1 Générécité

– Implémenter une version générique de tri (cf Figure 2).

– Implémenter une version générique de l'arbre binaire vu précédemment en cours (cf Figure 3).

2 Recherche textuelle

– Proposer un algorithme de recherche textuelle ;
Essayons le code Figure 4 avec (motif en 25e non trouvé) :

```
$ gcc -Wall naive.c -o naive
$ ./naive hehe 3134he45hehe0889973234hehehe099134hehe000
Motif trouve en 9ieme position
Motif trouve en 23ieme position
Motif trouve en 35ieme position
```

– L'algorithme KMP : Knuth-Morris-Pratt.
Essayons le code Figure 5 avec :

```
$ gcc -Wall KnuthMorrisPratt.c -o kmp
$ ./kmp ABCABCABCABABABC "ABCABCABABABCABCABCABABABC 23132 2131 ABCABCABCABABABC"
-1 0 0 0 1 2 3 4 5 6 7 8 1 2 1 2
Motif trouve en 11ieme position
Motif trouve en 39ieme position
```

3 Codage de Huffman

Vu en cours :

– Définition du codage statique, semi-adaptatif et adaptatif.

– Le codage semi-adaptatif : création de l'histogramme, de l'arbre et son utilisation en codage / décodage. Un exemple de text encodé et de l'arbre utilisé est donné Figure 1).

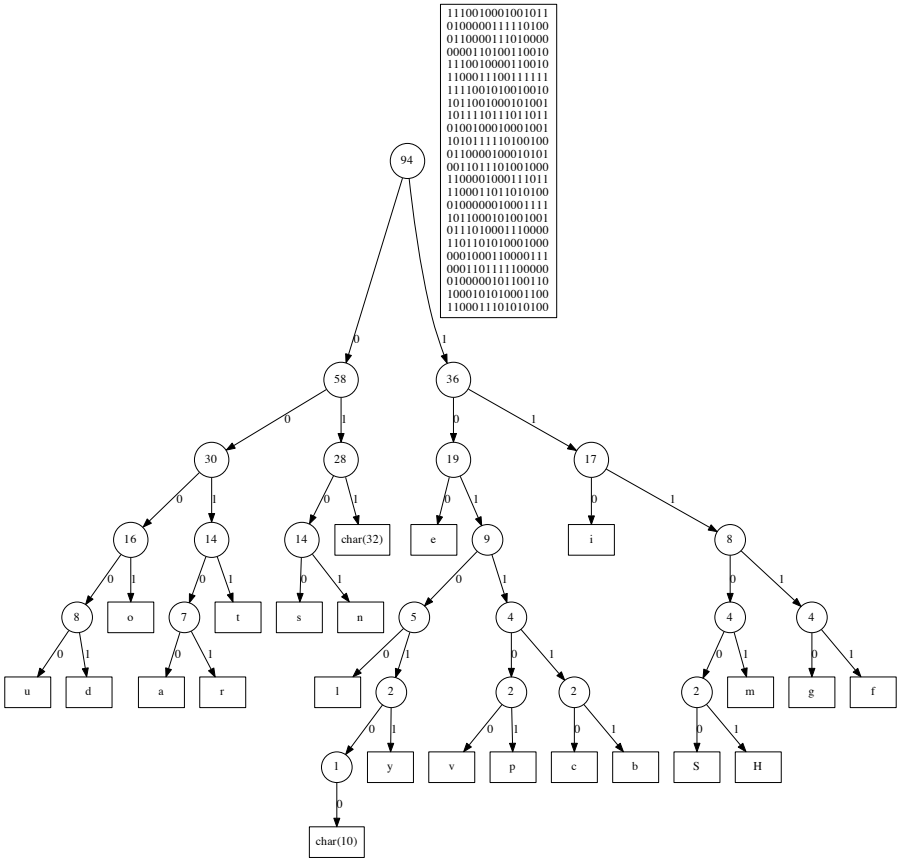

FIGURE 1 – Text encodé Huffman

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <string.h>
5  typedef struct point point;
6  struct point {
7    float x;
8    long long t;
9  };
10
11 void triInsertion(void * t, int n, size_t s, int (*compfunc)(const void *, const void *)) {
12   int i, j;
13   unsigned char * t2 = t;
14   unsigned char * v = malloc(s); assert(v);
15   for(i = 1; i < n; i++) {
16     memcpy(v, &t2[(j = i) * s], s);
17     while(j > 0 && compfunc(&t2[(j - 1) * s], v) > 0) {
18       memcpy(&t2[j * s], &t2[(j - 1) * s], s);
19       j--;
20     }
21     memcpy(&t2[j * s], v, s);
22   }
23   free(v);
24 }
25
26 int compPoint(const void * ptr1, const void * ptr2) {
27   point * p1 = (point *)ptr1;
28   point * p2 = (point *)ptr2;
29   return p1->x > p2->x ? 1 : (p1->x < p2->x ? -1 : 0);
30 }
31
32 int main(void) {
33   point p[10];
34   int i;
35   for(i = 0; i < 10; i++) {
36     p[i].x = rand() / (RAND_MAX + 1.0);
37   }
38   for(i = 0; i < 10; i++)
39     printf("%f ", p[i].x);
40   printf("\n");
41   triInsertion(p, 10, sizeof *p, compPoint);
42   //ici le quicksort de la lib standard :
43   //qsort(p, 10, sizeof *p, compPoint);
44   for(i = 0; i < 10; i++)
45     printf("%f ", p[i].x);
46   printf("\n");
47   return 0;
48 }
49
```

FIGURE 2 – Généricité du tri insertion (recopie de données contigues)

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

typedef struct node_t {
  void * data;
  struct node_t * fg, *fd;
} node_t;

#define N 256
#define MAX(a, b) ((a) > (b) ? (a) : (b))

node_t * newNode(void * data) {
  node_t * n = malloc(1 * sizeof *n); assert(n);
  n->data = data; n->fg = n->fd = NULL;
  return n;
}
node_t ** findPosition(void * data, node_t ** tree, int (*compar)(const void *, const void *)) {
  if(*tree == NULL)    return tree;
  if(compar(data, (*tree)->data) < 0) return findPosition(data, &((*tree)->fg), compar);
  return findPosition(data, &((*tree)->fd), compar);
}
void printTree(node_t * ptr, void (*printfunc)(const void * data)) {
  if(ptr == NULL) return;
  printTree(ptr->fg, printfunc); printfunc(ptr->data); printTree(ptr->fd, printfunc);
}
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->data);
  free(ptr);
  freeTree(tmp);
}
int comp(const void * ptr1, const void * ptr2) {
  return *((float *)ptr1) < *((float *)ptr2) ? -1 : 1;
}
void print(const void * data) {
  printf("%f ", *((float *)data));
}
int main(void) {
  node_t * a = NULL;
  int i, n = 100, p;
  float * v;
  for(i = 0; i < n; i++) {
    v = malloc(sizeof *v); assert(v);
    *v = (float)n * (rand() / (RAND_MAX + 1.0));
    *(findPosition(v, &a, comp)) = newNode(v);
  }
  p = maxDepth(a, 0); assert(p <= N);
  printf("Profondeur de l'arbre : %d\n", p);
  /* Imprimer l'arbre récursivement */
  printTree(a, print); printf("\n");
  /* Libérer l'arbre */
  freeTree(a); a = NULL;
  return 0;
}
```

FIGURE 3 – Généricité des arbres binaires

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

int rechercheNaive(char * m, char * t) {
  int i, j, lm, lt;
  lm = strlen(m); lt = strlen(t);
  for(i = j = 0; j < lm && i < lt; i++, j++)
    while(t[i] != m[j]) { i -= j - 1; j = 0; }
  if(j == lm) return i - lm;
  return i;
}

int main(int argc, char ** argv) {
  int p = 0;
  if(argc != 3) {
    fprintf(stderr, "usage : %s MOTIF TEXTE\n", argv[0]);
    exit(1);
  }
  while( (p += rechercheNaive(argv[1], &argv[2][p])) < strlen(argv[2]) ) {
    printf("Motif trouve en %dieme position\n", p + 1);
    p += strlen(argv[1]);
  }
  return 0;
}

```

FIGURE 4 – Recherche naïve

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

int * initialiserAutomate(char * m) {
  int i, j, lm = strlen(m);
  int * back = malloc(lm * sizeof back[0]);
  assert(back);
  for(i = 0, j = -1; i < lm; i++, j++) {
    back[i] = j;
    while( j >= 0 && m[i] != m[j] ) j = back[j];
  }
  for(i = 0; i < lm; i++)
    printf("%d ", back[i]);
  printf("\n");
  return back;
}

int rechercheKMP(int * back, char * m, char * t) {
  int i, j, lm, lt;
  lm = strlen(m); lt = strlen(t);
  for(i = j = 0; j < lm && i < lt; i++, j++)
    while( j >= 0 && t[i] != m[j] ) { j = back[j]; }
  if(j == lm) return i - lm;
  return i;
}

int main(int argc, char ** argv) {
  int p = 0, * back;
  if(argc != 3) {
    fprintf(stderr, "usage : %s MOTIF TEXTE\n", argv[0]);
    exit(1);
  }
  back = initialiserAutomate(argv[1]);
  while( (p += rechercheKMP(back, argv[1], &argv[2][p])) < strlen(argv[2]) ) {
    printf("Motif trouve en %dieme position\n", p + 1);
    p += strlen(argv[1]);
  }
  free(back);
  return 0;
}

```

FIGURE 5 – Algorithme Knuth-Morris-Pratt