Estrutura de Dados e Algoritmos 2: Lista 02

Maurício Serrano

Paulo Tada 11/0135431 Maria Luciene 12/0037742

Exercício 1

O algoritmo a seguir realiza a ordenação de uma estrutura de lista duplamente encadeada. Na troca realizada pela função swap, toda a estrutura do nó é modificada.

Listing 1: Algoritmo de Quick Sort para listas encadeada

```cpp
#include <iostream>
#include <vector>
#include <stdlib.h>
#include "double_list.h"
#include "vector_preconfig.h"

void create_list(List* list){
  insert(list,4);
  insert(list,6);
  insert(list,1);
  insert(list,8);
  insert(list,4);
  insert(list,2);
  insert(list,3);
}

void swap(List* list, int fk, int sk){
  Node* first = search(list,fk);
  Node* second = search(list,sk);

  int key = first->key;
  int data = first->data;
  Node* prev = first->prev;
  Node* next = first->next;

  first->key = second->key;
  first->data = second->data;
  first->prev = second->prev;
  first->next = second->next;

  second->key = key;
  second->data = data;
  second->prev = prev;
  second->next = next;

  // Exchange
  second->key = first->key;
  first->key = key;
  second->prev = first->prev;
  second->next = first->next;
  first->prev = prev;
  first->next = next;
}

void quick_sort(List* list, int left, int right){
  Node* pivo_node = search(list, left);
  int i = left;
```

```
  int j = right;

  int pivo = pivo_node->data;

  while(i<=j){
    while(search(list,i)->data < pivo && i < right){
      i++;
    }
    while(search(list,j)->data > pivo && j > left){
      j--;
    }
    if(i<=j){
      swap(list, i, j);
      i++;
      j--;
    }
  }
  if(j > left) quick_sort(list, left, j);
  if(i < right) quick_sort(list, i, right);
}

int main(){
  List* list = NULL;
  list = init();
  create_list(list);

  print_list(list);
  cout << endl;
  quick_sort(list,0,(list->num_nodes)-1);
  print_list(list);
  cout << endl;
  return 0;

}
```

A biblioteca mostrada abaixo contem as funções necessárias para realizar a estrutura de lista duplamente encadeada.

Listing 2: Biblioteca de lista duplamente encadeada

```
#include<stdio.h>
using namespace std;

typedef struct node Node;
typedef struct list List;

List* init();
void print_list();

struct node{
  int key;
  int data;
  Node* next;
  Node* prev;
```

```
};

struct list{
  Node* first;
  Node* last;
  int num_nodes;
};

List* init(){
  List* new_list = (List*) malloc (sizeof(List));
  new_list->first = NULL;
  new_list->last = NULL;
  new_list->num_nodes = 0;
  return new_list;
}

void print_node(Node* node){
  cout << "-----Node-----" << endl;
  if(node){
    cout << "key: " << node->key << endl;
    cout << "data: " << node->data << endl;
    if(node->next){
      cout << "next: " << node->next->key << endl;
      cout << "next->prev:" << node->next->prev->key << endl;
    }else{
      cout << "No NEXT node!" << endl;
    }
    if(node->prev){
      cout << "prev>" << node->prev->key << endl;
      cout << "prev->next:" << node->prev->next->key << endl;
    }else{
      cout << "No PREV node!" << endl;
    }
  }else{
    cout << "NULL Node!" << endl;
  }
  cout << "-----/node-----" << endl;
}

void print_list_reverse(List* list){
  Node* print = list->last;
  while(print != NULL){
    cout << print->data << " ";
    //cout << "(" << print->key << ")"<< print->data << " ";
    print = print->prev;
  }
}

void print_list(List* list){
  Node* print = list->first;
  while(print != NULL){
    cout << print->data << " ";
    //cout << "(" << print->key << ")"<< print->data << " ";
```

```c
    print = print->next;
  }
}

void insert(List* list, int element){
  Node* new_node = (Node*) malloc (sizeof(Node));
  new_node->data = element;
  new_node->next = NULL;

  if(list->num_nodes == 0){
    new_node->prev = NULL;
    new_node->key = 0;
    list->first = new_node;
    list->last = new_node;
  }else{
    new_node->prev = list->last;
    new_node->key = new_node->prev->key + 1;
    list->last->next = new_node;
    list->last = new_node;
  }
  list->num_nodes++;
}

Node* search(List* list, int key){
  Node* result = NULL;
  Node* search = list->first;
  while(search != NULL){
    if(search->key == key){
      result = search;
      break;
    }
    search = search->next;
  }
  return result;
}
```

Exercício 2

A estrutura apresentada abaixo é um algoritmo de ordenação heap utilizando apenas um único vetor para realizar a ordenação.

Listing 3: Heap utilizando dois vetores

```
#include <iostream>
#include <cmath>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

using namespace std;

int get_max(int array[], int left, int right, int root);
int extract_max(int array[], int size);
void build_heap(int array[], int array_size);
void print_heap(int array[], int array_size);
void insert_heap(int array[], int element, int size);
void heapify_up(int array[], int size);
void heapify(int array[], int i, int array_size);
void heap_sort(int array[], int numbers[], int array_size);

int main(){
     int array_size = 8;
     int array[array_size];
    int numbers[] = {10, 35, 6, 8, 3, 90, 1, 78};

     print_heap(numbers, array_size);

     // Inicializa vetor com zeros
     memset(array, 0, sizeof(array));

     heap_sort(array, numbers, array_size);

     return 0;
}

void heap_sort(int array[], int numbers[], int array_size){
  int sort[array_size];
  memset(sort, 0, sizeof(sort));

  for(int i = 0; i < array_size ; i++){
    insert_heap(array,numbers[i],i);
  }

  for(int i = array_size-1; i >= 0 ; i--){
    sort[i] = extract_max(array, i);
  }

  print_heap(sort, array_size);

}
```

```
int extract_max(int array[], int size){
   int maximo;
   if(size > 1){
      maximo = array[1];
      array[1] = array[size+1];
      heapify(array, 1, size+1);
   } else if(size == 1){
      maximo = array[1];
      size = 0;
   } else {
      maximo = -1;
   }
   return maximo;
}


// Coloar todos os filhos maiores na raiz
void heapify(int array[], int i, int array_size) {
     int left = 2*i;
     int right = 2*i + 1;
     int max_child = 0;

     if(left < array_size && right < array_size){

          max_child = get_max(array, left, right, i);

          if(max_child != i){
               swap(array[i], array[max_child]);
               heapify(array, max_child, array_size);
          }
     }
}

void insert_heap(int array[], int element, int size){
     size++;   // Primeiro elemento comeca na posicao 1
     array[size] = element;
     heapify_up(array, size);
}
void heapify_up(int array[], int index) {
     while(index>1){
          if(array[index]>array[index/2]){
               swap(array[index],array[index/2]);
          }
          index/=2;
     }
}




void print_heap(int array[], int array_size) {
     for (int i = 1; i < array_size; ++i)
```

```
    {
        printf("%d ", array[i]);
        if (array[i] != array[array_size-1])
            cout << ",";
    }
    cout << endl;
}


// Retorna o maior filho
int get_max(int array[], int left, int right, int root){
    int max_child = 0;

    if(array[left] > array[root]){
        max_child = left;
    }else{
        max_child = root;
    }

    if(array[right] > array[max_child]){
        max_child = right;
    }

    return max_child;
}
```

A próxima estrutura também é um heap, porém ele utiliza um segundo vetor para armazenar a extração contínua do maoir do vetor heap.

Listing 4: Heap utilizando dois vetores

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

using namespace std;

int extract_max(int array[], int left, int right, int root);
void heapify(int array[], int i, int array_size);
void build_heap(int array[], int array_size);
void heap_sort(int array[], int array_size);
void print_heap(int array[], int array_size);

int main(){
    int array[] = {2,7,26,25,19,17,1,90,3,36};

    print_heap(array, 10);
    heap_sort(array, 10);
    print_heap(array, 10);

    return 0;
}

void print_heap(int array[], int array_size) {
    for (int i = 0; i < array_size; ++i)
```

```
    {
        printf("%d ", array[i]);
        if (array[i] != array[array_size-1])
        {
            cout << ",";
        }
    }
    cout << endl;
}

void heap_sort(int array[], int array_size)
{

  build_heap(array, array_size);

  for (int i = array_size-1; i > 1; i--)
  {
    swap(array[0], array[i]);
    heapify(array, 0, i-1);
  }
}

// Garante que cada sub-arvore seja um heap;
void build_heap(int array[], int array_size) {
    for (int i = (array_size/2)-1; i >= 0; --i)
        heapify(array, i, array_size);
}

// Coloar todos os filhos maiores na raiz
void heapify(int array[], int i, int array_size) {
    int left = 2*i;
    int right = 2*i + 1;
    int max_child = 0;

    if(left < array_size && right < array_size){

        max_child = extract_max(array, left, right, i);

        if(max_child != i){
            swap(array[i], array[max_child]);
            heapify(array, max_child, array_size);
        }
    }
}

// Retorna o maior filho
int extract_max(int array[], int left, int right, int root){
    int max_child = 0;

    if(array[left] > array[root]){
        max_child = left;
    }else{
        max_child = root;
```

```
    }

    if(array[right] > array[max_child]){
        max_child = right;
    }

    return max_child;
}
```

Após a execução dos dois programa obteve-se o seguinte gráfico

Tamanho	Dois arrays	Um array
1000	0.000985	0.000588
100000	0.076722	0.073248
1000000	1.22516	0.941388

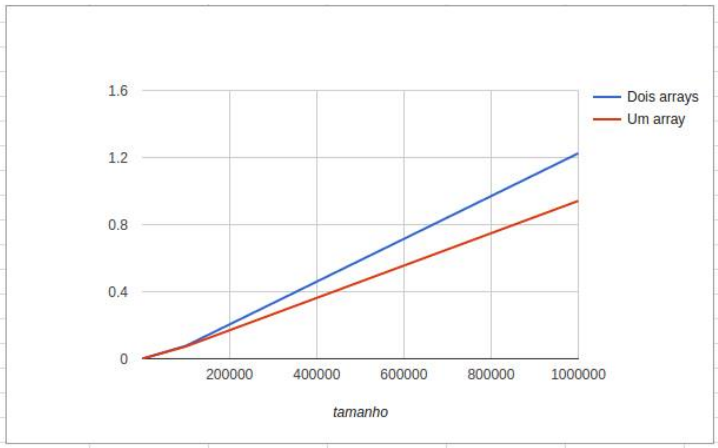

Figure 1: Crescimento linear da execução do Radix

Exercício 3

O algoritmo a seguir utiliza o algoritmo de ordenação Radix Sort que utiliza o algoritmo Counting Sort como ordenação "intermediaria".

Listing 5: Radix Sort

```
#include <iostream>
#include <ctime>
#include <stdlib.h>
#include "vector_preconfig.h"

using namespace std;

int get_max(int array[], int size){
    int max = array[0];
    for (int i = 1; i < size; i++){
        if (array[i] > max){
            max = array[i];
        }
    }
    return max;
}

void count_sort(int array[], int size, int exp){
    int output[size];
    int i, count[10] = {0};

    // Store count of occurrences in count[]
    for (i = 0; i < size; i++){
        int position = (array[i]/exp)%10;
        count[position]++;
    }

    for (i = 1; i < 10; i++){
        count[i] += count[i - 1];
    }

    // Build the output array
    for (i = size - 1; i >= 0; i--){
        int position = count[(array[i]/exp)%10];
        output[position-1] = array[i];
        count[(array[i]/exp)%10]--;
    }

    for (i = 0; i < size; i++){
        array[i] = output[i];
    }
}

void radix_sort(int array[], int size){
    int max = get_max(array, size);

    // exp =  1, 10, 100 ...
```

```cpp
    for (int exp = 1; max/exp > 0; exp *= 10){
        count_sort(array, size, exp);
    }
}

void print(int array[], int size){
    for (int i = 0; i < size; i++){
        cout << array[i] << " ";
    }
}

int main(){

    int array[100];
    preconfig(array,100);
    int size = sizeof(array)/sizeof(array[0]);

    clock_t start, end;
    start = clock();
    cout << endl;
    radix_sort(array, size);
    cout << endl;
    end = clock();

    long double tmili = (end - start)/(double)(CLOCKS_PER_SEC/1000);
    cout << "Time: " << tmili << " ms" << endl;

    return 0;
}
```

Como mostrado no gráfico, o crescimento do radix para a quantidade de números é essencialmente linear.

Qnt Dados	Tempo
10	160
100	171
1000	147
10000	155
100000	143
1000000	151

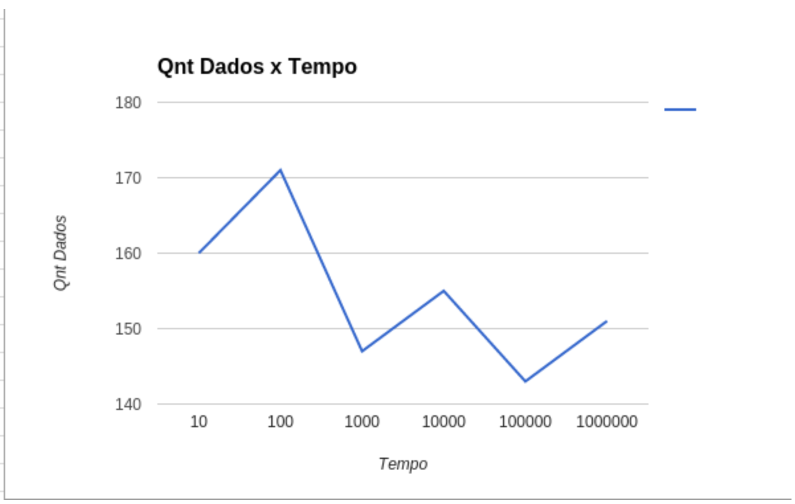

Figure 2: Crescimento linear da execução do Radix