Heap

1 Structs

```c
typedef struct {
  int chave;
  int dados;
  int prox;
} ttabpos;

struct smapa {
  int tam;
  int ocupadas;
  ttabpos *tabpos;
};
```

2 Headers

```c
typedef struct smapa Mapa;

Mapa* cria (void);
Mapa* insere (Mapa* m, int chave, int novodado);
Mapa* retira (Mapa *m, int chave);
void destroi (Mapa *m);

int busca (Mapa *m, int chave);
int compare (Mapa * m1, Mapa * m2);
int maior_cadeia(Mapa * m);
void mostra (Mapa *m);
```

3 Implementation

3.1 Core Functions

3.1.1 Hash Function

```c
static unsigned int hash1 (Mapa* m, int a) {
  return a%(m->tam);
}
static unsigned int hash (Mapa* m, int chave) {
  return hash1(m, chave);
}
```

3.1.2 Hash Cria

```c
Mapa* cria () {
  int i;
  Mapa* m = (Mapa*) malloc (sizeof(Mapa));
  if (m==NULL) {printf("erro na   alocao  ! \n"); exit(1);}
  m->tabpos = (ttabpos*) malloc (TAMINICIAL*sizeof(ttabpos));
  if (m->tabpos==NULL) {printf("erro na   alocao  ! \n"); exit(1);}
  m->tam = TAMINICIAL;
  m->ocupadas = 0;
  for (i=0;i<TAMINICIAL;i++) {
```

```
    m->tabpos[i].chave = -1;
    m->tabpos[i].prox = -1;
  }
  return m;
}
```

3.1.3 Insere

```
Mapa* insere (Mapa* m, int chave, int dados) {
  if (m->ocupadas > 0.75*m->tam) redimensiona(m);

  int pos = hash(m, chave);

  if (m->tabpos[pos].chave == -1) { /* est vazia */
    m->tabpos[pos].chave = chave;
    m->tabpos[pos].dados = dados;
    m->tabpos[pos].prox = -1;
  }
  else { /* conflito */
    /* procura proxima   posio   livre */
    int poslivre = pos;
    do
      poslivre = (poslivre+1) % (m->tam);
    while ((poslivre!=pos) && (m->tabpos[poslivre].chave!=-1));

    if (poslivre==pos) { /* tabela cheia -- no deveria         acontecer */
      printf (" pnico , tabela cheia!\n"); exit(1);
    }

    /* achou posicao livre - verificar quem vai para ela */
    int hashocupadora = hash(m, m->tabpos[pos].chave);
    if (hashocupadora==pos) { /* conflito primario: encadeia */
      m->tabpos[poslivre].chave = chave;
      m->tabpos[poslivre].dados = dados;
      m->tabpos[poslivre].prox = m->tabpos[pos].prox;
      m->tabpos[pos].prox = poslivre;

    }
    else { /* conflito secundario: expulsa o item atual de pos */

      m->tabpos[poslivre].chave = m->tabpos[pos].chave;
      m->tabpos[poslivre].dados = m->tabpos[pos].dados;
      m->tabpos[poslivre].prox = m->tabpos[pos].prox;

      while(m->tabpos[hashocupadora].prox != pos)
        hashocupadora = m->tabpos[hashocupadora].prox;
      m->tabpos[hashocupadora].prox = poslivre;

      m->tabpos[pos].chave = chave;
      m->tabpos[pos].dados = dados;
      m->tabpos[pos].prox = -1;


    }
  }
  (m->ocupadas)++; /* aumentou o nmero de itens na tabela */
  return m;
}
```

3.1.4 Redimensiona

```
static void redimensiona (Mapa* m) {
  int i;
  int tamanterior = m->tam;
  ttabpos* anterior = m->tabpos;
  printf ("redimensiona...\n");
  m->tam = 1.947*m->tam;
  printf("novo tamanho: %d\n", m->tam);
  m->tabpos = (ttabpos*) malloc (m->tam*sizeof(ttabpos));
  m->ocupadas = 0;
  for (i=0;i < m->tam;i++) {
    m->tabpos[i].chave = -1;
    m->tabpos[i].prox = -1;
  }
  for (i=0; i<tamanterior; i++)
    if (anterior[i].chave != -1)
      insere (m, anterior[i].chave, anterior[i].dados);
  free (anterior);
}
```

3.1.5 Removing

```
Mapa * retira(Mapa *m, int chave)
{
  if (m==NULL) return NULL;
  (m->ocupadas)--;
  int pos = hash(m, chave);
  int anterior = -1;
  while(m->tabpos[pos].chave != chave)
    {
      anterior = pos;
      pos = m->tabpos[pos].prox;
      if( pos == -1)
        {
          printf("Return map\n");
          return m;
        }
    }
  int prox = m->tabpos[pos].prox;
  if (prox != -1) {
    m->tabpos[pos].chave = m->tabpos[prox].chave;
    m->tabpos[pos].dados = m->tabpos[prox].dados;
    m->tabpos[pos].prox = m->tabpos[prox].prox;
    m->tabpos[prox].chave = -1;
    m->tabpos[prox].dados = -1;
    m->tabpos[prox].prox = -1;
    return m;
  }
  else
  {
    m->tabpos[pos].chave = -1;
    m->tabpos[pos].dados = -1;
    m->tabpos[pos].prox = -1;
  }

  if(anterior != -1){
     m->tabpos[anterior].prox = m->tabpos[pos].prox;
     return m;
  }
  else
```

```
  {
    m->tabpos[pos].chave = -1;
    m->tabpos[pos].dados = -1;
    m->tabpos[pos].prox = -1;
  }


  return m;

}
```

3.2 Auxiliary Methods

3.2.1 Busca

```
int busca (Mapa *m, int chave) {
  if (m==NULL) return -1;
  int pos = hash(m, chave);
  while(m->tabpos[pos].chave != chave)
    {
      pos = m->tabpos[pos].prox;
      if(m->tabpos[pos].chave == chave)
        return m->tabpos[pos].dados;
      else if( pos == -1)
        return -1;
    }
  return m->tabpos[pos].dados;
}
```

3.2.2 Mostra

```
void mostra (Mapa* m) {
  int i;
  for (i=0;i<m->tam;i++)
    if (m->tabpos[i].chave!=-1)
      printf ("posicao %d, chave %d, proximo %d\n", i, m->tabpos[i].chave, m->tabpos[i].prox);
}
```

3.2.3 Compare

```
int iguais (Mapa* m1, Mapa* m2) {
  int i;

  if (m1==NULL || m2==NULL) return (m1==NULL && m2==NULL);
  if (m1->tam != m2->tam) return 0;
  ttabpos* tp1 = m1->tabpos;
  ttabpos* tp2 = m2->tabpos;
  for (i = 0; i < m1->tam; i++)
    if ((tp1[i].chave != tp2[i].chave) ||
        (tp1[i].prox != tp2[i].prox))
      return 0;
  return 1;
}
```

3.2.4 Tamanho Maior Cadeia

```
int maior_cadeia(Mapa *m) {
  int max = 0;
```

```
  for(int i = 0; i < m->tam; i++){
    int pos = m->tabpos[i].chave;
    if( hash(m,i) == pos){
      int cadeia = 1;
      while(m->tabpos[pos].prox != -1){
        cadeia++;
        pos = m->tabpos[pos].prox;
      }
      if(cadeia > max) max = cadeia;
    }
  }

  return max;
}
```
