Notas de Aula - AED2 – Árvores: árvores B
Prof. Jefferson T. Oliva

Até o momento vocês viram métodos e estruturas para busca em memória primária, como: busca sequencial, busca binária, busca por interpolação, hashing, árvores binárias de busca, árvores AVL, árvores rubro-negras.

Hoje veremos uma estrutura de busca aplicada em memória secundária que pode ser utilizada em memória primária: árvores B

Árvores B

São árvores de pesquisa balanceadas projetadas para funcionar bem em discos magnéticos outros dispositivos de armazenamento secundário.

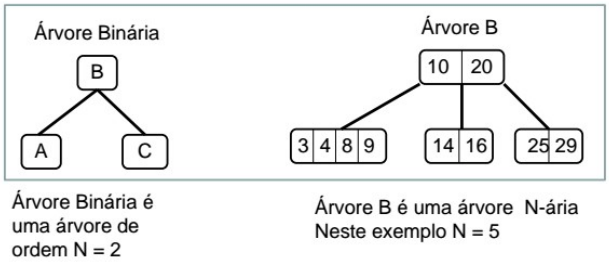
Essa estrutura possui diversas aplicações, como:
- Sistema de arquivos NTFS do Windows.
- Sistema de arquivos HFS do Mac.
- Sistema de arquivos ext4 do Linux.
- Bancos de dados, por exemplo, ORACLE, SQL e PostgreSQL.

Um nó em uma árvore B é também chamado de página: isso significa que um nó pode ter mais um elemento.

Ordem de uma árvore B:
- No livro de Cormen: número mínimo de filhos que uma árvore pode ter
- No livro de Knuth: número máximo de nós filhos.

Nessa disciplina, consideraremos a definição de Knuth, para mantermos a coerência com as árvores binárias (cada nó pode ter até 2 subárvores) apresentadas até agora.

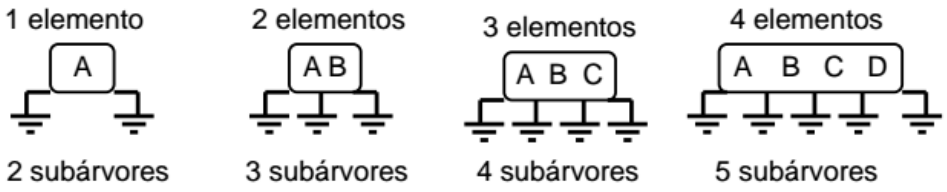
Uma árvore B de ordem N é uma árvore de busca multidirecional balanceada que satisfaz as seguintes condições
- todo nó possui N ou menos subárvores.
- todo nó, exceto o raiz e os folhas, possui no mínimo N / 2 subárvores (maior inteiro) e no máximo, N.
- o nó raiz possui no mínimo duas subárvores não vazias e, no máximo, N subárvores. Assim, a raiz possui no mínimo um elemento e no máximo N – 1 elementos.
- **todas as folhas estão no mesmo nível**
- **um nó não folha com k subárvores armazena k - 1 registros**
- **um nó folha armazena no máximo N - 1 e no mínimo ceil(N / 2) – 1 registros. A raiz possui, no mínimo, um elemento.**
- todos os nós pai (de derivação) possuem exclusivamente subárvores não vazias

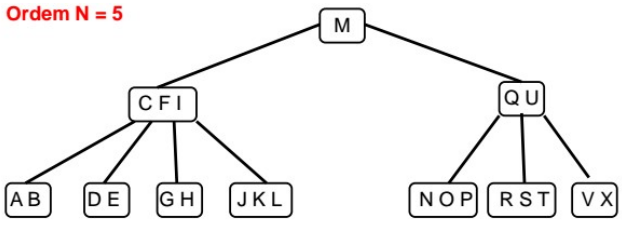


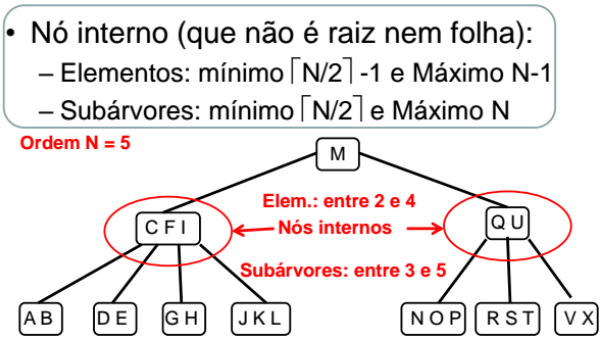
Raiz:
– Elementos: mínimo 1 e máximo N-1
– Subárvores: mínimo duas e no máximo N;

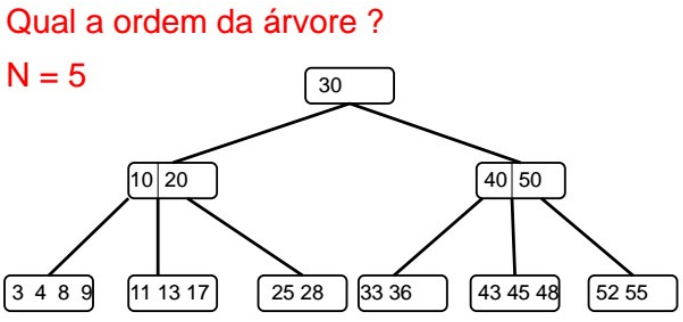
Nó interno: (que não é raiz nem folha):
– Elementos: mínimo round(N/2) -1 e Máximo N-1
– Subárvores: mínimo round(N/2) e Máximo N

Folhas:
– Todas as folhas estão no mesmo nível.









Estrutura para uma árvore B:

```
#define N ? /*ordem da árvore B*/

typedef struct NodeB NodeB;

struct NodeB{
    int nro_chaves;
    int chaves[N - 1];
    NodeB *filhos[N];
    int eh_no_folha;
};


int criar(NodeB *tree){
   int i;

   if (tree == NULL){
      tree = malloc(sizeof(NodeB));
      tree->eh_no_folha = 1;
      tree->nro_chaves = 0;

      for (i = 0; i < N; i++)
         tree->filhos[i] = NULL;

      return 1;
   }

   return 0;
}

int liberar(NodeB *tree){
   if (tree != NULL){
      free(tree);

      return 1;
   }

   return 0;
}
```

Pesquisa em Árvore B

Semelhante a busca em árvores binárias de busca (incluindo as AVL e as rubro-negras)

Para minimizar o custo da busca em cada página (nó), pode ser feita, inicialmente, uma busca binária na página. Caso o item não seja encontrado, a busca é continuada em um dos nós filhos.

```
// encontrar a posição da chave em um determinado nó
int busca_binaria(int key, NodeB *tree){
   int ini, fim, meio;

   if (tree != NULL){
      ini = 0;
      fim = tree->nro_chaves - 1;

      while (ini <= fim){
         meio = (ini + fim) / 2;

         if (tree->chaves[meio] == key)
            return meio; // Aqui, a chave foi encontrada
         else if (tree->chaves[meio] < key)
            ini = meio + 1;
         else
            fim = meio - 1;
      }

      return ini; // A chave não foi encontrada. Neste caso, o ini é posição do filho (ponteiro para filho)
   }

   return -1;
}


int pesquisar(int key, NodeB *tree){
   int pos = busca_binaria(key, tree);

   if (pos >= 0){
      if (tree->chaves[key] == key)
         return 1;
      else
         return pesquisar(key, tree->filhos[pos]);
   }

   return 0;
}
```

Complexidade de tempo? O(log_2 n): log_2 N (**log de N na base 2**) * log_N M (**log de M na base N**) = log_2 M. Se log_c a / log_c b = log_b a, então log_c b * log_b a = log_c a

Onde: N é a ordem da árvore B e M é quantidade total de chaves na árvore B.

Para busca de um nó, também pode ser utilizada a pesquisa sequencial:

```
int pesquisaSequencial(int key, NodeB *tree){
   int i;

   if (tree != NULL){
      for (i = 0; i < tree->nro_chaves && key < tree->chaves[i]; i++)
         if (key == tree->chaves[i])
            return 1;

      return pesquisaSequencial(key, tree->filhos[i]);
   }

   return 0;
}
```

Essa última solução possui uma complexidade de tempo na ordem de O(N * log_N M)

A complexidade de espaço para ambas abordagens de pesquisa é de ordem de Theta(M).

Inserção em árvore B

Diferentemente de uma árvore binária, em uma árvore B não podemos simplesmente criar um novo nó (página)

Começa com uma busca, a partir da página raiz, que continua até localizar a página onde deve ser inserido o novo elemento.

Se a página onde o elemento deve ser inserido tiver menos de N - 1 elementos, este é alocado nesta página. Caso contrário, pode ser necessário a criação de uma nova página. Para isso, a página é dividida em duas páginas.

A criação de uma nova página é feita nos seguintes passos:
1 - Primeiramente escolhe-se uma chave intermediária na página, considerando a nova chave que deverá ser inserida.
2 - Em seguida, uma nova página é criada, na qual serão posicionados os valores maiores que a chave intermediária e as menores permanecerão na página que foi dividida. Essa operação é denominada split.
3 – A chave intermediária é inserida na página pai, a qual pode estar cheia, sendo necessária uma nova divisão, na qual é criada uma outra página. O processo de criação de nova página pode ser propagada até a página raiz. Se não existir uma página pai durante a divisão, uma nova página deverá ser criada para ser a raiz.

```
// dividir página em duas
NodeB* split_pag(NodeB *pai, int posF_cheio){
   int i;
   NodeB *pag_esq = pai->filhos[posF_cheio];
   NodeB *pag_dir;

   criar(pag_dir);

   pag_dir->eh_no_folha = pag_esq->eh_no_folha;

   //Atualizar a quantidade de chaves nas páginas.
   pag_dir->nro_chaves = round((N - 1) / 2);
   pag_esq->nro_chaves = (N - 1) / 2;

   // preencher a nova página
   for (i = 0; i < pag_dir->nro_chaves; i++)
      pag_dir->chaves[i] = pag_esq->chaves[i + pag_esq->nro_chaves];

   // se a página esquerda não for nó-folha, a página direita deve herdar os seus respectivos descendentes
   if (!pag_esq->eh_no_folha)
      for (i = 0; i <= pag_dir->nro_chaves; i++)
         pag_dir->filhos[i] = pag_esq->filhos[i + pag_esq->nro_chaves];

   // Como a página pai já foi dividida anteriormente ateriormente, essa operação faz sentido
   // Os descendentes da página pai são deslocados em uma posição para a adição da nova página descendente
   for (i = pai->nro_chaves + 1; i > posF_cheio + 1; i--)
      pai->filhos[i + 1] = pai->filhos[i];

   pai->filhos[posF_cheio + 1] = pag_dir;

   for (i = pai->nro_chaves + 1; i > posF_cheio; i--)
      pai->chaves[i + 1] = pai->chaves[i];

   // promoção da maior chave da página esquerda (anteriormente, era a posição média)
   pai->chaves[posF_cheio] = pag_esq->chaves[(N - 1) / 2];

   pai->nro_chaves++;
}

void inserir_pagina_nao_cheia(NodeB *tree, int key){
   int i, pos = busca_binaria(key, tree); //encontrar a posição para adicionar a página

   if (tree->eh_no_folha){
      for (i = tree->nro_chaves; i > pos; i--)
         tree->chaves[i] = tree->chaves[i - 1];
      tree->chaves[i] = key;
      tree->nro_chaves++;
   }else{
      if (tree->filhos[pos]->nro_chaves == N - 1)
         split_pag(tree, pos);

         if (key > tree->chaves[pos])
            pos++;

         inserir_pagina_nao_cheia(tree->filhos[pos], key);
   }
}
```

```
void inserir(NodeB *tree, int key){
   NodeB *aux = tree;
   NodeB *nova_pag;

   if (aux->nro_chaves == N - 1){
      criar(nova_pag);
      tree = nova_pag;

      nova_pag->eh_no_folha = 0;

      nova_pag->chaves[0] = aux;

      split_pag(nova_pag, 0);
      inserir_pagina_nao_cheia(nova_pag, key);
   }else
      inserir_pagina_nao_cheia(nova_pag, key);
}
```

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

Marin, L. O. Árvores B. AE23CP - Algoritmos e Estrutura de Dados II. Slides. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2017.

Ziviani, N. Projeto de Algoritmos - com implementações em Java e C++. Thomson, 2007.