Estruturas de dados II

Árvores Binárias de Busca e de Huffman

André Tavares da Silva

andre.silva@udesc.br

Pereira, cap. 14

UDESC

Árvore Binária de Busca

- Uma árvore binária de busca (abreviada por ABB) serve para o armazenamento de dados no computador e a sua subsequente busca de informações.
- Em uma árvore binária de busca cada nó contém um campo contendo uma chave de busca, podendo haver outras informações, além dos ponteiros esquerda e direita.
- A chave de busca especifica em geral um determinado registro ou informação, como número de identidade, CPF, etc.
- Criada a árvore, é possível localizar qualquer elemento em O(log n) dependendo do balanceamento da mesma, podendo ser inseridos novos registros ou removidos para atualizar a árvore.

UDESC

Árvore Binária de Busca

- Uma árvore de busca binária A é uma árvore binária vazia ou, então, uma árvore binária cuja raiz armazena um item x e tem as seguintes propriedades:
  - Todo item na subárvore esquerda de A é menor ou igual a x.
  - Todo item na subárvore direita de A é maior que x.
  - Cada subárvore de A é uma árvore de busca binária.

- Uma consequência importante dessas propriedades é que a projeção de uma árvore de busca binária sempre produz uma sequência ordenada de itens.

Inserção em Árvores Binárias de Busca

- A função para inserir um novo item numa árvore de busca binária usa um percurso pré-ordem: primeiro ela verifica se a árvore está vazia e, caso esteja, ela coloca o novo item na raiz da árvore; se não, se o novo item for menor ou igual àquele na raiz da árvore, recursivamente, ela o insere na subárvore esquerda; caso contrário, recursivamente, ela o insere na subárvore direita. No código abaixo, como o ponteiro para a árvore pode ser alterado durante uma inserção, ele deve ser passado por referência.

```
void insere(float valor) {
    No *no=root, *prev = NULL;
    while(no != NULL) {
        prev = no;
        if(valor <= no->valor)
            no = no->esquerda;
        else
            no = no->direita;
    }
    if(root == NULL) root = new No(valor);
    else if(valor < prev->valor) prev->esquerda = new No(valor);
    else prev->direita = new No(valor);
}
```

Consulta em Árvores Binárias de Busca

- Consultar se determinado novo nó existe em uma árvore binária de busca é uma tarefa similar à inserção de um novo nó. Basicamente, o que temos que fazer é percorrer os nós da árvore usando o seguinte conjunto de passos: primeiro compare o valor buscado com a raiz; se o valor é menor do que a raiz: vá para a subárvore da esquerda; se o valor é maior do que a raiz: vá para a subárvore da direita; aplique o método até que a raiz seja igual ao valor buscado.

```
int busca(float valor) {
    No *no=root, *prev = NULL;
    while(no != NULL) {
        prev = no;
        if(valor <= no->valor)
            no = no->esquerda;
        else
            no = no->direita;
    }
    if(root == NULL) return 0; // Árvore vazia
    else if(valor == prev->valor) return 1;
    Return 0; // elemento não localizado
}
```

UDESC

Remoção em Árvores Binárias de Busca

- A remoção em árvore de busca binária é mais difícil do que a inserção e a busca. Primeiro vamos considerar a remoção de um item máximo da árvore de busca binária. Para isso, partindo da raiz da árvore, seguimos sempre o ponteiro à direita, enquanto ele não for NULL. Quando chegarmos a um nó cujo ponteiro à direita é NULL, estaremos no nó que guarda um item máximo da árvore. Então, basta remover esse nó e devolver o item que ele armazena.

- O nó que guarda um item máximo da árvore de busca binária não pode ter filho à direita (senão, ele não seria máximo), mas ele pode ter um filho à esquerda. Portanto, quando esse nó é removido, o ponteiro que o apontava deve passar a apontar seu filho à esquerda (caso ele não tenha filho à esquerda, o ponteiro que o apontava ficará nulo).

UDESC

Remoção em Árvores Binárias de Busca

- Agora, vamos considerar a remoção de um item x, que está na raiz da árvore de busca binária A. Então, há três casos possíveis: Se o nó apontado por A não tem filhos, então ele deve ser desalocado e o ponteiro A deve ser anulado (isto é, a árvore fica vazia). Se o nó apontado por A tem um único filho, então ele deve ser desalocado e o ponteiro A deve passar a apontar esse filho. Se o nó apontado por A tem dois filhos, então o item nesse nó deve ser substituído por um item máximo removido de sua subárvore esquerda.
  - Para entender a estratégia adotada nesse último caso, lembre que, numa árvore de busca binária A, cada item na subárvore esquerda de A é menor que todo item na subárvore direita de A. Portanto, substituindo o item na raiz de A por um item máximo removido de sua subárvore esquerda, podemos garantir que a árvore resultante da remoção do item A ainda será uma árvore de busca binária.

Remoção em Árvores Binárias de Busca

- Para generalizar essa ideia, precisamos considerar que o item a ser removido pode estar em qualquer nó da árvore, não apenas na raiz. Nesse caso, antes de removê-lo, precisamos encontrá-lo na árvore. Para isso, basta adaptar a lógica da função de busca em árvore.

```
void  remove (float valor)  {
   No *no=root, *prev = NULL;
   int filho_esq = 1;
   while (no != NULL) {
      prev = no;
      if (valor <= no->valor)  {
         no = no->esquerda;
         filho_esq = 1;
      } else {
         no = no->direita;
         filho_esq = 0;
      }
   }
   if (no == NULL)
      return; // não localizado ou árvore vazia
   // se folha:
   else if ((no->esquerda == NULL) && (no->esquerda == NULL)) {
      if(no==root) root = NULL;
      else if (filho_esq) prev->esquerda = NULL;
      else prev->direita == NULL;
// (continua...)
```

```
// continuação:
   } else if (no->esquerda == NULL) {
    // se não tem filho a esquerda, substitui pela subarvore dir
      if (filho_esq) prev->esquerda = no->direita;
      else prev->direita = no->direita;
   } else if (no->direita == NULL) {
    // se não tem filho a direita, substitui pela subarvore esq
      if (filho_esq) prev->esquerda = no->esquerda;
      else prev->direita = no->esquerda;
   } else {
     // se possui mais de um filho, procura nó máximo (dir/esq)
      No *maximo = remmax(no);
      if (filho_esq) prev->esquerda = maximo;
      else prev->direita = maximo;
   }
}
```

UDESC

Exercícios

1) Supondo que a árvore seja criada inserindo valores em uma sequência aleatória de valores utilizando sempre um mesmo conjunto de dados. A árvore sempre será criada com a mesma estrutura ou poderá resultar em árvores diferentes?

2) Uma árvore binária de busca sempre será balanceada?

3) Indique um caso ideal em relação à ordem de inserção de nós para criação de uma árvore de busca balanceada. Explique o motivo.

4) Informe qual o pior caso ao criar uma árvore de busca. Justifique.

UDESC

Exercícios de implementação

1) Crie um programa para gerar uma ABB (árvore binária de busca). O programa deve ser capaz de inserir novos elementos, remover elementos da árvore, realizar uma busca e listar todos os elementos da árvore de maneira ordenada.

2) Escreva programa deve ser capaz de ler uma árvore binária (qualquer) e informar as seguintes informações sobre a árvore:

   - Se a árvore é vazia;
   - Tamanho da árvore (número de nós)
   - Altura ou profundidade da árvore (nível máximo)
   - Largura da árvore em cada nível
   - Se árvore está completa
   - Se a árvore está cheia

Arvore de Huffman

UDESC

Árvore de Huffman

- Uma árvore de Huffman é uma árvore estritamente binária usada para compressão de arquivos, visando reduzir o espaço necessário para armazená-los em disco ou o tempo necessário para transmiti-los por um canal de comunicação.
- Uma árvore binária completa, chamada de árvore de Huffman é construída recursivamente a partir da junção dos dois símbolos de menor probabilidade, que são então somados em símbolos auxiliares e estes símbolos auxiliares recolocados no conjunto de símbolos. O processo termina quando todos os símbolos forem unidos em símbolos auxiliares, formando uma árvore binária. A árvore é então percorrida, atribuindo-se valores binários de 1 ou 0 para cada aresta, e os códigos são gerados a partir desse percurso.

UDESC

Árvore de Huffman

- De fato, como veremos a seguir, uma árvore de Huffman é uma estrutura que representa um código de tamanho variável, que minimiza o número de bits necessários para representar um arquivo.
- Diferentemente do código ASCII, que é padrão, o código de Huffman varia em função do arquivo a ser comprimido (e, portanto, deve ser criado especificamente para cada arquivo).
- O código de Huffman é livre de prefixo, isto é, para quaisquer dois caracteres a e b no alfabeto S, o código de a não é um prefixo do código de b. Essa propriedade garante que a descompressão de um arquivo s pode ser feita sem ambiguidade.

UDESC

Árvore de Huffman

- Dada uma cadeia de caracteres s, representando um arquivo a ser comprimido, uma árvore de Huffman correspondente pode ser construída do seguinte modo:
  - Para cada caractere ASCII c, obtenha a frequência f (c) de c em s.
  - Seja S o conjunto de caracteres ASCII c, tais que f(c) > 0.
  - Para cada caractere c em S, crie uma árvore binária correspondente com uma folha contendo f(c). O conjunto de árvores criadas nesse passo é uma floresta. O valor na raiz de cada árvore da floresta é o peso da árvore.
  - Enquanto houver mais que uma árvore na floresta: remova duas árvores de pesos mínimos, digamos A_e e A_d; em seguida, crie uma árvore binária A cuja raiz guarde a soma dos pesos de A_e e A_d e cujos filhos sejam A_e e A_d; finalmente, insira a árvore A na floresta.

UDESC

Árvore de Huffman

- No fim da repetição, a árvore que sobra é uma árvore de Huffman para o arquivo representado por s.
- Para obter os códigos comprimidos, rotule as ligações à esquerda com 0 e as ligações à direita com 1.
- Depois disso, o código de Huffman para cada caractere c é a sequência de bits que rotula o caminho que vai da raiz da árvore até a folha associada ao caractere c.

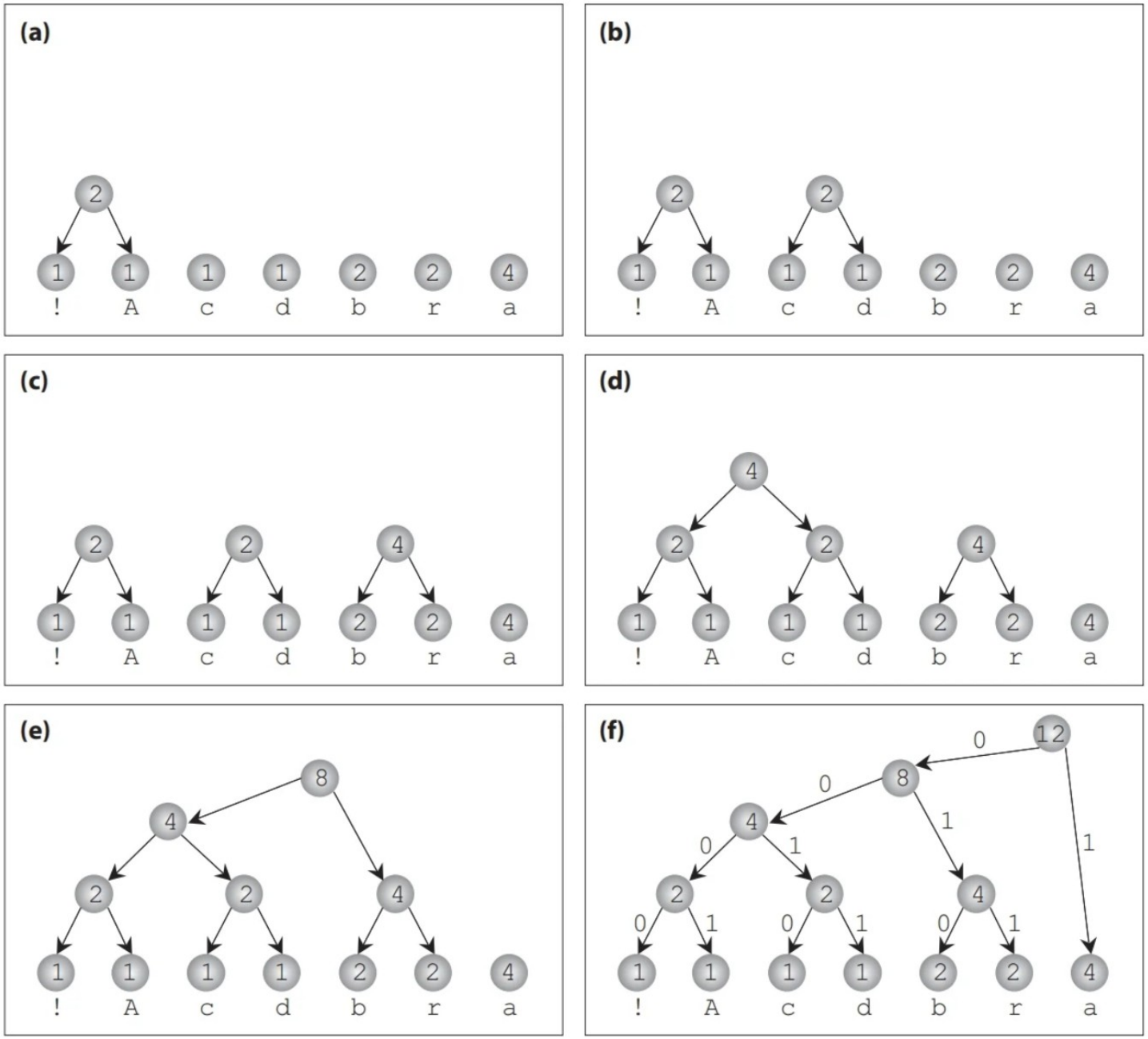


Figura 14.2 | Criação de uma árvore de Huffman para σ = "*Abracadabra*!".

UDESC

Código de Huffman – histograma de frequências

- Veremos uma implementação do código de Huffman. Dada uma cadeia de caracteres s, representando o conteúdo de um arquivo a ser comprimido, é criado um vetor de inteiros *f* tal que, para cada caractere ASCII c, *f*[c] é a frequência de c em s:

```
int *freq(char *s) {
  static int f[256];
  for(int i=0; i<256; i++) f[i] = 0;
  for(int i=0; s[i]; i++) f[s[i]]++;
  return f;
}
```

UDESC

Código de Huffman – criação da árvore

- Para criar uma árvore de Huffman, vamos usar a seguinte estrutura de dados:

```
typedef struct arvh {
  char chr;
  int frq;
  struct arvh *esq, *dir;
} *Arvh;
```

UDESC

Código de Huffman – criação da árvore

- Para criar uma árvore de Huffman, uma função recebe como entrada um caractere, sua frequência e duas árvores de Huffman e, como saída, ela devolve o endereço de um nó criado e preenchido com esses valores.
- Assim, cada folha numa árvore de Huffman guardará um caractere e sua frequência.

```
Arvh criaAH(Arvh e, char c, int f, Arvh d) {
  Arvh n = malloc(sizeof(struct arvh));
  n->esq = e;
  n->chr = c;
  n->frq = f;
  n->dir = d;
  return n;
}
```

UDESC

Código de Huffman – criação da árvore

- Uma floresta será representada por um vetor F com m árvores de Huffman, em ordem decrescente de peso. Para garantir a ordenação da floresta, usaremos a função abaixo. Dadas uma árvore A e uma floresta F com m árvores, essa função insere a árvore A em F, em ordem decrescente de peso, e incrementa m. O parâmetro m, que indica o tamanho da floresta, é passado por referência.

```
void insf(Arvh A, Arvh F[], int *m) {
  int i = *m;
  while( i>0 && F[i-1]->frq < A->frq ) {
    F[i] = F[i-1];
    i--;
  }
  F[i] = A;
  (*m)++;
}
```

Código de Huffman – criação da árvore

- Para remoção de uma árvore da floresta sabendo que F é um vetor de árvores em ordem decrescente de peso, basta remover a última árvore do vetor retornando ela como resposta:

```
Arvh remf(Arvh F[], int *m) {
  if( *m == 0 ) return NULL;
  return F[--(*m)];
}
```

UDESC

Código de Huffman – criação da árvore

- Para criar a árvore de Huffman a partir de uma cadeia de caractere s:

```
Arvh huffman(char *s) {
  Arvh F[256];
  int m=0;
  int *f = freq(s);
  for(int c=0; c<256; c++)
    if( f[c]>0 ) insf(criaAH(NULL,c,f[c],NULL),F,&m);
  while( m>1 ) {
    Arvh d = remf(F,&m);
    Arvh e = remf(F,&m);
    insf(criaAH(e,'-',e->frq+d->frq,d),F,&m);
  }
  return F[0];
}
```

Código de Huffman – codificando/comprimindo

- Construída a árvore A, agora podemos gerar o código binário a partir dela e da própria cadeia de caractere s. Para definir qual sequência binária para cada letra do alfabeto:

```
void tabela(Arvh A, char *H[]) {
  static char c[256], t = -1;
  if( A == NULL ) return;
  if( A->esq == NULL && A->dir == NULL )
    H[A->chr] = strndup(c,t+1);
  else {
    t++;
    c[t] = '0'; tabela(A->esq,H);
    c[t] = '1'; tabela(A->dir,H);
    t--;
  }
}
```

Código de Huffman – codificando/comprimindo

- Sabendo o código de cada letra, podemos agora gerar o código a partir da string e da árvore de Huffman:

```
void comprimir(char *s, Arvh A) {
  char *T[256];
  for(int c=0; c<256; c++) T[c] = NULL;
  tabela(A,T);
  for(int i=0; s[i]; i++) printf("%s",T[s[i]]);
  for(int c=0; c<256; c++) free(T[c]);
}
```

Código de Huffman – decodificando/descomprimindo

- Para descomprimir uma cadeia de dígitos d, basta ter a árvore de Huffman A, que foi usada para comprimi-la. A cada valor binário, a árvore é percorrida até chegar no nos nós folha:

```
void descomprimir(char *d, Arvh A) {
  if( A == NULL ) return;
  Arvh n = A;
  for(int i=0; d[i]; i++) {
    n = (d[i]=='0') ? n->esq : n->dir;
    if( n->esq == NULL && n->dir == NULL ) {
      printf("%c",n->chr);
      n = A;
    }
  }
}
```

Código de Huffman – exemplo **Jararaca**

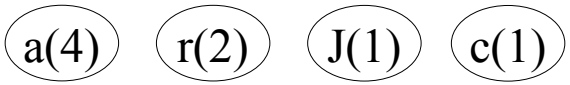
- Vamos ver como criar a árvore para Jararaca. Primeiro o histograma de frequência das letras de “Jararaca”:

4	2	1	1
a	r	J	c

Código de Huffman – exemplo **Jararaca**

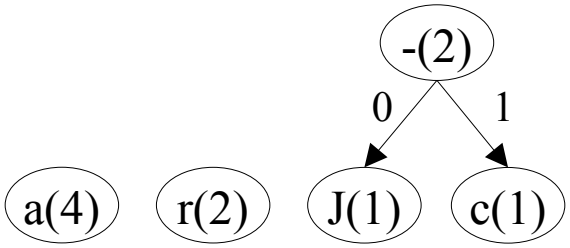
- Criando a árvore, primeiramente temos uma floresta com apenas o nó raiz/folha (um único nó por árvore):

4	2	1	1
a	r	J	c



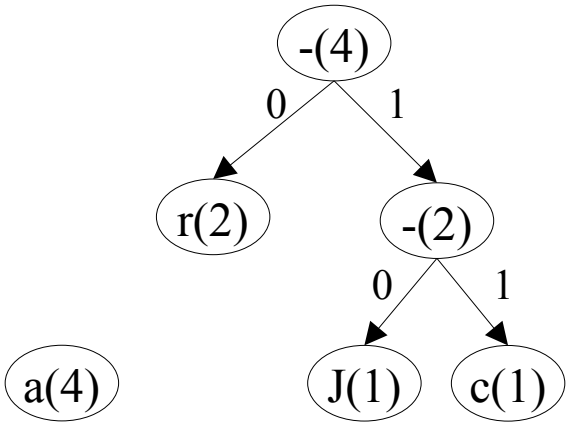
Código de Huffman – exemplo **Jararaca**

- A cada par com menores valores, juntamos os nós em uma nova árvore. Removemos “J” e “c” da floresta e criamos outra árvore com um nó raiz e ambos filhos dessa nova (folhas nesse caso):



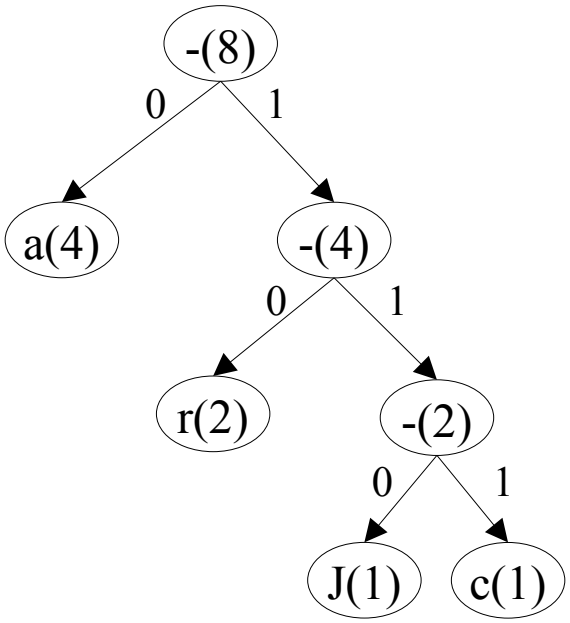
Código de Huffman – exemplo **Jararaca**

- A cada par com menores valores, juntamos os nós em uma nova árvore. Removemos “r” e última árvore e criamos outra com um nó raiz e ambos filhos dessa nova:



Código de Huffman – exemplo **Jararaca**

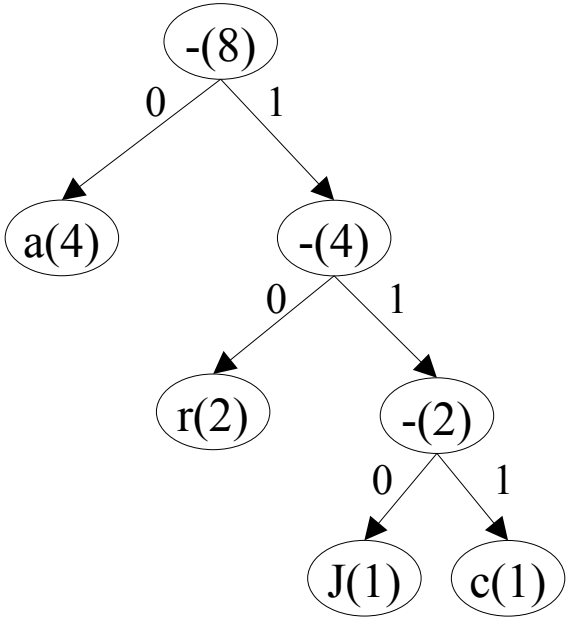
- A cada par com menores valores, juntamos os nós em uma nova árvore. Por fim, juntamos o “a” e última árvore e teremos a árvore de Huffman:



Código de Huffman – exemplo **Jararaca**

- Temos então o seguinte código para cada letra:

```
a: 0
r: 10
J: 110
c: 111
```

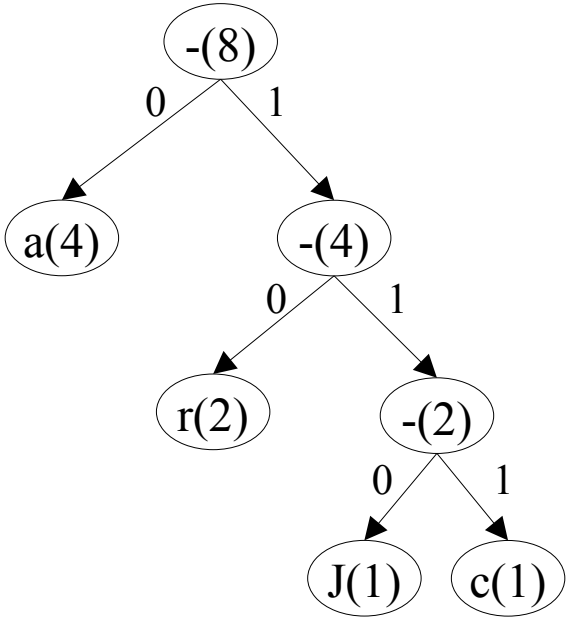


Código de Huffman – exemplo **Jararaca**

- Codificando:

```
a: 0
r: 10
J: 110
c: 111
```

código: 11001001001110



UDESC

Exercícios de implementação

1) A partir das funções apresentadas aqui, crie um programa para gerar uma árvore de Huffman. O programa deve receber uma frase, gerar a árvore e exibir o código para gerar a frase.

2) Faça uma função para exibir a árvore de Huffman gerada.

Estruturas de dados II

Árvores Binárias de Busca e de Huffman

André Tavares da Silva

andre.silva@udesc.br