Example

Hash function: $h(z) = z \bmod M$.

Ex. $\mathscr{E} = \{11, 59, 32, 44, 301, 26, 199\}$; $M = 10$; $\mathscr{T}$ is an array of lists:

i	0	1	2	3	4	5	6	7	8	9
$\mathscr{T}_i$		11	32							59

$32 = 3 \times 10 + 2 \Rightarrow 32 \bmod 10 = 2$

Example

Hash function: $h(z) = z \bmod M$.

Ex. $\mathscr{E} = \{11, 59, 32, 44, 301, 26, 199\}$; $M = 10$; $\mathscr{T}$ is an array of lists:

i	0	1	2	3	4	5	6	7	8	9
$\mathscr{T}_i$		11	32		44					59

$44 = 4 \times 10 + 4 \Rightarrow 44 \bmod 10 = 4$

Example

Hash function: $h(z) = z \bmod M$.

Ex. $\mathscr{E} = \{11, 59, 32, 44, 301, 26, 199\}$; $M = 10$; $\mathscr{T}$ is an array of lists:

i	0	1	2	3	4	5	6	7	8	9
$\mathscr{T}_i$		11, 301	32		44					59

$301 = 30 \times 10 + 1 \Rightarrow 301 \bmod 10 = 1$

collision!

Example

Hash function: $h(z) = z \bmod M$.

Ex. $\mathscr{E} = \{11, 59, 32, 44, 301, 26, 199\}$; $M = 10$; $\mathscr{T}$ is an array of lists:

i	0	1	2	3	4	5	6	7	8	9
$\mathscr{T}_i$		11, 301	32		44		26			59

Example

Hash function: $h(z) = z \bmod M$.

Ex. $\mathscr{E} = \{11, 59, 32, 44, 301, 26, 199\}$; $M = 10$; $\mathscr{T}$ is an array of lists:

i	0	1	2	3	4	5	6	7	8	9
$\mathscr{T}_i$		11, 301	32		44		26			59, 199

collision!

Example

Hash function: $h(z) = z \bmod M$.

Ex. $\mathscr{E} = \{11, 59, 32, 44, 301, 26, 199\}$; $M = 10$; $\mathscr{T}$ is an array of lists:

i	0	1	2	3	4	5	6	7	8	9
$\mathscr{T}_i$		11, 301	32		44		26			59, 199

Rem. We cannot avoid collisions, but if h is well chosen, the lists do not explode.

Rem. We can hash every object we want, since we can map any object to a string then to an integer.

Handling collisions using lists

```
#define HMAX 256
#define HMOD 256

/* fast way of computing x mod 2^n */
#define h(x) ((x) & (HMOD-1))

extern void hash_init(list H[]);
extern void hash_insert(list H[], int n);
extern int hash_is_in(list H[], int n);
extern void hash_print(list H[]);
```

hash.c

```
void hash_init(list H[]){
    int i;

    for(i = 0; i < HMOD; i++)
        H[i] = NULL;
}

void hash_insert(list H[], int n){
    int hn = h(n);

    if(list_is_in(n, H[hn]) == NULL)
        H[hn] = list_add_first(n, H[hn]);
}

int hash_is_in(list H[], int n){
    int hn = h(n);

    return list_is_in(n, H[hn]) != NULL;
}
```


hash.c

```
void hash_init(list H[]){
    int i;

    for(i = 0; i < HMOD; i++)
        H[i] = NULL;
}

void hash_insert(list H[], int n){
    int hn = h(n);

    if(list_is_in(n, H[hn]) == NULL)
        H[hn] = list_add_first(n, H[hn]);
}

int hash_is_in(list H[], int n){
    int hn = h(n);

    return list_is_in(n, H[hn]) != NULL;
}
```


hash.c

```
void hash_init(list H[]){
    int i;

    for(i = 0; i < HMOD; i++)
        H[i] = NULL;
}

void hash_insert(list H[], int n){
    int hn = h(n);

    if(list_is_in(n, H[hn]) == NULL)
        H[hn] = list_add_first(n, H[hn]);
}

int hash_is_in(list H[], int n){
    int hn = h(n);

    return list_is_in(n, H[hn]) != NULL;
}
```


hash.c (cont'd)

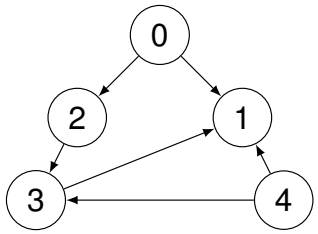
```
void hash_print(list H[]){
    int i;
    for(i = 0; i < HMOD; i++)
        if(H[i] != NULL){
            printf("H[%d]=", i);
            list_print(H[i]);
            printf("\n");
        }
}
```

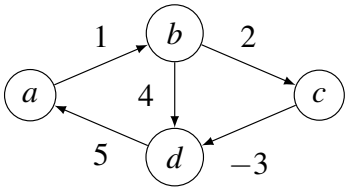
Example

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"
#include "hash.h"

int main(int argc, char *argv[]){
    list tabh[HMAX];
    int i;
    hash_init(tabh);
    for(i = 0; i < 100; i++)
        hash_insert(tabh, abs(random()) % 1000);
    hash_print(tabh);
}
```

```
H[9]= -> 777
H[11]= -> 11
H[12]= -> 12
H[14]= -> 526 -> 782
...
```


V. Graphs





- Modelling networks: metro, train, electricity, INTERNET, etc.
- Representing logical links: ecosystems; dependency graph between source files, heritance diagram, etc.
- Ressource allocation: minimal cost, etc.

Representing graphs

Abstraction: $\mathscr{G} = (\mathscr{V}, \mathscr{E})$.

$\mathscr{V}$**:** set of vertices, e.g., $[0..n-1]$.

$\mathscr{E}$**:** set of edges, e.g., array of lists.

Ex.

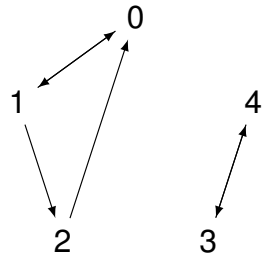
```
L[0] = (1, 2)
L[1] = ()
L[2] = (3)
L[3] = (2, 4)
L[4] = ()
```


When $\mathscr{V}$ is small: adjacency matrix

For a simple graph $\mathscr{G}$, matrix $n \times n$ s.t.

$$M_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \mathscr{E}, \\ 0 & \text{otherwise.} \end{cases}$$

Ex.



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

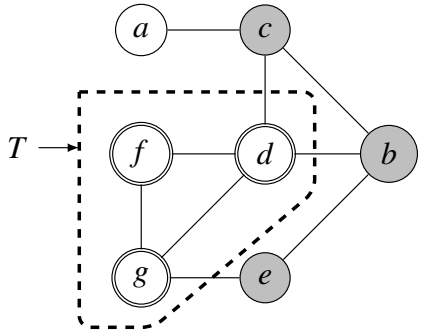
Graph with weights: $M_{i,j} = v(i,j)$.

Comparing both implementations

type	memory	utilisation
matrix	$O(n^2)$	dense graph
lists	$O(\|\mathscr{E}\|+n)$	sparse graph

Traversals (1/2)

Principle: starting from a vertex s, we traverse a subset of $\mathscr{G}$, the neighbours of s. Incrementally, we choose the next vertex in the frontier of the graph.



Def. The frontier $\mathscr{F}(T)$ of $T \subset \mathscr{V}$ is the set of vertices of $\mathscr{V} - T$ that are adjacent to one of the vertices of T.

Traversals (2/2)

Def. A traversal of $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ (connex) starting from s is a list of vertices L s.t.

- the first vertex of L is s;
- each vertex in $\mathscr{V}$ appears only once in L;
- each verext in L (except s) is adjacent in $\mathscr{G}$ to at least a vertex preceding in L.

Rem. There is no canonical (nor best) traversal.

Rem. The two most frequently used traversals are depth search first (DFS), and breadth first search (BFS).

Traversals (2/2)

Def. A traversal of $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ (connex) starting from s is a list of vertices L s.t.

- the first vertex of L is s;
- each vertex in $\mathscr{V}$ appears only once in L;
- each verext in L (except s) is adjacent in $\mathscr{G}$ to at least a vertex preceding in L.

Rem. There is no canonical (nor best) traversal.

Rem. The two most frequently used traversals are depth search first (DFS), and breadth first search (BFS).

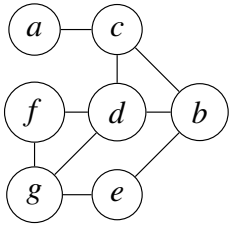
IV. DFS

(*Depth First Search*) the next vertex to be traversed is the most recent vertex of type **beingexplored**.

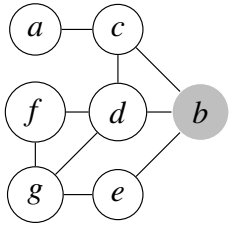
```
function dfs(G, s)
0. forall vertex t do
       state[t] <- unexplored;
1. dfsRec(G, s, state);

function dfsRec(G, s, state)
if state[s] == unexplored then
    state[s] <- beingexplored;
    forall t adjacent to s
        dfsRec(G, t, state);
    state[s] <- explored;
```

Thm. if all list operations cost $O(1)$, then the complexity of DFS is $O(|\mathscr{V}|+|\mathscr{E}|) = O(n+m)$.

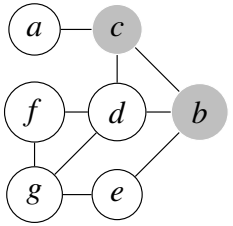
Example



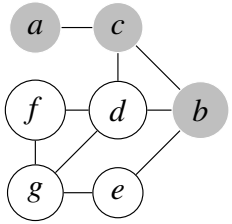
Example



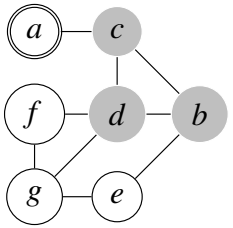
Example



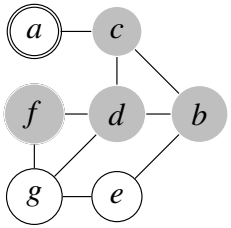
Example



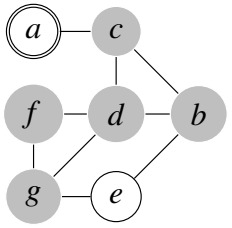
Example



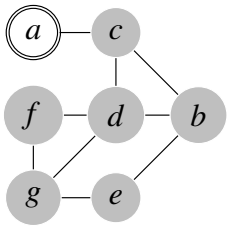
Example



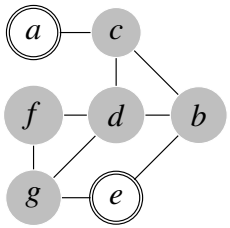
Example



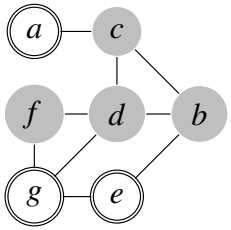
Example



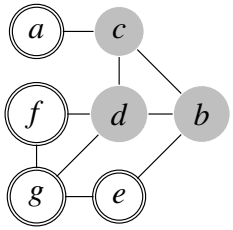
Example



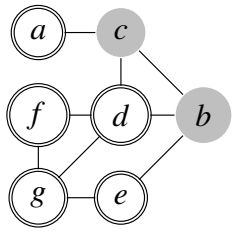
Example



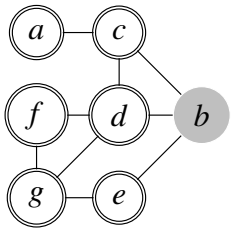
Example



Example



Example



Example