Prova 2

Data Criado	@August 11, 2022 5:37 PM
Tipo	
Finalizado	☑️
Fazer em	

Tabela Hash

Introdução e Implementação Inicial

- **Vantagens:** inserção e busca em tempo constante (O(1)). → Quanto maior o tamanho da tabela hash, maior o número/tempo de operações.
- A ordenação acontece para que a busca (binária → lgn) seja eficiente e custe pouco, muito bom quando são muitas buscas.
- Ideia: A tabela Hash armazena **chave** (index) e **valor** (item).
  - As chaves não necessariamente estão ordenadas.
  - As chaves são codificadas em um índice do vetor.
- Normalmente, usa-se o tamanho da tabela hash como o número primo mais próximo ao valor que quer se armazenar.
- Função para criar o Hash de Número Inteiro:

```
int hashN (int N){
  return n % valor_da_tabela; // Valor da tabela sendo o primo mais próximo
}
// Mesma coisa, caso não seja primo, 618033*v % M
#define hash(v, M) (v%M)

int main(void){
  v[hashN[elem]] = elem; // Colocar o valor na sua chave.
}
```

- Não há como garantir não colisão, uma vez que sempre há um valor limitado de valores e alguns não aparecerão.

- Função para criar o Hash Universal para String (pode ficar muito caro):

```
int hashU (char *v, int M){
  int h, a = 31415, b = 27183;
  for(h = 0; *v != '\0'; v++){ // percorrer toda a string
    a = a*b % (M-1);
    h = (a*h + *v) % M; // *v = ASCII do elemento -- Sempre vai entrar no tamanho da tabela
  }
  return h;
}
```

- Funções essenciais para tabela Hash:

```
typedef struct Item{
  int chave;
  long elemento;
} Item;

#define key(x) (x.chave)
#define null(A) (key(ht[A]) == NULL)

static int N, M;
static Item *ht;

void htInit(int max){
  int i;
  N = 0;
  M = 2*max; // pode selecionar primo
  ht = malloc(sizeof(Item)*M);
  for(i = 0; i < M; i++)
    ht[i] = NULL;
}

int htInsert(Item item){
  int v = key(Item);
  int i = hash(v, M);
  int try = 0;
  while(!null(ht[i])){ // Enquanto não encontrar um vazio
    if(try == MAXTRY) return 0;
    i = (i+1)%M; // Passa i para o próximo elemento
    try++;
  }

  ht[i] = item; // ht[i] = malloc(tam); strcpy(ht[i], item); -> se for string
  N++;
  return 1;
}

void htSearch(int v){
  int i = hash(v, M);
  int try = 0;
```

```
  while(!null(i)){ // Se a tabela tá cheia, entra em loop infinito
    if(try == MAXTRY) return NULL;
    if(eq(v, key(ht[i]))
      return st[i]; // Cópia do item
    else
      i = (i+1) % M;
    try++;
  }
  return NULL; // Não encontrou
}

int verifica_colisao(item M){
    int k = hash(M), prox_pos;

    for(int i = 0; i < 20; i++){
        prox_pos = (k+i*i+23LL*i) % 101;
        if(ht[prox_pos] != NULL && strcmp(ht[prox_pos], palavra) == 0){
            return 0; // adicao nao pode ser feita - colisao
        }
    }
    return 1;
}

int htRemove(int v){
  int k = hash(palavra), prox_pos;
  for (int l = 0; l < 20; l++){
    prox_pos = (k + l*l + 23LL*l) % 101;

    if(ht[prox_pos] != NULL && strcmp(ht[prox_pos], palavra) == 0){
        free(ht[prox_pos]);
        ht[prox_pos] = NULL;
        tam--;
        break;
    }
  }
}
```

Estratégias para Colisão

- **Encadeamento Separado**: cada indíce do vetor guarda uma lista encadeada. Logo, toda vez que acontecer uma colisão, um novo elemento entra na lista.
- **Endereçamento aberto**: insere no próximo indíce vazio. Pode acontecer muitas colisões se a tabela Hash tiver tamanho próximo à quantidade de valores que serão inseridos. Para resolver, deve-se crescer a tabela e recalcular tudo de novo.

Double Hashing

- Ao invés de fazer o endereçamento aberto pulando apenas para o próximo item, calcula outra hash para o item (um salto diferente).

```
#define hash(v, M) (v % M)
#define hashtwo(v, M) (v % 97)

void htInsert(Item item){
  keytype v = key(item);
  int i = hash(v, M);
  int k = hashtwo(v, M);
  while(!null(i))
    i = (i + k) % M; // Rodar com pulos diferentes -> Contador de tentativas
  ht[i] = item;
  N++;
  return;
}

Item htSearch(key v){
  int i = hash(v, M);
  int k = hashtwo(v, M);
  while(!null(i))
    if(eq(v, key(ht[i])
      return ht[i];
    else
        i = (i+k) % M; // Aplica salto até encontrar
  return NULL;
}
```

- **Hash 1 está com problema:** Colisão em todos os elementos (custo de inserção continua o mesmo, mas a busca fica cara).
- **Hash 2 está com problema:** Colisão em alguns elementos na hash 1 e colisão total na segunda hash.
- **Hash 1 e 2 erradas**: Inserção linear e busca linear.

Tabela Hash Dinâmica

- Em alguns casos, é necessário expandir a tabela hash para alocar mais elementos.
- Tem que recalcular a hash de cada um dos elementos, o que pode acabar fazendo muitos cálculos caros.

```
void htInsert(Item item){
  keytype v = key(item);
  int i = hash(v, M);
  int k = hashtwo(v, M);
  while(!null(i))
    i = (i + k) % M; // Rodar com pulos diferentes -> Contador de tentativas
  ht[i] = item;
  N++;
  if(N >= M/2) expand();
```

```
  return;
}

void expand(){
  // Item *t = ht;
  // Malloc de um ht com tamanho M *2
  // For de i = 0 até i < m/2
    // if(key(t[i]) != NULL)
      // insertNovaHash(t[i]);
}
```

Usos práticos de Hash

- Caso queira salvar com os índices do vetor:

```
key(A) (A.count)
// caso seja igual o contador, escolhe pela chave
#define less(A, B) (Key(A) == Key(B) ? A.chave < B.chave : Key(A) < Key(B))

memset(v, 0, sizeof(Item)*N); // iniciar com 0

th[i].chave = i;
th[i].count++; // depois um sort para saber o maior valor
```

- Hash com lista encadeada:

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

//HT = HashTable = Tabela Hash
// para um vetor o intervalo fechado é [0,262143]
#define HTSIZE 140000
#define HTNULL -1

typedef struct no
{
  int Pi;
  struct no *prox;
} no;

typedef struct lista_st
{
  no *head;
  int count;
} lista_st;

void LEinit(lista_st *lista)
{
  lista->head=NULL;
```

```
  lista->count=0;
}

void LEinsert(lista_st *lista, int Pi)
{
  no *l=malloc(sizeof(no));
  l->Pi=Pi;
  l->prox=lista->head;
  lista->head=l;
  lista->count++;
}

int LEsearch(lista_st *lista,int x)
{
  no *aux=lista->head;
  while(aux!=NULL)
  {
    if(aux->Pi==x)
      return 1;
    aux=aux->prox;
  }
  return 0;

}

typedef struct HT_st
{
  lista_st *ht;
  int count;
} HT_st;

int hash(int Pi)
{
  return Pi%HTSIZE;
}

void HTinit(HT_st *HT)
{
  HT->ht=malloc(sizeof(lista_st)*HTSIZE);
  HT->count=0;

  //elemento vazio da tabela hash será o -1
  for(int i=0;i<HTSIZE;i++)
    LEinit(&HT->ht[i]);
}

void HTinsert(HT_st *HT,int x)
{
  int hashv=hash(x);
  LEinsert(&HT->ht[hashv],x);
  HT->count++;
}

int HTsearch(HT_st *HT, int x)
{
  int hashv=hash(x);
```

```
  return LEsearch(&HT->ht[hashv],x);

}

int main(void)
{
  HT_st hashtable;
  HTinit(&hashtable);

  //Os números proibidos variam entre [0,2^31]
  int N;
  scanf("%d",&N);
  for(int i=0;i<N;i++)
  {
    int Pi;
    scanf("%d",&Pi);
    HTinsert(&hashtable,Pi);
  }
  #if 0
  for(int i=0;i<HTSIZE;i++)
    if(hashtable.ht[i].count>=2)
      printf("Colisao em %d, temos %d elementos\n",i,hashtable.ht[i].count);
  #endif

  int pergunta;
  while(scanf("%d",&pergunta)==1)
  {
    if(HTsearch(&hashtable,pergunta))
      printf("sim\n");
    else
      printf("nao\n");
  }
}
```

Filas de Prioridade (Priority Queue)

Introdução

- Não é de busca, é uma fila.
- Manipulam conjuntos de itens, em que um item K é:
  - **Máximo** se **nenhum** item é estritatemente **maior** que K.
  - **Mínimo** se **nenhum** item é estritamente **menor** que K.
    - Podem ter **mais** de um item máximo e mais de um item mínimo.

- O uso de **listas encadeadas** possui inserção (ou remoção por busca) ordenada cara O(n).
- O uso de **vetores** tem o mesmo caso que listas encadeadas.
  - Inserção, remoção do maior, remoção e mudar a prioridade **lg(n)**. Encontrar o maior é constante.
- Não usada para buscas além do maior e menor.

Heap

- Tipo de árvore binária para implementar fila de prioridade eficiente. Dois tipos:
  - **Decrescente**: o **máximo** fica no **topo**. Nó sempre é maior ou igual ao pai.
  - **Crescente**: o **mínimo** fica no **topo**. Nó sempre é menor ou igual ao pai.
- Em vetor:
  - Não usa o índice 0.
  - Nós da esquerda: 2*K (índice).
  - Nós da direita: 2*K + 1 (índice).
  - Pai sempre é o chão de K/2.

Aplicação:

- Consertar de baixo para cima (fixUp → filho é maior que pai) e de cima para baixo (fixDown → pai é menor que filho):

```
void fixUp(item *v, int k){ // índice estragado
  while(k > 1 && less(v[k/2], v[k]){ // Verifica se o filho é maior que o pai
    exch(v[k], v[k/2]); // Se sim, troca
    k = k/2; // Começa a analisar o pai
  }
}

void fixDown(item *v, int k, int N){ // índice estragado e tamanho da heap com intervalo fechado
  int j;
  while(2*k <= N){ // enquanto não acabar a Heap
    j=2*k; // vê o filho
    if(j < N && less(v[j], v[j+1])) j++;
    if(!less(v[k], v[j])) break;
    exch(v[k], v[j]);
    k = j;
```

```
  }
}
```

- Funções essenciais:

```
// Podia ser uma struct também
static Item *pq;
static int N;

void PQinit(int maxN){
  pq = malloc(sizeof(Item)*(maxN+1));
  N = 0;
}

int PQempty(){
  return N == 0;
}

void PQinsert(Item novo){
  pq[++N] = novo;
  fixUp(pq, N); // Adiciona no final e arruma
}

Item PQdelMax(){
  exch(pq[1], pq[N]);
  fixDown(pq, 1, --N);
  return pq[N+1]; // Joga o maior para o final, arruma e retorna ele
}

PQchange{
  fixUp;
  fixDown;
}
```

Auxiliar (Merge Sort)

```
void merge(Item *vetor, int l, int m, int r){
    int l1 = l, r1 = m, l1i = l1;
    int l2 = m+1, r2 = r, l2i = l2;

    Item *c = malloc((r-l+1)*sizeof(Item));
    int ci = 0;

    int rc = r-l;

    while (l1i <= r1 && l2i <= r2){
        if(less(vetor[l1i], vetor[l2i])){
            c[ci++] = vetor[l1i++];
```

```
        } else{
            c[ci++] = vetor[l2i++];
        }
    }

    while (l1i <= r1){
        c[ci++] = vetor[l1i++];
    }
    while (l2i <= r2){
        c[ci++] = vetor[l2i++];
    }

    l1i = l;
    for(ci=0; ci <= rc; ci++){
        //printf("%c - %d\n", vetor[ci].caracter, vetor[ci].count);
        vetor[l1i++] = c[ci];
    }

    free(c);
}

void mergesort(Item *vetor, int l, int r){
    if (l >= r) return;
    int meio = (r - l)/2 + l;
    mergesort(vetor, l, meio);
    mergesort(vetor, meio+1, r);
    merge(vetor, l, meio, r); // junta os dois vetores
}
```