F12 – Simulação da Prova 2 - Solução

Atividade Avaliativa para Cômputo de Frequência
Algoritmos e Estruturas de Dados 1 (1001502)

Orientações Gerais

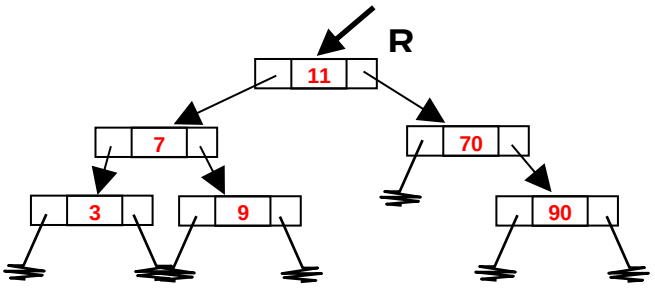
- **Tempo para elaboração da simulação / prova: 3h**.

Orientações Quanto a Notação, Nomes das Variáveis, e Estruturas

- Use os **mesmos nomes fornecidos no enunciado** (R, X, Ok, V, N, Heap, LastPosition, etc.). Utilize variáveis auxiliares temporárias, o tanto quanto forem necessárias. É só declarar e usar. Mas **não considere a existência de nenhuma outra variável permanente**, além das definidas no enunciado. Não considere prontas para uso nenhuma operação, salvo se explicitamente indicado no enunciado da questão.
- Considere as **estruturas exatamente conforme definido no enunciado**, seja no texto da questão, seja nos diagramas.
- Para o desenvolvimento de algoritmos, é possível usar a notação conceitual adotada no Livro Texto (p = NewNode; Deletenode(P), P->Info e P->FDir (filho direito de P), P->FEsq (filho esquerdo de P) e, quando desejar P->Pai (pai de P), sendo P uma variável do tipo NodePtr - ponteiro para nó). Também é possível implementar em C ou C++.

Questão 1 (3 pontos) Considere uma ***Árvore Binária de Busca*** (ABB), de raiz R, implementada com alocação encadeada e dinâmica de memória, conforme os diagramas abaixo. A Árvore não contém elementos repetidos. Implemente a operação:

void Remove (variável por referência R do tipo ABB, Variável X do tipo int, variável por referência Ok do tipo bool);
/* esta função deve procurar X na ABB R e, caso encontrar, deve remover da árvore e retornar Ok = true. Caso não encontrar, Ok deve retornar false. O tipo ABB é análogo ao tipo NodePtr, ou seja, ponteiro para o nó da árvore */




```
Remove (parâmetro por referência R tipo ABB, parâmetro X do tipo Inteiro, parâmetro por referência Ok tipo Boolean) {
// Remove X da ABB R. Ok retorna Verdadeiro para o caso de X ter sido encontrado e removido, e Falso  caso contrário
Variável Aux do tipo NodePtr;
Se (R == Null)
Então       Ok = Falso;        // Caso 1: Árvore Vazia: não remove e encerra o algoritmo.
Senão       Se (R→Info > X)
            Então       Remove (R→Esq, X , Ok); // Caso 3: remove X da Subárvore Esq de R
            Senão       Se (R→Info < X)
                        Então Remove(R→Dir, X, Erro); // Caso 4: remove X da Subárv Dir de R
                        Senão
/* Caso 2: Encontrou X - Remove e Ajusta a Árvore. Existem 3 casos: Nó com 0, 1 ou 2 Filhos - Quadros 8.23 e 8.24 */
{
Aux = R;
Ok = Verdadeiro;

Se (R→Esq = Null E R→Dir = Null)  // Caso 2a: Zero Filhos
Então       { DeleteNode(Aux); R = Null; } // fim Caso 2a
Senão       Se (R→Dir != Null E Esq( R ) != Null)  // Caso 2c: 2 Filhos
            Então       {
                        /* Acha o Nó que contém o Maior Elemento da Subárvore Esquerda
                         de R. O maior é o elemento mais a direita da Subárvore. Ele nunca terá
                        o Filho Direito. */
                        Aux = R→Esq;
                        Enquanto (Aux→Dir != Null) Faça Aux = Aux→Dir;

                        /* Substitui o valor de R→Info - que é o elemento que estarmos querendo
                         eliminar - pelo valor do Maior da Subárvore Esquerda de R. A Árvore
                        ficará com 2 elementos com o mesmo valor. */
                         R→Info = Aux→Info;   // Aux aponta para o Nó que contém o Maior

                        /* Remove o valor repetido da Subárvore Esquerda de R, através de
                         uma chamada recursiva. Atenção aos parâmetros. Não estamos mais
                         removendo X, mas sim Aux→Info, que está repetido. Não estamos mais
                         removendo de R e sim de R→Esq */
```

```
                Remove(R→Esq, Aux→Info, Ok);
                } // fim Caso 2c

        Senão   {                       // Caso 2b: 1 Único Filho
                Se (R→Esq == Null)
                Então   R = R→Dir;      // "puxa" o Filho Direito; Filho esquerdo é nulo
                Senão   R = R→Esq;      // "puxa" o Filho Esquerdo; Filho direito é nulo
                DeleteNode (Aux);       // desaloca o Nó
                } // fim Caso 2b
} // fim remove
```

Obs: Os valores do diagrama estavam fora da sequência de uma ABB na folha de questão da simulação. Ajustamos aqui.

Questão 2 (3 pontos) Na ordenação de vetores pelo algoritmo ***BubbleSort***, percorremos o vetor do início ao fim, várias vezes, invertendo de posição elementos adjacentes que estiverem fora de ordem. Em cada passada um elemento do vetor fica na posição correta. Na primeira passada o último elemento já está na posição correta. Na segunda passada, o penúltimo elemento já está na posição correta. Assim, uma otimização do BubbleSort é sempre reduzir o número de elementos a serem comparados em cada chamada. Em outra otimização do BubbleSort, interrompemos o processo de ordenação quando em uma mesma "passada" (ou seja, ao percorrer o vetor do início ao fim, como no diagrama abaixo), nenhuma troca for efetuada.

(a) Desenvolva um procedimento para ordenar um vetor pelo método BubbleSort, com as duas otimizações indicadas no enunciado da questão:

```
void BubbleSort (variável por referência v do tipo vetor de inteiros, variável n do tipo inteiro)
/* ordena o vetor V de tamanho N, pelo algoritmo BubbleSort, interrompendo o processo quando nenhuma troca for efetuada em uma
"passada". Além disso, */

{
   int j, i, aux, pos_ult_inv, lim_dir;
   lim_dir = n;
   for (j = 0; j < n; j++)
   {
      pos_ult_inv = 0;
      for (i = 1; i < lim_dir; i++)
         if (v[i - 1] > v[i])
         {
            aux = v[i - 1];
            v[i - 1] = v[i];
            v[i] = aux;
            pos_ult_inv = i; // marca a posição da última inversão
         }
      lim_dir = pos_ult_inv; // o limite direito passará a ser a posição da última inversão. Dali para a direita, o vetor já está ordenado.
   }
}
```

(b) Qual é a organização inicial do vetor para o **pior caso** de execução do algoritmo? Qual é o número de passos (comparações) no pior caso? Use a notação big-O (limite assintótico superior) em função do tamanho ***N*** de elementos do vetor a ser ordenado.

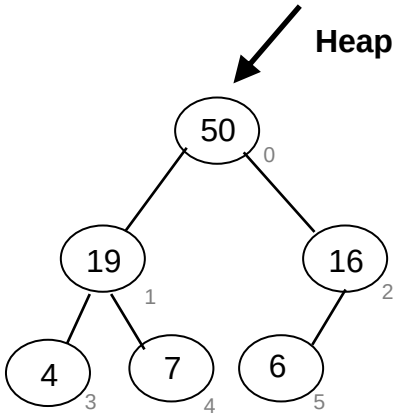
No pior caso, o menor elemento do vetor estar é na última posição do vetor inicial. Nesse caso, o número de comparações será $O(n^2)$.

(c) Qual é a organização inicial do vetor para o **melhor** caso de execução do algoritmo? Qual é o número de passos (comparações) no melhor caso?

No melhor caso, o vetor inicial está ordenado ou contém inversões apenas entre elementos adjacentes. Nesse caso, o número de comparações será O(n).

Questão 3 (4 pontos) Em um ***Heap-Binário-de-Máximo***, o elemento que está em um determinado ***nó*** da árvore tem valor maior ou igual do que o valor de seus ***filhos*** direito e esquerdo. Entre os nós ***irmãos***, não há necessariamente uma ordenação, como mostra o diagrama (a), a seguir. Um ***Heap-Binário-de-Máximo*** proporciona uma implementação eficiente de filas de prioridades, pois na operação de remoção, o acesso ao maior elemento (ou elemento de maior prioridade) é direto, já que este estará armazenado na raiz. Nesta questão os itens ***a***, ***b,*** **c** *e* **d** são diagramas de explicação, e você deve responder os itens ***i*** e ***ii.***

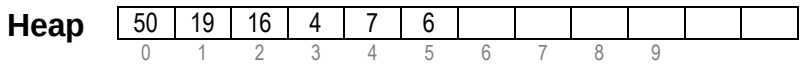
a)



Um ***Heap-Binário-de-Máximo*** pode ser implementado em um vetor, como mostra o diagrama (b), a seguir. Note que o vetor do diagrama (b) corresponde à implementação da árvore do diagrama (a). Nessa implementação, temos que:

- O ***pai*** de um elemento armazenado na posição i encontra-se armazenado na posição (i-1)/2;
- O ***filho-esquerdo*** de um elemento armazenado na posição i encontra-se armazenado na posição 2*i+1;
- O **filho-direito** de um elemento armazenado na posição i encontra-se armazenado na posição 2*i+2.

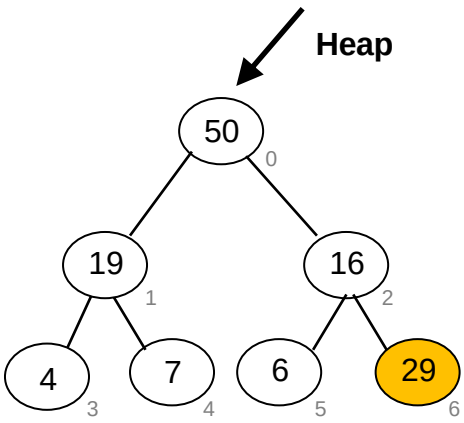
b)



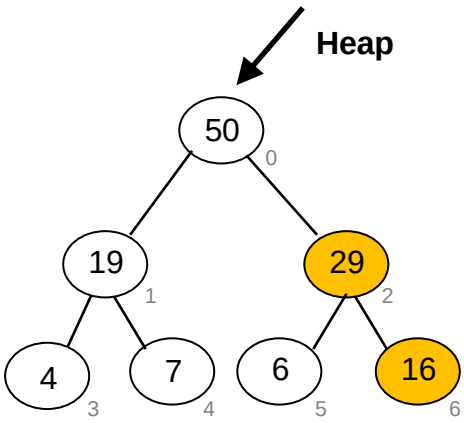
Na operação de **remoção de um elemento de um Heap-Binário-de-Máximo**, retiramos o elemento que está na raiz, e então corrigimos o Heap, escolhendo o elemento mais adequado para posicionar na raiz, dentre o filho direito e o filho esquerdo, e assim sucessivamente, ***descendo***, até chegar ao final da árvore/ vetor.

Na operação de **inserção em um Heap-Binário-de-Máximo**, acrescentamos um elemento no "final" (ou seja, na primeira posição livre) do vetor (será um nó folha/sem filhos na árvore). Em seguida, vamos corrigindo o heap/árvore/vetor, trocando de posição o elemento que acabou de ser inserido com seu pai, sucessivamente, até que seja encontrada a posição adequada na árvore, sempre subindo, no máximo até chegar à raiz. Por exemplo, a partir da situação dos diagramas (a) e (b), foi inserido o elemento 29 ao "final" do vetor - posição 6 (diagrama c). Em seguida, esse elemento trocou de posição com seu pai (valor 16 – posição 2 do vetor), de modo a ficar na posição correta do Heap - diagrama (d). Se o valor inserido fosse 52, ao invés de 29, ocorreria ainda mais uma troca, entre o 52 e o 50.

c)



d)



i) Implemente a operação:

void **CorrigeHeapSubindo** (variável por referência Heap do tipo vetor de inteiros, variável LastPosition do tipo inteiro)

/* este procedimento corrige o Heap, posicionando o elemento que acabou de ser inserido em Heap[LastPosition] em sua posição adequada no heap/árvore/vetor. */

```
{
   int corr = LastPosition;
   while (corr > 0 && Heap[PAI(corr)] < Heap[corr])
   {
      troca(&Heap[corr], &Heap[PAI(corr)]);
      corr = PAI(corr);
   }
}

int Pai (int i )
{ return (i-1)/2 }

void troca (int *i, int *j) {
int aux = i;
i = j;
j = aux; }
```

ii) No pior caso, quantas trocas ocorrerão em um vetor com N elementos, ao executarmos a operação CorrigeHeapSubindo?

$Log_2 N$