BCC202 - Estruturas de Dados I

Aula 20: Árvores de Pesquisa

Pedro Silva

Universidade Federal de Ouro Preto, UFOP
Departamento de Computação, DECOM
Email: **silvap@ufop.edu.br**

2021

Abordagens de pesquisa em Memória Primária

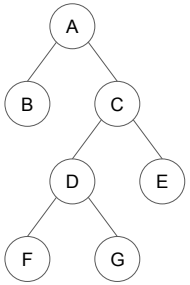
- Pesquisa Sequencial.
- Pesquisa Binária.
- Árvores de Pesquisa:
  - Árvores Binárias de Pesquisa.
  - Árvores AVL.
- Transformação de Chave (*Hashing*):
  - Listas Encadeadas.
  - Endereçamento Aberto.
  - *Hashing* Perfeito.

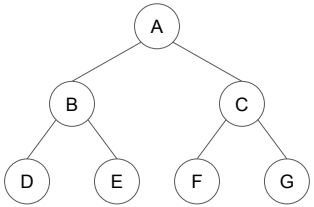
Conteúdo

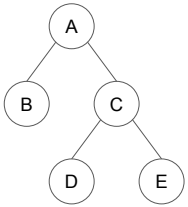
Conteúdo

Classificação de Árvores

- Árvore Estritamente Binária.
  - Se cada nó não-folha em uma árvore binária não tem subárvores esquerda e direita vazias.

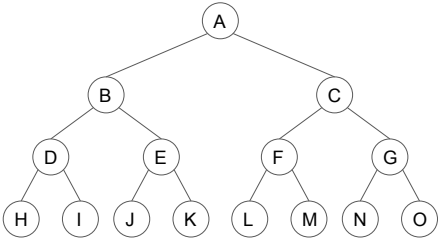


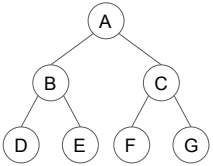


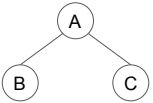


Classificação de Árvores

- Árvore Binária Completa.
  - Uma árvore binária completa de nível n é a árvore estritamente binária, onde todos os nós folhas estão no nível n.

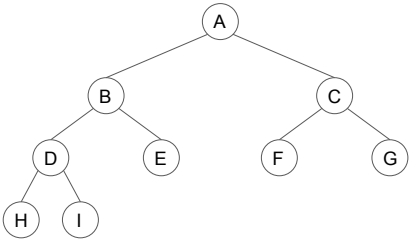


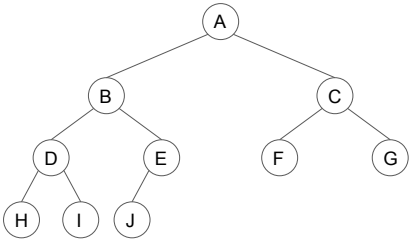




Classificação de Árvores

- Árvore Binária Quase Completa
  - Uma árvore binária de nível n é uma árvore binária quase completa se:
    - Cada nó folha na árvore esta no nível n ou no nível $n-1$.
    - Para cada nó n_d na árvore com um descendente direito no nível n, todos os descendentes esquerdos de n_d que são folhas estão também no nível n.





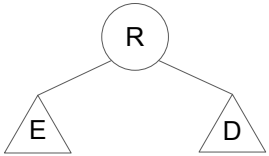
Conteúdo

Árvores de Pesquisa

- A árvore de pesquisa é uma estrutura de dados muito eficiente para armazenar informação.
- Particularmente adequada quando existe necessidade de considerar todos ou alguma combinação de:
  1. Acesso direto e sequencial eficientes.
  2. Facilidade de inserção e retirada de registros.
  3. Boa taxa de utilização de memória.
  4. Utilização de memória primária e secundária.

Árvores Binárias de Pesquisa

- Para qualquer nó que contenha um registro.

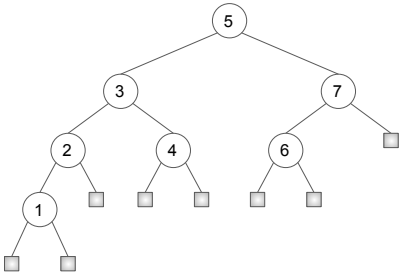


- Temos a relação invariante.

  - Os registros com chaves menores estão na subárvore à esquerda.
  - Os registros com chaves maiores estão na subárvore à direita.

Árvores Binárias de Pesquisa

- O nível do nó raiz é 0.
  - Se um nó está no nível i então a raiz de suas subárvores estão no nível $i + 1$.
  - A altura de um nó é o comprimento do caminho mais longo deste nó até um nó folha.
  - A altura de uma árvore é a altura do nó raiz.



TAD - Árvore Binária de Pesquisa

```c
typedef long TChave;

typedef struct {
    /* outros componentes */
    TChave Chave;
} TItem;

typedef struct No {
  TItem item;
  struct No *pEsq, *pDir;
} TNo;

typedef TNo * TArvore;
```

Árvores Binárias de Pesquisa

- Para encontrar um registro com uma chave x:
  - Compare-a com a chave que está na raiz.
  - Se x é menor, vá para a subárvore esquerda.
  - Se x é maior, vá para a subárvore direita.
  - Repita o processo recursivamente, até que a chave procurada seja encontrada ou um nó folha seja atingido.
  - Se a pesquisa tiver sucesso então o conteúdo do registro retorna no próprio registro x.

Pesquisa na Árvore

```
1  int TArvore_Pesquisa(TArvore *pRaiz, TChave c, TItem *pX) {
2      if (pRaiz == NULL)
3          return 0;
4
5      if (c < pRaiz->item.chave)
6          return TArvore_Pesquisa(pRaiz->pEsq, c, pX);
7      if (c > pRaiz->item.chave)
8          return TArvore_Pesquisa(pRaiz->pDir, c, pX);
9
10     *pX = pRaiz->item;
11     return 1;
12 }
```

Pesquisa na Árvore (não recursiva)

```
1  int TArvore_Pesquisa(TArvore *pRaiz, TChave c, TItem *pX) {
2      TNo *pAux;
3
4      pAux = pRaiz;
5      while (pAux != NULL) {
6          if (c == pAux->item.chave) {
7              *pX = pAux->item;
8              return 1;
9          }
10         else if (c > pAux->item.chave)
11             pAux = pAux->pDir;
12         else
13             pAux = pAux->pEsq;
14     }
15     return 0; // não encontrado!
16 }
```

Procedimento para Inserir na Árvore

- Onde inserir?
  - Atingir um ponteiro nulo em um processo de pesquisa significa uma pesquisa sem sucesso.
  - O ponteiro nulo atingido é o ponto de inserção.
- Como inserir?
  1. Cria célula contendo registro.
  2. Procurar o lugar na árvore.
  3. Se registro não tiver na árvore, insere-o.

Inserir na Árvore

```
1  int TArvore_Insere(TNo** ppR, TItem x) {
2      if (*ppR == NULL) {
3          *ppR = TNo_Cria(x);
4          return 1;
5      }
6
7      if (x.chave < (*ppR)->item.chave)
8          return TArvore_Insere(&((*ppR)->pEsq), x);
9      if (x.chave > (*ppR)->item.chave)
10         return TArvore_Insere(&((*ppR)->pDir), x);
11
12     return 0; // elemento jah existe
13 }
```

Inserir na Árvore (apenas árvore não vazia)

```
1  int TArvore_Insere(TNo *pRaiz, TItem x) {
2      if (pRaiz == NULL) return -1; // arvore vazia
3
4      if (x.chave < pRaiz->item.chave) {
5          if (pRaiz->pEsq == NULL) {
6              pRaiz->pEsq = TNo_Cria(x);
7              return 1;
8          }
9          return TArvore_Insere(pRaiz->pEsq, x);
10     }
11     if (x.chave > pRaiz->item.chave) {
12         if (pRaiz->pDir == NULL) {
13             pRaiz->pDir = TNo_Cria(x);
14             return 1;
15         }
16         return TArvore_Insere(pRaiz->pDir, x);
17     }
18     return 0; // elemento jah existe
19 }
```

Inserir na Árvore

```
1 void TArvore_Insere_Raiz(TNo **ppRaiz, TItem x) {
2     if (*ppRaiz == NULL) {
3         *ppRaiz = TNo_Cria(x);
4         return;
5     }
6
7     TArvore_Insere(*ppRaiz, x);
8 }
```

Inserir na Árvore (não recursivo)

```
1  int TArvore_Insere(TNo **ppRaiz, TItem x) {
2      TNo **ppAux;
3      ppAux = ppRaiz;
4
5      while (*ppAux != NULL) {
6          if (x.chave < (*ppAux)->item.chave)
7              ppAux = &((*ppAux)->pEsq);
8          else if (x.chave > (*ppAux)->item.chave)
9              ppAux = &((*ppAux)->pDir);
10         else
11             return 0;
12     }
13     *ppAux = TNo_Cria(x);
14     return 1;
15 }
```

Procedimento que Cria um Nó

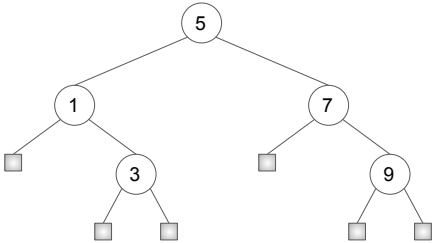
```
1 TNo *TNo_Cria(TItem x) {
2     TNo *pAux = (TNo*)malloc(sizeof(TNo));
3     pAux->item = x;
4     pAux->pEsq = NULL;
5     pAux->pDir = NULL;
6     return pAux;
7 }
```

Procedimentos para Inicializar a Árvore

```
1 void TArvore_Inicia(TNo **pRaiz) {
2     *pRaiz = NULL;
3 }
```

Retirada de um Registro da Árvore

- Como retirar?



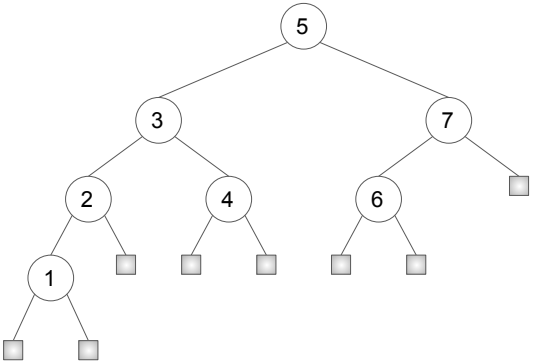
- Folha?
- Nó interno, contendo um filho?
- Raiz da árvore ou subárvore?

Retirada de um Registro da Árvore

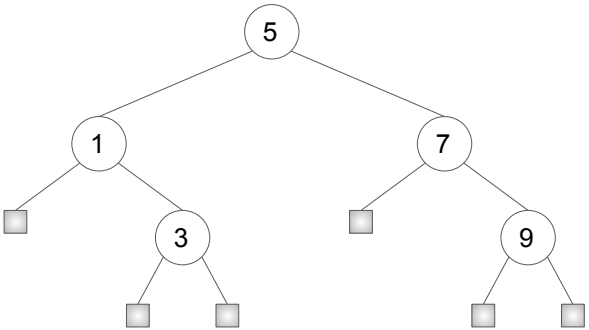
- Alguns comentários:
  1. A retirada de um registro não é tão simples quanto a inserção.
  2. Se o nó que contém o registro a ser retirado possuir no máximo um descendente $\Rightarrow$ a operação é simples.
  3. No caso do nó conter dois descendentes o registro a ser retirado deve ser primeiro:
     - substituído pelo registro mais à direita na subárvore esquerda.
     - ou pelo registro mais à esquerda na subárvore direita.

Retirada de um Registro da Árvore

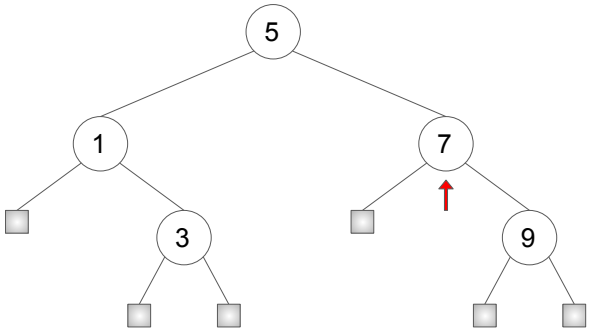
- Assim: para retirar o registro com chave 5 da árvore basta trocá-lo pelo registro com chave 4 ou pelo registro com chave 6, e então retirar o nó que recebeu o registro com chave 6.



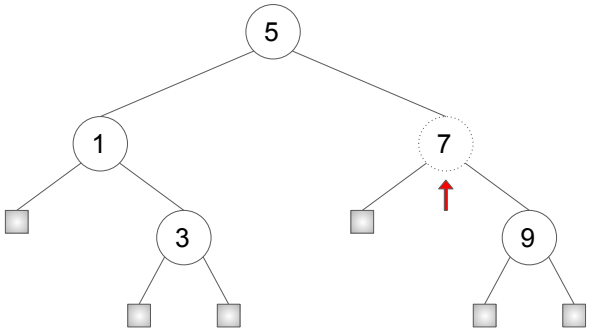
Exemplo de Retirada dos Nós 7 e 5



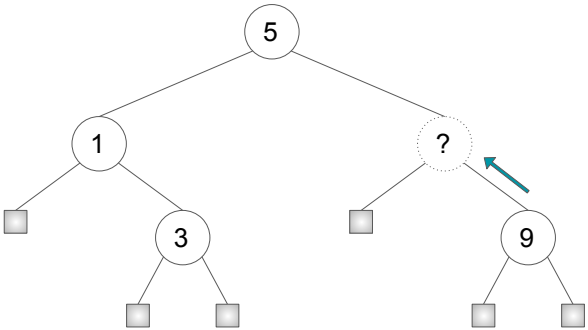
Exemplo de Retirada dos Nós 7 e 5



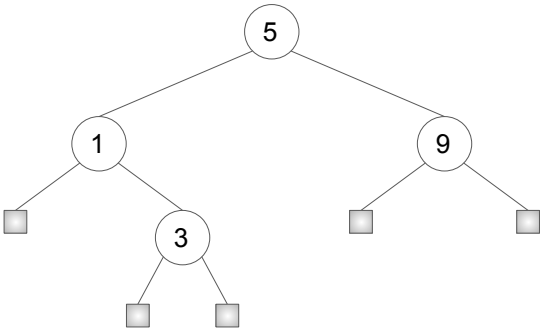
Exemplo de Retirada dos Nós 7 e 5



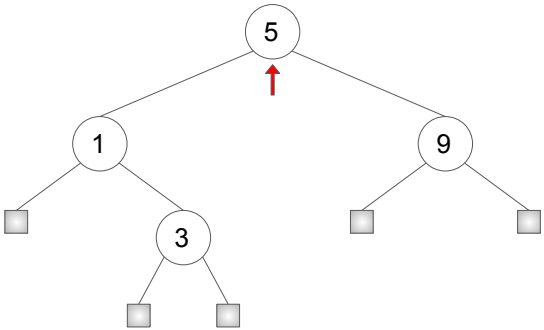
Exemplo de Retirada dos Nós 7 e 5



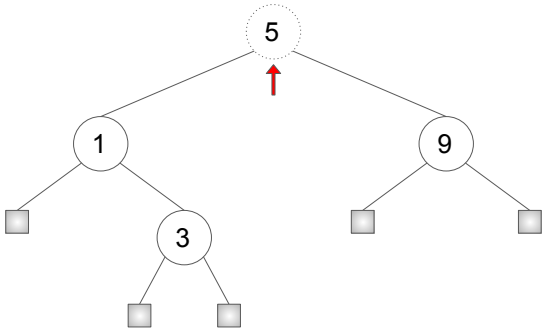
Exemplo de Retirada dos Nós 7 e 5



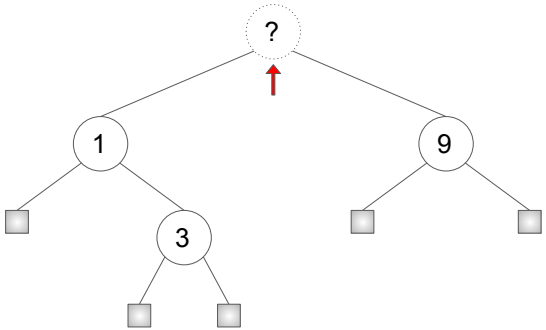
Exemplo de Retirada dos Nós 7 e 5



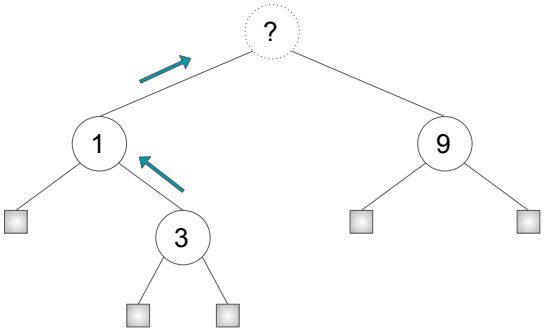
Exemplo de Retirada dos Nós 7 e 5



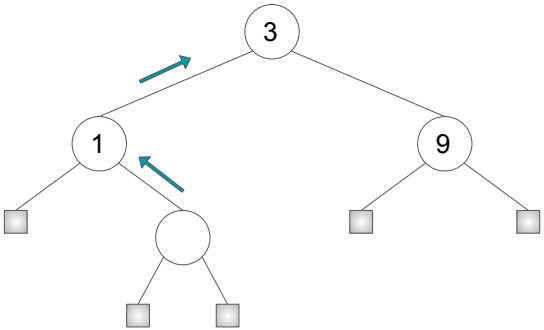
Exemplo de Retirada dos Nós 7 e 5



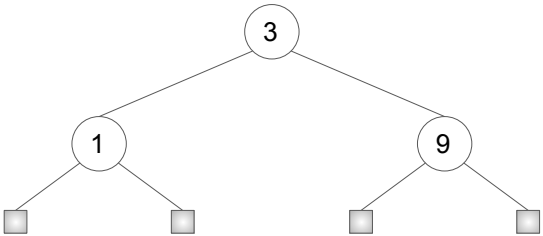
Exemplo de Retirada dos Nós 7 e 5



Exemplo de Retirada dos Nós 7 e 5



Exemplo de Retirada dos Nós 7 e 5



Retirada de Nó da Árvore

```
1  int TArvore_Retira(TNo **p, TItem x)
2  {
3      TNo *pAux;
4      if (*p == NULL)
5          return 0;
6
7      if (x.chave < (*p)->item.chave)
8          return TArvore_Retira(&((*p)->pEsq), x);
9      if (x.chave > (*p)->item.chave)
10         return TArvore_Retira(&((*p)->pDir), x);
11
12     if ((*p)->pEsq == NULL && (*p)->pDir == NULL) { // no eh folha
13         free(*p);
14         *p = NULL;
15         return 1;
16     }
17
18     ...
```

Retirada de Nó da Árvore

```
19      ...
20
21      if ((*p)->pEsq != NULL && (*p)->pDir == NULL) { // esq
22          pAux = *p;
23          *p = (*p)->pEsq;
24          free(pAux);
25          return 1;
26      }
27      if ((*p)->pDir != NULL && (*p)->pEsq == NULL) { // dir
28          pAux = *p;
29          *p = (*p)->pDir;
30          free(pAux);
31          return 1;
32      }
33
34      // no possui dois filhos
35      TArvore_Sucessor(*p, &((*p)->pDir));
36      // equivalente a TArvore_Antecessor(*p, &((*p)->pEsq));
37      return 1;
38  }
```

Retirada de Nó da Árvore

```
1  void TArvore_Sucessor (TNo *q, TNo **r) {
2      TNo *pAux;
3      if ((*r)->pEsq != NULL) {
4          TArvore_Sucessor(q, &(*r)->pEsq);
5          return;
6      }
7      q->item = (*r)->item;
8      pAux = *r;
9      *r = (*r)->pDir;
10     free(pAux);
11 }
```

Conteúdo

Caminhamento

- A ordem dos filhos dos nós em uma árvore pode ser ou não significativa.
  - Exemplos, no heap, a ordem dos filhos não tem significado.
  - Outros casos, pode se ter um significado (como é o caso em pesquisa em árvores binárias)
- Diversas formas de percorrer ou caminhar em uma árvore listando seus nós, as principais:
  - Pré-ordem (Pré-fixa).
  - Central (Infixa).
  - Pós-ordem (Pós-fixa).

Pré-Ordem

- Pré-ordem: lista o nó raiz, seguido de suas subárvores (da esquerda para a direita), cada uma em pré-ordem.

```
1 Algorithm: PRE_ORDEM
  Input: TNo n
2 begin
3     Lista(n)
4     for each child f of n from left to right do
5         PRE_ORDEM(f)
6     end
7 end
```

Pós-Ordem

- Pós-ordem: Lista os nós das subárvores (da esquerda para a direita) cada uma em pós-ordem, lista o nó raiz.

```
1 Algorithm: POS_ORDEM
  Input: TNo n
2 begin
3     for each child f of n from left to right do
4         POS_ORDEM(f)
5     end
6     Lista(n)
7 end
```

Caminhamento Pré-Ordem e Pós-Ordem

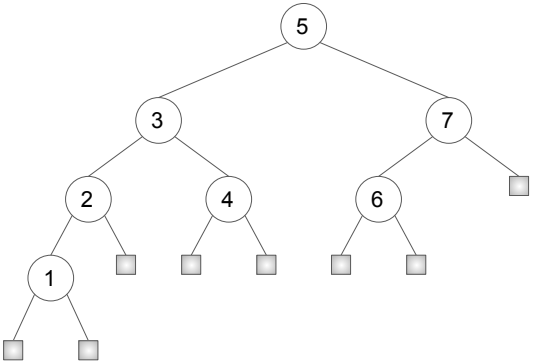
```
1  void PreOrdem(TNo *p) {
2      if (p == NULL)
3          return;
4      printf("%ld\n", p->item.chave);
5      PreOrdem(p->pEsq);
6      PreOrdem(p->pDir);
7  }
8
9
10 void PosOrdem(TNo *p) {
11     if (p == NULL)
12         return;
13     PosOrdem(p->pEsq);
14     PosOrdem(p->pDir);
15     printf("%ld\n", p->item.chave);
16 }
```

Central

- Em árvores de pesquisa a ordem de caminhamento mais útil é a chamada ordem de **caminhamento central**.
- O caminhamento central é mais bem expresso em termos recursivos:
  1. Caminha na subárvore esquerda na ordem central.
  2. Visita a raiz.
  3. Caminha na subárvore direita na ordem central.

Central

- Percorrer a árvore:



- Em que ordem chaves são recuperadas (usando caminhamento central)?

  1, 2, 3, 4, 5, 6 e 7.

Caminhamento Central

```
1 void Central(TNo *p) {
2     if (p == NULL)
3         return;
4     Central(p->pEsq);
5     printf("%ld\n", p->item.chave);
6     Central(p->pDir);
7 }
```

Conteúdo

Análise

- Comparações em uma pesquisa com sucesso:
  - Melhor caso: $C(n) = O(1)$.
  - Pior caso: $C(n) = O(n)$.
  - Caso médio: $C(n) = O(\log n)$.
- O tempo de execução dos algoritmos para árvores binárias de pesquisa dependem muito do formato das árvores, ou seja, se ela está balanceada ou não.

Análise

- Para obter o pior caso basta que as chaves sejam inseridas em ordem crescente ou decrescente. Neste caso a árvore resultante é uma lista linear, cujo número médio de comparações é $(n+1)/2$.
- Para uma árvore de pesquisa aleatória o número esperado de comparações para recuperar um registro qualquer é cerca de $1,39 \log n$, apenas 39% pior que a árvore completamente balanceada.

Vantagens e Desvantagens

- Vantagens:
  - Custo de pesquisa $O(\log n)$ para o caso médio.
  - Custo de inserção e remoção: $O(\log n)$ para o caso médio.
  - Custo para obter os registros em ordem: $O(n)$.
- Desvantagens:
  - Pior caso é $O(n)$ tanto para **pesquisa** quanto para **inserção** e **remoção**.

Conteúdo

Conclusão

- O uso de árvores favorecem pesquisas mais eficientes.
- O pior caso das árvores devem ser tratados.

Árvores de Pesquisa AVL

Conteúdo

Bibliografia

Os conteúdos deste material, incluindo figuras, textos e códigos, foram extraídos ou adaptados de:


Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford.
Introduction to Algorithms.
The MIT Press, 2011.

Exercício

- Dada a sequência de números: `10, 20, 5, 8, 12, 22, 23, 24, 11, 13, 18`.
- Mostre (desenhe) uma árvore binária de pesquisa após a inserção de cada um dos elementos acima.
- Mostre como ficará a árvore criada após a remoção dos seguintes elementos na seguinte ordem: `22, 11, 10`.