```c
#ifndef __TOOLS_H

#define __TOOLS_H

typedef struct node_t {
  int id;
  struct node_t * fg, *fd;
} node_t;
#define N 256
#define MAX(a, b) ((a) > (b) ? (a) : (b))

extern void push(node_t * v);
extern node_t * pop(void);
extern int empty(void);
extern void enfiler(int v);
extern int defiler(void);
extern int vide(void);

#endif
```

```c
#include "tools.h"

static int haut = 0, file[N + 1], debut = 0, fin = 0;
static node_t * pile[N];

extern void push(node_t * v) {
  pile[haut++] = v;
}
extern node_t * pop(void) {
  return pile[--haut];
}
extern int empty(void) {
  return !haut;
}
extern void enfiler(int v) {
  file[fin++] = v;
  if(fin > N)
fin = 0;
}
extern int defiler(void) {
  int v = file[debut++];
  if(debut > N)
    debut = 0;
  return v;
}
extern int vide(void) {
  return debut == fin;
}
```

FIGURE 1 – Arbres binaires, piles et files (fichiers `tool.h` et `tool.c`)

```c
#include "tools.h"
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

node_t * newNode(int id) {
  node_t * n = malloc(1 * sizeof *n); assert(n);
  n->id = id; n->fg = n->fd = NULL;
  return n;
}
node_t ** findPosition(int id, node_t ** tree) {
  if(*tree == NULL)    return tree;
  if(id < (*tree)->id) return findPosition(id, &((*tree)->fg));
  return findPosition(id, &((*tree)->fd));
}
void printTree(node_t * ptr) {
  if(ptr == NULL) return;
  printTree(ptr->fg); printf("%d ", ptr->id); printTree(ptr->fd);
}
void printTreeP(node_t * ptr) {
  while(ptr != NULL || !empty()) {
    if(ptr) {
      push(ptr);
      ptr = ptr->fg;
      continue;
    } else
      ptr = pop();
    printf("%d ", ptr->id);
    ptr = ptr->fd;
  }
}
void tree2File(node_t * ptr) {
  if(ptr == NULL) return;
  tree2File(ptr->fg); enfiler(ptr->id); tree2File(ptr->fd);
}
void printFile() {
  while(!vide()) printf("%d ", defiler());
  printf("\n");
}
int maxDepth(node_t * ptr, int cd) {
  int a, b;
  if(ptr == NULL) return cd;
  a = maxDepth(ptr->fg, ++cd); b = maxDepth(ptr->fd, cd);
  return MAX(a, b);
}
void freeTree(node_t * ptr) {
  node_t * tmp;
  if(ptr == NULL) return;
  freeTree(ptr->fg);
  tmp = ptr->fd;
  free(ptr); freeTree(tmp);
}
int main(void) {
  node_t * a = NULL;
  int i, n = 100, v;
  for(i = 0; i < n; i++) {
    v = n * (rand() / (RAND_MAX + 1.0));
    *(findPosition(v, &a)) = newNode(v);
  }
  v = maxDepth(a, 0); assert(v <= N);
  printf("Profondeur de l'arbre : %d\n", v);
  /* Imprimer l'arbre récursivement */
  printTree(a); printf("\n");
  /* Imprimer l'arbre en utilisant Notre PILE */
  printTreeP(a); printf("\n");
  /* Utiliser Notre FILE */
  tree2File(a); printFile();
  /* Libérer l'arbre */
  freeTree(a); a = NULL;
  return 0;
}
```

FIGURE 2 – Arbres binaires, piles et files (fichier `ab.c`)

```
1  digraph G {
2          A [shape=rect];
3          A -> B;
4          B -> C;
5          C -> D;
6          D -> A;
7  }
```

FIGURE 3 – Exemple de description de graphe avec GraphViz

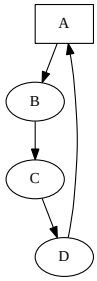


FIGURE 4 – Résultat obtenu pour l'exemple Figure 3

```
1  digraph G {
2          A [shape=box];
3          B [label="Je change de nom"];
4          C [color=red];
5          A -> B;
6          B -> C;
7          C -> D;
8          D -> A [style=dotted];
9  }
```

FIGURE 5 – Autre exemple de description de graphe avec GraphViz

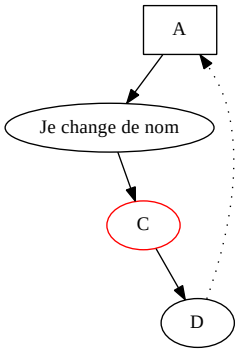


FIGURE 6 – Résultat obtenu pour l'exemple Figure 5

```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
typedef struct node_t node_t;
struct node_t {
  int id;
  struct node_t * fg, *fd;
};
#define MAX(a, b) ((a) > (b) ? (a) : (b))
node_t * newNode(int id) {
  node_t * n = malloc(1 * sizeof *n); assert(n);
  n->id = id; n->fg = n->fd = NULL;
  return n;
}
node_t ** findPosition(int id, node_t ** tree) {
  if(*tree == NULL) return tree;
  if(id < (*tree)->id) return findPosition(id, &((*tree)->fg));
  return findPosition(id, &((*tree)->fd));
}
int maxDepth(node_t * ptr, int cd) {
  int a, b;
  if(ptr == NULL) return cd;
  a = maxDepth(ptr->fg, ++cd);
  b = maxDepth(ptr->fd, cd);
  return MAX(a, b);
}
void freeTree(node_t * ptr) {
  node_t * tmp;
  if(ptr == NULL) return;
  freeTree(ptr->fg);
  tmp = ptr->fd;
  free(ptr); freeTree(tmp);
}
static int myNum = 0;
void tree2GVsub(node_t * ptr, FILE * out, int cd) {
  if(ptr->fg) {
    fprintf(out, "\t\"%p\" -> \"%p\";\n", ptr, ptr->fg);
    tree2GVsub(ptr->fg, out, cd - 1);
  }
  fprintf(out, "\t\"%p\" [label=\"%d\", pos=\"%f,%f!\"];\n", ptr, ptr->id,myNum / 10.0, cd / 10.0);
  myNum++;
  if(ptr->fd) {
    fprintf(out, "\t\"%p\" -> \"%p\";\n", ptr, ptr->fd);
    tree2GVsub(ptr->fd, out, cd - 1);
  }
}
void tree2GV(node_t * ptr, FILE * out) {
  fprintf(out, "digraph G {\n");
  if(ptr) {
    myNum = 1;
    tree2GVsub(ptr, out, maxDepth(ptr, 0));
  }
  fprintf(out, "}\n");
}
int main(void) {
  node_t * a = NULL;
  srand(time(NULL));
  int i, n = 10, v;
  for(i = 0; i < n; i++) {
    v = n * (rand() / (RAND_MAX + 1.0));
    *(findPosition(v, &a)) = newNode(v);
  }
  /* Méta-prog produisant du GraphViz */
  /* faire quelque chose comme:
     gcc -Wall ab.c -o ab && ./ab > out.gv &&
     dot -Kfdp -Tps out.gv -o out.ps && ps2pdf out.ps
     pour obtenir un fichier pdf contenant votre arbre
  */
  tree2GV(a, stdout);
  /* Libérer l'arbre */
  freeTree(a); a = NULL;
  return 0;
}
```

FIGURE 7 – Graphes (ici arbre binaire) et méta-programmation pour l'outil GraphViz.

- Voir l'outil GraphViz ;

- Voir (sur la page web du cours) le support concernant la génération de graphes connexes (labyrinthes) et la résolution de plus courts chemins dans ces graphes.